Lecture Notes in Computer Science 3503

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison
*Lancaster University, UK*

Takeo Kanade
*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler
*University of Surrey, Guildford, UK*

Jon M. Kleinberg
*Cornell University, Ithaca, NY, USA*

Friedemann Mattern
*ETH Zurich, Switzerland*

John C. Mitchell
*Stanford University, CA, USA*

Moni Naor
*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz
*University of Bern, Switzerland*

C. Pandu Rangan
*Indian Institute of Technology, Madras, India*

Bernhard Steffen
*University of Dortmund, Germany*

Madhu Sudan
*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos
*New York University, NY, USA*

Doug Tygar
*University of California, Berkeley, CA, USA*

Moshe Y. Vardi
*Rice University, Houston, TX, USA*

Gerhard Weikum
*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Sotiris E. Nikoletseas (Ed.)

# Experimental and Efficient Algorithms

4th International Workshop, WEA 2005
Santorini Island, Greece, May 10-13, 2005
Proceedings

Springer

Volume Editor

Sotiris E. Nikoletseas
University of Patras and Computer Technology Institute (CTI)
61 Riga Fereou Street, 26221 Patras, Greece
E-mail: nikole@cti.gr

Library of Congress Control Number: 2005925473

CR Subject Classification (1998): F.2.1-2, E.1, G.1-2, I.3.5, I.2.8

ISSN 0302-9743
ISBN-10 3-540-25920-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25920-6 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11427186 06/3142 5 4 3 2 1 0

# Preface

This proceedings volume contains the accepted papers and invited talks presented at the 4th International Workshop of Efficient and Experimental Algorithms (WEA 2005), that was held May 10–13, on Santorini Island, Greece.

The WEA events are intended to be an international forum for research on the design, analysis and especially the experimental implementation, evaluation and engineering of algorithms, as well as on combinatorial optimization and its applications.

The first three workshops in this series were held in Riga (2001), Monte Verita (2003) and Rio de Janeiro (2004).

This volume contains 3 invited papers related to corresponding keynote talks: by Prof. Christos Papadimitriou (University of California at Berkeley, USA), Prof. David Bader (University of New Mexico, USA) and Prof. Celso Ribeiro (University of Rio de Janeiro, Brazil).

This proceedings includes 54 papers (47 regular and 7 short), selected out of a record number of 176 submissions. Each paper was reviewed by at least 2 Program Committee members, while many papers got 3 or 4 reviews. A total number of 419 reviews were solicited, with the help of trusted external referees.

In addition to the 54 papers included in this volume, 6 papers were accepted as poster presentations: these papers were published in a separate poster proceedings volume by CTI Press and a major publisher in Greece, "Ellinika Grammata." The presentation of these posters at the event was expected to create a fruitful discussion on interesting ideas.

The 60 papers accepted to WEA 2005 demonstrate the international character of the event: 33 authors are based in Germany, 20 in the USA, 13 in Italy, 12 in Greece, 9 each in Switzerland, France and Brazil, 6 each in Canada, Poland and Belgium, 5 in the Netherlands, to list just the countries with the largest participations.

Selected papers of WEA 2005 will be considered for a Special Issue of the ACM Journal on Experimental Algorithmics (JEA, http://www.jea.acm.org/) dedicated to the event.

We would like to thank all authors who submitted papers to WEA 2005. We especially thank the distinguished invited speakers (whose participation honors the event a lot), and the members of the Program Committee, as well as the external referees and the Organizing Committee members.

We would like to thank the Ministry of National Education and Religious Affairs of Greece for its financial support of the event. Also, we gratefully acknowledge the support from the Research Academic Computer Technology Institute (RACTI, Greece, http://www.cti.gr), and the European Union (EU) IST/FET (Future and Emerging Technologies) R&D projects FLAGS (Foundational As-

pects of Global Computing Systems) and DELIS (Dynamically Evolving, Large-Scale Information Systems).

I wish to personally acknowledge the great job of the WEA 2005 Publicity Chair Dr. Ioannis Chatzigiannakis, and Athanasios Kinalis for maintaining the Web page and processing this volume with efficiency and professionalism.

I am grateful to the WEA Steering Committee Chairs Prof. Jose Rolim and Prof. Klaus Jansen for their trust and support.

Finally, we wish to thank Springer Lecture Notes in Computer Science (LNCS), and in particular Alfred Hofmann and his team, for a very nice and efficient co-operation in preparing this volume.

May 2005 Sotiris Nikoletseas

# Organization

## Program Committee Chair

| | |
|---|---|
| Sotiris Nikoletseas | University of Patras and CTI, Greece |

## Program Committee

| | |
|---|---|
| Edoardo Amaldi | Politecnico di Milano, Italy |
| Evripidis Bampis | Université d'Evry, France |
| David A. Bader | University of New Mexico, USA |
| Cynthia Barnhart | MIT, USA |
| Azzedine Boukerche | SITE, University of Ottawa, Canada |
| Gerth Brodal | University of Aarhus, Denmark |
| Rainer Burkard | Graz University of Technology, Austria |
| Giuseppe Di Battista | Universita' degli Studi Roma Tre, Italy |
| Rudolf Fleischer | Fudan University, Shanghai, China |
| Pierre Fraigniaud | CNRS, Université Paris-Sud, France |
| Mark Goldberg | Rensselaer Polytechnic Institute, USA |
| Juraj Hromkovic | ETH Zurich, Switzerland |
| Giuseppe Italiano | Universita' di Roma Tor Vergata, Italy |
| Christos Kaklamanis | University of Patras and CTI, Greece |
| Helen Karatza | Aristotle University of Thessaloniki, Greece |
| Ludek Kucera | Charles University, Czech Republic |
| Shay Kutten | Technion - Israel Institute of Technology, Israel |
| Catherine McGeoch | Amherst College, USA |
| Simone Martins | Universidade Federal Fluminense, Brazil |
| Bernard Moret | University of New Mexico, USA |
| Ian Munro | University of Waterloo, Canada |
| Sotiris Nikoletseas | University of Patras and CTI, Greece (Chair) |
| Andrea Pietracaprina | University of Padova, Italy |
| Tomasz Radzik | King's College London, UK |
| Rajeev Raman | University of Leicester, UK |
| Mauricio Resende | AT&T Labs Research, USA |
| Maria Serna | T.U. of Catalonia, Spain |
| Paul Spirakis | University of Patras and CTI, Greece |
| Eric Taillard | EIVD, Switzerland |
| Dorothea Wagner | University of Karlsruhe, Germany |
| Stefan Voss | University of Hamburg, Germany |
| Christos Zaroliagis | University of Patras and CTI, Greece |

## Steering Committee Chairs

| | |
|---|---|
| Klaus Jansen | University of Kiel, Germany |
| Jose Rolim | University of Geneva, Switzerland |

## Organizing Committee

| | |
|---|---|
| Ioannis Chatzigiannakis | CTI, Greece, (Co-chair) |
| Rozina Efstathiadou | CTI, Greece, (Co-chair) |
| Lena Gourdoupi | CTI, Greece |
| Athanasios Kinalis | University of Patras and CTI, Greece |

## Referees

Nazim Agoulmine
Roberto Aringhieri
Yossi Azar
Ricardo Baeza-Yates
Michael Baur
Amos Beimel
Pietro Belotti
Alberto Bertoldo
Mauro Bianco
Maria Blesa
Roderick Bloem
Christian Blum
Maria Cristina Boeres
Thomas Buchholz
Costas Busch
Sergiy Butenko
Roberto Wolfler Calvo
Antonio Capone
Ioannis Caragiannis
Massimiliano Caramia
Matteo Cesana
Ioannis Chatzigiannakis
Yinong Chen
Francis Chin
Pier Francesco Cortese
Yves Crama
Cid de Souza
Josep Diaz
Tassos Dimitriou
Rolf Fagerberg
Carlo Fantozzi
Antonio Fernández
Irene Finocchi
Dimitris Fotakis
Joaquim Gabarró
Marco Gaertler
Giulia Galbiati
Clemente Galdi
Giovanni Gallo
Efstratios Gallopoulos
Fabrizio Grandoni
Peter Greistorfer
Nir Halman
Refael Hassin
Martin Holzer
Ja Hoogeveen
Stanislaw Jarecki
Jiang Jun
Sam Kamin
Howard Karloff
Dukwon Kim
Athanasios Kinalis
Sigrid Knust
Elisavet Konstantinou
Charalambos Konstantopoulos
Spyros Kontogiannis
Dimitrios Koukopoulos
Joachim Kupke
Giovanni Lagorio
Giuseppe Lancia
Carlile Lavor
Helena Leityo
Zvi Lotker
Abilio Lucena
Francesco Maffioli
Malik Magdon-Ismail
Christos Makris
Federico Malucelli
Carlos Alberto Martinhon
Constandinos Mavromoustakis
Steffen Mecke
John Mitchell
Ivone Moh
Gabriel Moruz
Pablo Moscato
Matthias Mueller-Hannemann
Maurizio Naldi
Filippo Neri
Sara Nicoloso
Gaia Nicosia
Mustapha Nourelfath
Carlos A.S. Oliveira
Mohamed Ould-Khaoua

Andrea Pacifici
Evi Papaioannou
Panos M. Pardalos
Paolo Penna
Pino Persiano
Enoch Peserico
Jordi Petit
Ugo Pietropaoli
Mustafa Pinar
Evaggelia Pitoura
Maurizio Pizzonia
Alexandre Plastino
Daniele Pretolani
Daniela Pucci de Farias
Naila Rahman
Massimo Rimondini
Isabel Rosseti
Harilaos Sandalidis
Haroldo Santos
Thomas Schank
Elad Schiller
Frank Schulz
Sebastian Seibert
Spyros Sioutas
Spiros Sirmakessis
Riste Skrekovski
Stênio Soares
Yannis Stamatiou
Maurizio Strangio
Tami Tamir
Leandros Tassiulas
Dimitrios M. Thilikos
Marco Trubian
Manolis Tsagarakis
George Tsaggouris
Gabriel Wainer
Renato Werneck
Igor Zwir

## Sponsoring Institutions


- Ministry of National Education and Religious Affairs of Greece
- Research Academic Computer Technology Institute (RACTI), Greece
- EU-FET R&D project "Foundational Aspects of Global Computing Systems" (FLAGS)
- EU-FET R&D project "Dynamically Evolving, Large-Scale Information Systems" (DELIS)

# Table of Contents

## Invited Talks

*Τα Παιδία Παίζει* The Interaction Between Algorithms and Game Theory . . . . . 1
*Christos H. Papadimitriou*

Using an Adaptive Memory Strategy to Improve a Multistart Heuristic for Sequencing by Hybridization
*Eraldo R. Fernandes, Celso C. Ribeiro* . . . . . 4

High-Performance Algorithm Engineering for Large-Scale Graph Problems and Computational Biology
*David A. Bader* . . . . . 16

## Contributed Regular Papers

The "Real" Approximation Factor of the MST Heuristic for the Minimum Energy Broadcasting
*Michele Flammini, Alfredo Navarra, Stephane Perennes* . . . . . 22

Implementing Minimum Cycle Basis Algorithms
*Kurt Mehlhorn, Dimitrios Michail* . . . . . 32

Rounding to an Integral Program
*Refael Hassin, Danny Segev* . . . . . 44

Rectangle Covers Revisited Computationally
*Laura Heinrich-Litan, Marco E. Lübbecke* . . . . . 55

Don't Compare Averages
*Holger Bast, Ingmar Weber* . . . . . 67

Experimental Results for Stackelberg Scheduling Strategies
*A.C. Kaporis, L.M. Kirousis, E.I. Politopoulou, P.G. Spirakis* . . . . . 77

An Improved Branch-and-Bound Algorithm for the Test Cover Problem
*Torsten Fahle, Karsten Tiemann* . . . . . 89

Degree-Based Treewidth Lower Bounds
*Arie M.C.A. Koster, Thomas Wolle, Hans L. Bodlaender* . . . . . 101

Inferring AS Relationships: Dead End or Lively Beginning?
*Xenofontas Dimitropoulos, Dmitri Krioukov, Bradley Huffaker, kc claffy, George Riley* ..... 113

Acceleration of Shortest Path and Constrained Shortest Path Computation
*Ekkehard Köhler, Rolf H. Möhring, Heiko Schilling* ..... 126

A General Buffer Scheme for the Windows Scheduling Problem
*Amotz Bar-Noy, Jacob Christensen, Richard E. Ladner, Tami Tamir* ..... 139

Implementation of Approximation Algorithms for the Multicast Congestion Problem
*Qiang Lu, Hu Zhang* ..... 152

Frequency Assignment and Multicoloring Powers of Square and Triangular Meshes
*Mustapha Kchikech, Olivier Togni* ..... 165

From Static Code Distribution to More Shrinkage for the Multiterminal Cut
*Bram De Wachter, Alexandre Genon, Thierry Massart* ..... 177

Partitioning Graphs to Speed Up Dijkstra's Algorithm
*Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, Thomas Willhalm* ..... 189

Efficient Convergence to Pure Nash Equilibria in Weighted Network Congestion Games
*Panagiota N. Panagopoulou, Paul G. Spirakis* ..... 203

New Upper Bound Heuristics for Treewidth
*Emgad H. Bachoore , Hans L. Bodlaender* ..... 216

Accelerating Vickrey Payment Computation in Combinatorial Auctions for an Airline Alliance
*Yvonne Bleischwitz, Georg Kliewer* ..... 228

Algorithm Engineering for Optimal Graph Bipartization
*Falk Hüffner* ..... 240

Empirical Analysis of the Connectivity Threshold of Mobile Agents on the Grid
*Xavier Pérez* ..... 253

Multiple-Winners Randomized Tournaments with Consensus for Optimization Problems in Generic Metric Spaces
*Domenico Cantone, Alfredo Ferro, Rosalba Giugno, Giuseppe Lo Presti, Alfredo Pulvirenti* ..... 265

On Symbolic Scheduling Independent Tasks with Restricted Execution Times
*Daniel Sawitzki* ..... 277

A Simple Randomized $k$-Local Election Algorithm for Local Computations
*Rodrigue Ossamy* ..... 290

Generating and Radiocoloring Families of Perfect Graphs
*M.I. Andreou, V.G. Papadopoulou, P.G. Spirakis, B. Theodorides, A. Xeros* ..... 302

Efficient Implementation of Rank and Select Functions for Succinct Representation
*Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, Kunsoo Park* ..... 315

Comparative Experiments with GRASP and Constraint Programming for the Oil Well Drilling Problem
*Romulo A. Pereira, Arnaldo V. Moura, Cid C. de Souza* ..... 328

A Framework for Probabilistic Numerical Evaluation of Sensor Networks: A Case Study of a Localization Protocol
*Pierre Leone, Paul Albuquerque, Christian Mazza, Jose Rolim* ..... 341

A Cut-Based Heuristic to Produce Almost Feasible Periodic Railway Timetables
*Christian Liebchen* ..... 354

GRASP with Path-Relinking for the Weighted Maximum Satisfiability Problem
*Paola Festa, Panos M. Pardalos, Leonidas S. Pitsoulis, Mauricio G.C. Resende* ..... 367

New Bit-Parallel Indel-Distance Algorithm
*Heikki Hyyrö, Yoan Pinzon, Ayumi Shinohara* ..... 380

Dynamic Application Placement Under Service and Memory Constraints
*Tracy Kimbrel, Malgorzata Steinder, Maxim Sviridenko, Asser Tantawi* ..... 391

Integrating Coordinated Checkpointing and Recovery Mechanisms into DSM Synchronization Barriers
*Azzedine Boukerche, Jeferson Koch, Alba Cristina Magalhaes Alves de Melo* ..... 403

Synchronization Fault Cryptanalysis for Breaking A5/1
*Marcin Gomułkiewicz, Mirosław Kutyłowski, Heinrich Theodor Vierhaus, Paweł Wlaź* ..... 415

An Efficient Algorithm for $\delta$-Approximate Matching with $\alpha$-Bounded Gaps in Musical Sequences
*Domenico Cantone, Salvatore Cristofaro, Simone Faro* ..... 428

The Necessity of Timekeeping in Adversarial Queueing
*Maik Weinard* ..... 440

BDDs in a Branch and Cut Framework
*Bernd Becker, Markus Behle, Friedrich Eisenbrand, Ralf Wimmer* ... 452

Parallel Smith-Waterman Algorithm for Local DNA Comparison in a Cluster of Workstations
*Azzedine Boukerche, Alba Cristina Magalhaes Alves de Melo, Mauricio Ayala-Rincon, Thomas M. Santana* ..... 464

Fast Algorithms for Weighted Bipartite Matching
*Justus Schwartz, Angelika Steger, Andreas Weißl* ..... 476

A Practical Minimal Perfect Hashing Method
*Fabiano C. Botelho, Yoshiharu Kohayakawa, Nivio Ziviani* ..... 488

Efficient and Experimental Meta-heuristics for MAX-SAT Problems
*Dalila Boughaci, Habiba Drias* ..... 501

Experimental Evaluation of the Greedy and Random Algorithms for Finding Independent Sets in Random Graphs
*M. Goldberg, D. Hollinger, M. Magdon-Ismail* ..... 513

Local Clustering of Large Graphs by Approximate Fiedler Vectors
*Pekka Orponen, Satu Elisa Schaeffer* ..... 524

Almost FPRAS for Lattice Models of Protein Folding
*Anna Gambin, Damian Wójtowicz* ..... 534

Vertex Cover Approximations: Experiments and Observations
*Eyjolfur Asgeirsson, Cliff Stein* ..... 545

GRASP with Path-Relinking for the Maximum Diversity Problem
*Marcos R.Q. de Andrade, Paulo M.F. de Andrade, Simone L. Martins, Alexandre Plastino* . . . . . 558

How to Splay for loglogN-Competitiveness
*George F. Georgakopoulos* . . . . . 570

Distilling Router Data Analysis for Faster and Simpler Dynamic IP Lookup Algorithms
*Filippo Geraci, Roberto Grossi* . . . . . 580

## Contributed Short Papers

Optimal Competitive Online Ray Search with an Error-Prone Robot
*Tom Kamphans, Elmar Langetepe* . . . . . 593

An Empirical Study for Inversions-Sensitive Sorting Algorithms
*Amr Elmasry, Abdelrahman Hammad* . . . . . 597

Approximation Algorithm for Chromatic Index and Edge-Coloring of Multigraphs
*Martin Kochol, Naďa Krivoňáková, Silvia Smejová* . . . . . 602

Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study
*Thomas Schank, Dorothea Wagner* . . . . . 606

Selecting the Roots of a Small System of Polynomial Equations by Tolerance Based Matching
*H. Bekker, E.P. Braad, B. Goldengorin* . . . . . 610

Developing Novel Statistical Bandwidths for Communication Networks with Incomplete Information
*Janos Levendovszky, Csego Orosz* . . . . . 614

Dynamic Quality of Service Support in Virtual Private Networks
*Yuxiao Jia, Dimitrios Makrakis, Nicolas D. Georganas, Dan Ionescu* . . . . . 618

**Author Index** . . . . . 623

# *Τα Παιδία Παίζει*
# The Interaction Between Algorithms and Game Theory*

Christos H. Papadimitriou

UC Berkeley
christos@cs.berkeley.edu

The theories of algorithms and games were arguably born within a year of each other, in the wake of two quite distinct breakthroughs by John von Neumann, in the former case to investigate the great opportunities – as well as the ever mysterious obstacles – in attacking problems by computers, in the latter to model and study rational selfish behavior in the context of interaction, competition and cooperation. For more than half a century the two fields advanced as gloriously as they did separately. There was, of course, a tradition of computational considerations in equilibria initiated by Scarf [13], work on computing Nash and other equilibria [6, 7], and reciprocal isolated works by algorithms researchers [8], as well as two important points of contact between the two fields *à propos* the issues of repeated games and bounded rationality [15] and learning in games [2]. But the current intensive interaction and cross-fertilization between the two disciplines, and the creation of a solid and growing body of work at their interface, must be seen as *a direct consequence of the Internet*.

By enabling rapid, well-informed interactions between selfish agents (as well as by being itself the result of such interactions), and by creating new kinds of markets (besides being one itself), the Internet challenged economists, and especially game theorists, in new ways. At the other bank, computer scientists were faced for the first time with a mysterious artifact that was not designed, but had emerged in complex, unanticipated ways, and had to be approached with the same puzzled humility with which other sciences approach the cell, the universe, the brain, the market. Many of us turned to Game Theory for enlightenment.

The new era of research in the interface between Algorithms and Game Theory is rich, active, exciting, and fantastically diverse. Still, one can discern in it three important research directions: *Algorithmic mechanism design, the price of anarchy*, and *algorithms for equilibria*.

If mainstream Game Theory models rational behavior in competitive settings, *Mechanism Design* (or *Reverse Game Theory*, as it is sometimes called) seeks to create games (auctions, for example) in which selfish players will behave in ways conforming to the designers objectives. This modern but already

* Research supported by NSF ITR grant CCR-0121555 and by a grant from Microsoft Research. The title phrase, a Greek version of "games children play", is a common classroom example of a syntactic peculiarity (singular verb form with neutral plural subject) in the Attic dialect of ancient Greek.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 1–3, 2005.
© Springer-Verlag Berlin Heidelberg 2005

mathematically well-developed branch of Game Theory received a shot in the arm by the sudden influx of computational ideas, starting with the seminal paper [9]. Computational Mechanism Design is a compelling research area for both sides of the fence: Several important classical existence theorems in Mechanism Design create games that are very complex, and can be informed and clarified by our fields algorithmic and complexity-theoretic ideas; it presents a new genre of interesting algorithmic problems; and the Internet is an attractive theater for incentive-based design, including auction design.

Traditionally, distributed systems are designed centrally, presumably to optimize the sum total of the users objectives. The Internet exemplified another possibility: A distributed system can also be designed by the interaction of its users, each seeking to optimize his/her own objective. Selfish design has advantages of architectural and political nature, while central design obviously results in better overall performance. The question is, how much better? The *price of anarchy* is precisely the ratio of the two. In game-theoretic terms, it is the ratio of the sum of player payoffs in the worst (or best) equilibrium, divided by the payoff sum of the strategy profile that maximizes this sum. This line of investigation was initiated in [5] and continued by [11] and many others. That economists and game theorists had not been looking at this issue is surprising but not inexplicable: In Economics central design is not an option; in Computer Science it has been the default, a golden standard that invites comparisons. And computer scientists have always thought in terms of ratios (in contrast, economists favor the difference or "regret"): The approximation ratio of a hard optimization problem [14] can be thought of as the price of complexity; the competitive ratio in an on-line problem [4] is the price of ignorance, of lack of clairvoyance; in this sense, the price of anarchy had been long in coming.

This sudden brush with Game Theory made computer scientists aware of an open algorithmic problem: *Is there a polynomial-time algorithm for finding a mixed Nash equilibrium in a given game?* Arguably, and together with factoring, this is the most fundamental open problem in the boundary of P and NP: Even the 2-player case is open – we recently learned [12] of certain exponential examples to the pivoting algorithm of Lemke and Howson [6]. Even though some game theorists are still mystified by our fields interest efficient algorithms for finding equilibria (a concept that is not explicitly computational), many more are starting to understand that the algorithmic issue touches on the foundations of Game Theory: An intractable equilibrium concept is a poor model and predictor of player behavior. In the words of Kamal Jain "If your PC cannot find it, then neither can the market". Research in this area has been moving towards games with many players [3, 1]), necessarily under some succinct representation of the utilities (otherwise the input would need to be astronomically large), recently culminating in a polynomial-time algorithm for computing correlated equilibria (a generalization of Nash equilibrium) in a very broad class of multiplayer games [10].

## References


1. Fabrikant, A., Papadimitriou, C., Talwar, K.: The Complexity of Pure Nash equilibria. STOC (2004)
2. Fudenberg, D., Levine, D. K.: Theory of Learning in Games. MIT Press, (1998)
3. Kearns, M., Littman, M., Singh, S.: Graphical Models for Game Theory. Proceedings of the Conference on Uncertainty in Artificial Intelligence, (2001) 253–260
4. Koutsoupias, E., Papadimitriou, C. H.: On the $k$-Server Conjecture. JACM 42(5), (1995), 971–983
5. Koutsoupias, E., Papadimitriou, C. H.: Worst-case Equilibria. Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science, (1999)
6. Lemke, C. E., Howson, J. T.: Equilibrium Points of Bimatrix Games. Journal of the Society of Industrial and Applied Mathematics, 12, (1964), 413–423
7. McKelvey, R., McLennan, A.: Computation of Equilibria in Finite Games. In the Handbook of Computation Economics, Vol. I. Elsevier, Eds. Amman, H., Kendrick,D. A., Rust, J. (1996) 87–142
8. Megiddo, N.: Computational Complexity of the Game Theory Approach to Cost Allocation on a Tree. Mathematics of Operations Research 3, (1978) 189–196
9. Nisan, N., Ronen, A.: Algorithmic Mechanism Design. Games and Economic Behavior, 35, (2001) 166–196
10. Papadimitriou, C.H.: Computing Correlated Equilibria in Multiplayer Games. STOC (2005)
11. Roughgarden, T., Tardos, .: How Bad is Selfish Routing? JACM 49, 2, (2002) 236–259
12. Savani, R., von Stengel, B.: Long Lemke-Howson Paths. FOCS (2004)
13. Scarf, H.: The Computation of Economic Equilibria. Yale University Press, (1973)
14. Vazirani, V. V.: Approximation Algorithms. Springer-Verlag, (2001)
15. Papadimitriou, Christos H., Yannakakis, M.: On Complexity as Bounded Rationality (extended abstract). STOC (1994) 726–733

# Using an Adaptive Memory Strategy to Improve a Multistart Heuristic for Sequencing by Hybridization

Eraldo R. Fernandes[1] and Celso C. Ribeiro[2]

[1] Department of Computer Science, Catholic University of Rio de Janeiro, Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, Brazil
eraldoluis@inf.puc-rio.br

[2] Department of Computer Science, Universidade Federal Fluminense, Rua Passo da Pátria 156, 24210-240 Niterói, Brazil
celso@ic.uff.br

**Abstract.** We describe a multistart heuristic using an adaptive memory strategy for the problem of sequencing by hybridization. The memory-based strategy is able to significantly improve the performance of memoryless construction procedures, in terms of solution quality and processing time. Computational results show that the new heuristic obtains systematically better solutions than more involving and time consuming techniques such as tabu search and genetic algorithms.

## 1 Problem Formulation

A DNA molecule may be viewed as a word in the alphabet {A,C,G,T} of nucleotides. The problem of DNA sequencing consists in determining the sequence of nucleotides that form a DNA molecule. There are currently two techniques for sequencing medium-size molecules: gel electrophoresis and the chemical method. The novel approach of *sequencing by hybridization* offers an interesting alternative to those above [8, 9].

Sequencing by hybridization consists of two phases. The first phase is a biochemical experiment involving a DNA *array* and the molecule to be sequenced, i.e. the *target sequence*. A DNA array is a bidimensional grid in which each cell contains a small sequence of nucleotides which is called a *probe*. The set of all probes in a DNA array is denominated a *library*. Typically, a DNA array represented by $C(\ell)$ contains all possible probes of a fixed size $\ell$. After the array has been generated, it is introduced into an environment with many copies of the target sequence. During the experiment, a copy of the target sequence reacts with a probe if the latter is a subsequence of the former. This reaction is called *hybridization*. At the end of the experiment, it is possible to determine which probes of the array reacted with the target sequence. This set of probes contains all sequences of size $\ell$ that appear in the target sequence and is called the *spectrum*. An illustration of the hybridization experiment involving the tar-

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 4–15, 2005.
© Springer-Verlag Berlin Heidelberg 2005

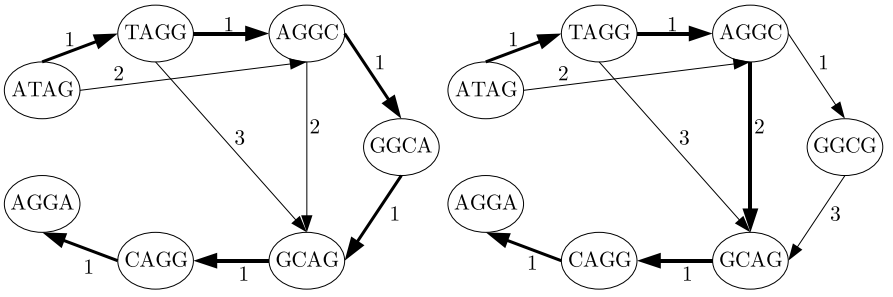
| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AAAA | AAAC | AAAG | AAAT | AACA | AACC | AACG | AACT | AAGA | AAGC | AAGG | AAGT | AATA | AATC | AATG | AATT |
| ACAA | ACAC | ACAG | ACAT | ACCA | ACCC | ACCG | ACCT | ACGA | ACGC | ACGG | ACGT | ACTA | ACTC | ACTG | ACTT |
| AGAA | AGAC | AGAG | AGAT | AGCA | AGCC | AGCG | AGCT | AGGA | AGGC | AGGG | AGGT | AGTA | AGTC | AGTG | AGTT |
| ATAA | ATAC | ATAG | ATAT | ATCA | ATGC | ATCG | ATCT | ATGA | ATGC | ATGG | ATGT | ATTA | ATTC | ATTG | ATTT |
| CAAA | CAAC | CAAG | CAAT | CACA | CACC | CACG | CACT | CAGA | CAGC | CAGG | CAGT | CATA | CATC | CATG | CATT |
| CCAA | CCAC | CCAG | CCAT | CCCA | CCCC | CCCG | CCCT | CCGA | CCGC | CCGG | CCGT | CCTA | CCTC | CCTG | CCTT |
| CGAA | CGAC | CGAG | CGAT | CGCA | CGCC | CGCG | CGCT | CGGA | CGGC | CGGG | CGGT | CGTA | CGTC | CGTG | CGTT |
| CTAA | CTAC | CTAG | CTAT | CTCA | CTCC | CTCG | CTCT | CTGA | CTGC | CTGG | CTGT | CTTA | CTTC | CTTG | CTTT |
| GAAA | GAAC | GAAG | GAAT | GACA | GACC | GACG | GACT | GAGA | GAGC | GAGG | GAGT | GATA | GATC | GATG | GATT |
| GCAA | GCAC | GCAG | GCAT | GCCA | GCCC | GCCG | GCCT | GCGA | GCGC | GCGG | GCGT | GCTA | GCTC | GCTG | GCTT |
| GGAA | GGAC | GGAG | GGAT | GGCA | GGCC | GGCG | GGCT | GGGA | GGGC | GGGG | GGGT | GGTA | GGTC | GGTG | GGTT |
| GTAA | GTAC | GTAG | GTAT | GTCA | GTCC | GTCG | GTCT | GTGA | GTGC | GTGG | GTGT | GTTA | GTTC | GTTG | GTTT |
| TAAA | TAAC | TAAG | TAAT | TACA | TACC | TACG | TACT | TAGA | TAGC | TAGG | TAGT | TATA | TATC | TATG | TATT |

**Fig. 1.** Hybridization experiment involving the target sequence ATAGGCAGGA and all probes of size $\ell = 4$

get sequence ATAGGCAGGA and $C(4)$ is depicted in Figure 1. The highlighted cells are those corresponding to the spectrum.

The second phase of the sequencing by hybridization technique consists in using the spectrum to determine the target sequence. The latter may be viewed as a sequence formed by all $n - \ell + 1$ probes in the spectrum, in which the last $\ell-1$ letters of each probe coincide with the first $\ell-1$ letters of the next. However, two types of errors may be introduced along the hybridization experiment. *False positives* are probes that appear in the spectrum, but not in the target sequence. *False negatives* are probes that should appear in the spectrum, but do not. A particular case of false negatives is due to probes that appear multiple times in the target sequence, since the hybridization experiment is not able to detect the number of repetitions of the same probe. Therefore, a probe appearing $m$ times in the target sequence will generate $m-1$ false negatives. The problem of sequencing by hybridization (SBH) is formulated as follows: given the spectrum $S$, the probe length $\ell$, the size $n$ and the first probe $s_0$ of the target sequence, find a sequence with length smaller than or equal to $n$ containing a maximum number of probes. The maximization of the number of probes of the spectrum corresponds to the minimization of the number of errors in the solution. Errors in the spectrum make the reconstruction problem NP-hard [5].

An instance of SBH may be represented by a directed weighted graph $G(V, E)$, where $V = S$ is the set of nodes and $E = \{(u, v) \mid u, v \in S\}$ is the set of arcs. The weight of the arc $(u, v)$ is given by $w(u, v) = \ell - o(u, v)$, where $o(u, v)$ is the size of the largest sequence that is both a suffix of $u$ and a prefix of $v$. The value $o(u, v)$ is the *superposition* between probes $u$ and $v$. A feasible solution to SBH is an acyclic path in $G$ emanating from node $s_0$ and with total weight smaller than or equal to $n - \ell$. This path may be represented by an ordered node list $a =< a_1, \ldots, a_k >$, with $a_i \in S, i = 1, \ldots, k$. Let $S(a) = \{a_1, \ldots, a_k\}$ be the set of nodes visited by a path $a$ and denote by $|a| = |S(a)|$ the number of nodes in this path. The latter is a feasible solution to SBH if and only if $a_1 = s_0$, $a_i \neq a_j$ for all $a_i, a_j \in S(a)$, and $w(a) \leq n - \ell$, where $w(a) = \sum_{h=1,\ldots,|a|-1} w(a_h, a_{h+1})$ is the sum of the




(a) No errors in the spectrum (b) Errors in the spectrum

**Fig. 2.** Graphs and solutions for the target sequence ATAGGCAGGA with the probe size $\ell = 4$: (a) no errors in the spectrum, (b) one false positive error (GGCG) and one false negative error (GGCA) in the spectrum (not all arcs are represented in the graph)

weights of all arcs in the path. Therefore, SBH consists in finding a maximum cardinality path satisfying the above constraints.

The graph associated with the experiment depicted in Figure 1 is given in Figure 2 (a). The solution is a path visiting all nodes and using only unit weight arcs, since there are no errors in the spectrum. The example in Figure 2 (b) depicts a situation in which probe GGCA was erroneously replaced by probe GGCG, introducing one false positive and one false negative error. The new optimal solution does not visit all nodes (due to the false positive) and uses one arc with weight equal to 2 (due to the false negative).

Heuristics for SBH, handling both false positive and false negative errors, were proposed in [3, 4, 6]. We propose in the next section a new memory-based multistart heuristic for SBH, also handling both false positive and false negative errors. The algorithm is based on an adaptive memory strategy using a set of elite solutions visited along the search. Computational results illustrating the effectiveness of the new memory-based heuristic are reported in Section 3. Concluding remarks are made in the final section.

## 2 Memory-Based Multistart Heuristic

The memory-based multistart heuristic builds multiple solutions using a greedy randomized algorithm. The best solution found is returned by the heuristic. An adaptive memory structure stores the best elite solutions found along the search, which are used within an intensification strategy [7].

The memory is formed by a pool $Q$ that stores $q$ elite solutions found along the search. It is initialized with $q$ null solutions with zero probes each. A new solution $a$ is a candidate to be inserted into the pool if $|a| > \min_{a' \in Q} |a'|$. This solution replaces the worst in the pool if $|a| > \max_{a' \in Q} |a'|$ (i.e., $a$ is better than the best solution currently in the pool) or if $\min_{a' \in Q} dist(a, a') \geq d$, where $d$ is a parameter of the algorithm and $dist(a, a')$ is the number of probes with

different successors in $a$ and $a'$ (i.e., $a$ is better than the worst solution currently in the pool and sufficiently different from every other solution in the pool).

The greedy randomized algorithm iteratively extends a path $a$ initially formed exclusively by probe $s_0$. At each iteration, a new probe is appended at the end of the path $a$. This probe is randomly selected from the restricted candidate list $R = \{v \in S \setminus S(a) \mid o(u,v) \geq (1-\alpha) \cdot \max_{t \in S \setminus S(a)} o(u,t)$ and $w(a) + w(u,v) \leq n - \ell\}$, where $u$ is the last probe in $a$ and $\alpha \in [0,1]$ is a parameter. The list $R$ contains probes with a predefined minimum superposition with the last probe in $a$, restricting the search to more promising regions of the solution space. The construction of a solution stops when $R$ turns up to be empty.

The probability $p(u,v)$ of selecting a probe $v$ from the restricted candidate list $R$ to be inserted after the last probe $u$ in the path $a$ is computed using the superposition between probes $u$ and $v$, and the frequency in which the arc $(u,v)$ appears in the set $Q$ of elite solutions. We define $e(u,v) = \lambda \cdot x(u,v) + y(u,v)$, where $x(u,v) = \min_{t \in S \setminus S(a)}\{w(u,t)/w(u,v)\}$ is higher when the superposition between probes $u$ and $v$ is larger, $y(u,v) = \sum_{a'' \in Q \mid (u,v) \in a''}\{|a''|/\max_{a' \in Q}|a'|\}$ is larger for arcs $(u,v)$ appearing more often in the elite set $Q$, and $\lambda$ is a parameter used to balance the two criteria. Then, the probability of selecting a probe $v$ to be inserted after the last probe $u$ in the path $a$ is given by

$$p(u,v) = \frac{e(u,v)}{\sum_{t \in R} e(u,t)}.$$

The value of $\lambda$ should be high in the beginning of the algorithm, when the information in the memory is still weak. The value of $\alpha$ should be small in

```
procedure MultistartHeuristic(S, s0, ℓ, n)
1.   Initialize o, w, α, q, d, Q;
2.   a* ← null;
3.   for i = 1 to N do
4.        Set a ← (s0);
5.        Build the restricted candidate list R;
6.        while R ≠ ∅ do
7.             Compute the selection probability for each probe v ∈ R;
8.             Randomly select a probe v ∈ R;
9.             Extend the current solution a by appending v to its end;
10.            Update the restricted candidate list R;
11.       end;
12.       Use a to update the pool of elite solutions Q;
13.       if |a| > |a*| then set a* ← a;
14.  end;
15.  return a*;
end;
```

**Fig. 3.** Memory-based multistart heuristic

the beginning, to allow for the construction of good solutions by the greedy randomized heuristic and so as to quickly enrich the memory. The value of $\alpha$ is progressively increased along the algorithm when the weight $\lambda$ given to the superposition information decreases, to increase the diversity of the solutions in the list $R$.

We sketch in Figure 3 the pseudo-code with the main steps of the memory-based multistart heuristic, in which $N$ iterations are performed.

## 3 Numerical Results

The memory-based multistart heuristic was implemented in C++, using version 3.3.2 of the GNU compiler. The `rand` function was used for the generation of pseudo-random numbers. The computational experiments were performed on a 2.4 GHz Pentium IV machine with 512 MB of RAM.

Two sets of test instances have been generated from human and random DNA sequences. Instances in group A were built from 40 human DNA sequences obtained from GenBank [2], as described in [4]. Prefixes of size 109, 209, 309, 409, and 509 were extracted from these sequences. For each prefix, a hybridization experiment with the array $C(10)$ was simulated, producing spectra with 100, 200, 300, 400, and 500 probes. Next, false negatives were simulated by randomly removing 20% of the probes in each spectrum. False positives were simulated by inserting 20% of new probes in each spectrum. Overall, we have generated 200 instances in this group, 40 of each size. Instances in group R were generated from 100 random DNA sequences with prefixes of size 100, 200, ..., and 1000. Once again, 20% false negatives and 20% false positives have been generated. There are 100 instances of each size in this group, in a total of 1000 instances.

Preliminary computational experiments have been performed to tune the main parameters of the algorithm. The following settings were selected: $N = 10n$ (number of iterations performed by the multistart heuristic), $q = n/80$ (size of the pool of elite solutions), and $d = 2$ (minimum difference for a solution to be accepted in the pool). Parameters $\alpha$ and $\lambda$ used by the greedy randomized construction heuristic are self-tuned. Iterations of this heuristic are grouped in 20 blocks. Each block performs $n/2$ iterations. In the first block, $\lambda = 100q$. In the second block, $\lambda = 10q$. The value of $\lambda$ is reduced by $q$ at each new block, until it is made equal to zero. The value of $\alpha$ is initialized according to Tables 1 and 2, and increased by 0.1 after every five blocks of $n/2$ iterations, until it is made equal to one.

Two versions of the `MultistartHeuristic` algorithm described in Figure 3 were implemented: MS is a purely multistart procedure that does not make use of memory, while MS+Mem fully exploits the adaptive memory strategy described

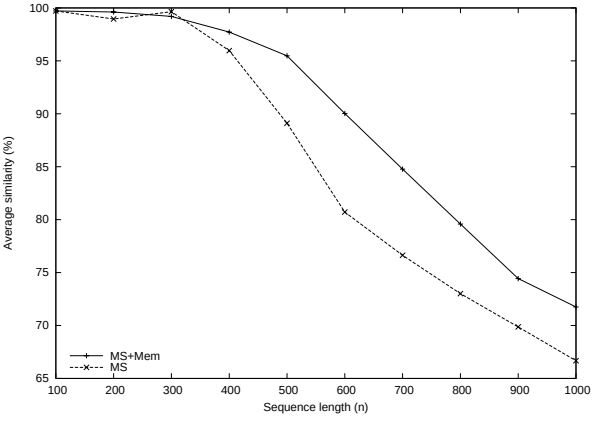
**Table 1.** Initial values of $\alpha$ for the instances in group R

| $n$ | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\alpha$ | 0.5 | 0.3 | 0.2 | 0.1 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

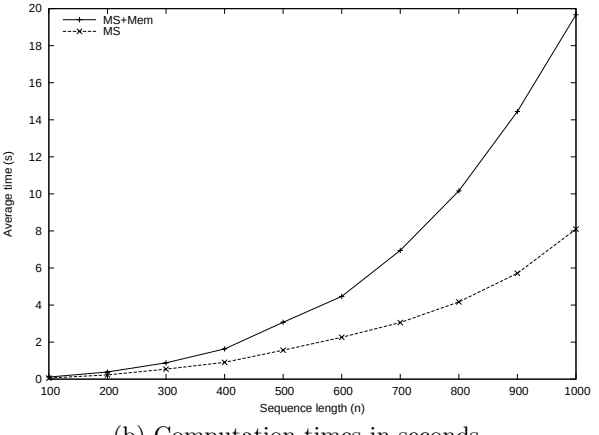
**Table 2.** Initial values of $\alpha$ for the instances in group A

| $n$ | 109 | 209 | 309 | 409 | 509 |
|---|---|---|---|---|---|
| $\alpha$ | 0.5 | 0.3 | 0.2 | 0.1 | 0.1 |

in the previous section. To evaluate the quality of the solutions produced by the heuristics, we performed the alignment of their solutions with the corresponding target sequences, as in [4]. The *similarity* between two sequences is defined as the fraction (in percent) of symbols that coincide in their alignment. A similarity of 100% means that the two sequences are identical. Average similarities and average computation times in seconds over all test instances in group R for both heuristics are displayed in Figure 4. These results clearly illustrate the

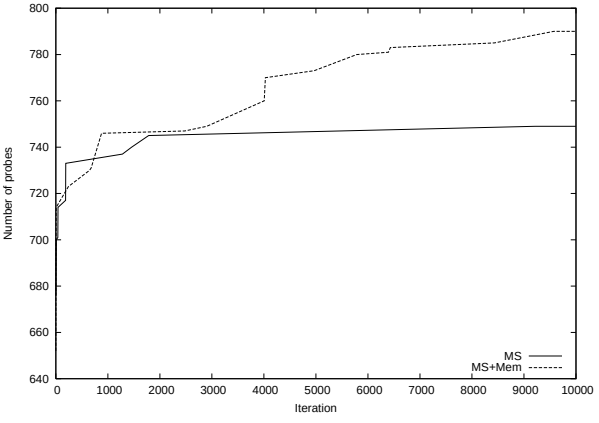



(a) Similarities

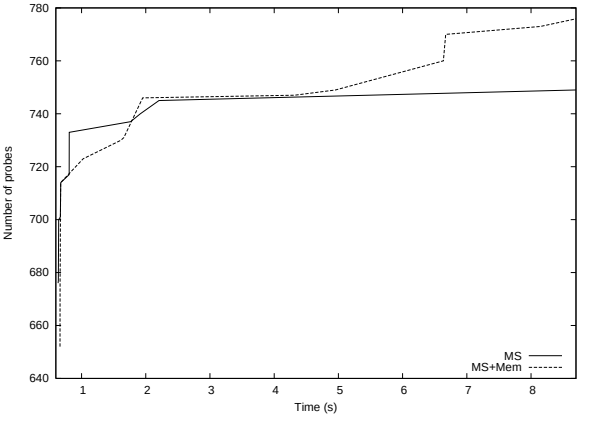



(b) Computation times in seconds

**Fig. 4.** Computational results obtained by heuristics MS+Mem and MS for the instances in group R




(a) Best solutions along 10000 iterations




(b) Best solutions along 8.7 seconds of processing time

**Fig. 5.** Probes in the best solutions found by heuristics MS and MS+Mem for an instance with $n = 1000$ from group R

contribution of the adaptive memory strategy to improve the performance of the purely multistart heuristic.

We have performed another experiment to further evaluate the influence of the adaptive memory strategy on the multistart heuristic. We illustrate our findings for one specific instance with size $n = 1000$ from group R. Figure 5 (a) displays the number of probes in the best solution obtained by each heuristic along 10000 iterations. We notice that the best solution already produced by MS+Mem until a given iteration is consistently better than that obtained by MS, in particular after a large number of iterations have been performed. Figure 5 (b) depicts the same results along 8.7 seconds of processing time. The purely multistart heuristic seems to freeze and prematurely converge to a local minimum very quickly. The use of the adaptive memory strategy leads

the heuristic to explore other regions of the solution space and to find better solutions.

To give further evidence concerning the performance of the two heuristics, we used the methodology proposed by Aiex et al. [1] to assess experimentally the behavior of randomized algorithms. This approach is based on plots showing empirical distributions of the random variable *time to target solution value*. To plot the empirical distribution, we select a test instance, fix a target solution value, and run algorithms MS and MS+Mem 100 times each, recording the running time when a solution with cost at least as good as the target value is found. For each algorithm, we associate with the $i$-th sorted running time $t_i$ a probability $p_i = (i - \frac{1}{2})/100$ and plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, 100$.

Since the relative performance of the two heuristics is quite similar over all test instances, we selected one particular instance of size $n = 500$ from group R and used its optimal value as the target. The computational results are displayed in Figure 6. This figure shows that the heuristic MS+Mem using the adaptive memory strategy is capable of finding target solution values with higher probability or in smaller computation times than the pure multistart heuristic MS, illustrating once again the contribution of the adaptive memory strategy. These results also show that the heuristic MS+Mem is more robust.

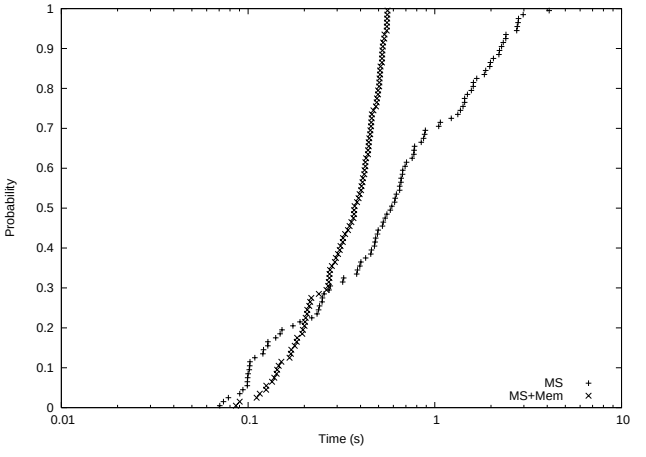
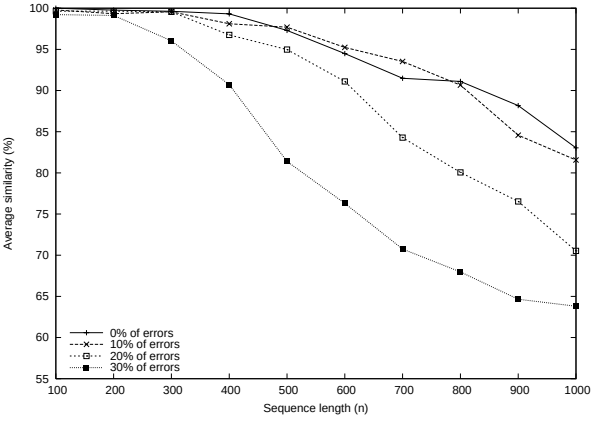
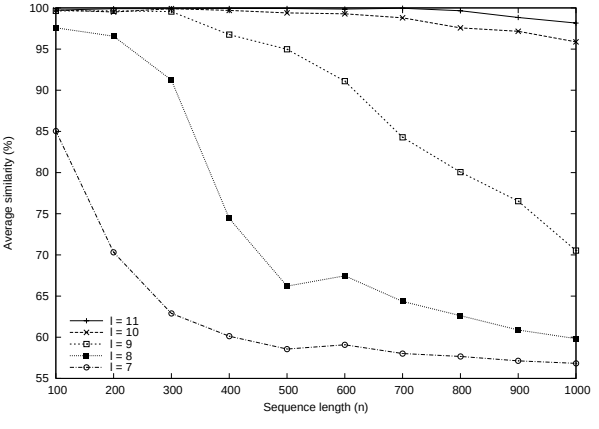



**Fig. 6.** Empirical probability distributions of time to target solution value for heuristics MS+Mem and MS for an instance of size $n = 500$ from group R

We have also considered the behavior of the heuristic MS+Mem when the number of errors and the size of the probes vary. The algorithm was run on randomly generated instances as those in group R, for different rates of false negative and false positive errors: 0%, 10%, 20%, and 30%. Similarly, the




(a) Rates of errors: 0%, 10%, 20%, and 30%




(b) Probe sizes: $\ell = 7, 8, 9, 10, 11$

**Fig. 7.** Results obtained by the heuristic MS+Mem for instances with different rates of errors (a) and probe sizes (b)

**Table 3.** Average similarities for the instances in group A

| | $n$ | | | | |
|---|---|---|---|---|---|
| Algorithm | 109 | 209 | 309 | 409 | 509 |
| TS | 98.6 | 94.1 | 89.6 | 88.5 | 80.7 |
| OW | 99.4 | 95.2 | 95.7 | 92.1 | 90.1 |
| GA | 98.3 | 97.9 | 99.1 | 98.1 | 93.5 |
| MS+Mem | 100.0 | 100.0 | 99.2 | 99.4 | 99.5 |

algorithm was also run on randomly generated instances as those in group R with different probe sizes $\ell = 7, 8, 9, 10, 11$. Numerical results are displayed in Figure 7.

**Table 4.** Average computation times in seconds for the instances in group A

| | $n$ | | | | |
|---|---|---|---|---|---|
| Algorithm | 109 | 209 | 309 | 409 | 509 |
| TS | <1.0 | 5.0 | 14.0 | 28.0 | 51.0 |
| OW | <1.0 | <1.0 | <1.0 | <1.0 | <1.0 |
| GA | 0.1 | 0.3 | 0.9 | 1.5 | 2.1 |
| MS+Mem | 0.1 | 0.4 | 0.8 | 1.6 | 3.0 |

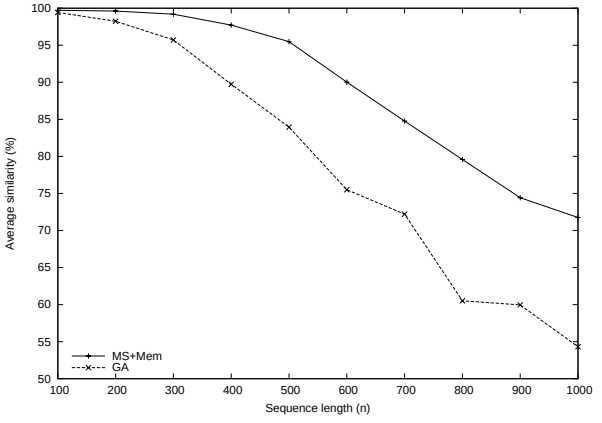
**Table 5.** Target sequences exactly reconstructed for the instances in group A

| | $n$ | | | | |
|---|---|---|---|---|---|
| Algorithm | 109 | 209 | 309 | 409 | 509 |
| TS | 28 | 23 | 17 | 10 | 10 |
| OW | 28 | 20 | 21 | 13 | 14 |
| GA | 37 | 30 | 37 | 30 | 28 |
| MS+Mem | 40 | 40 | 39 | 39 | 39 |

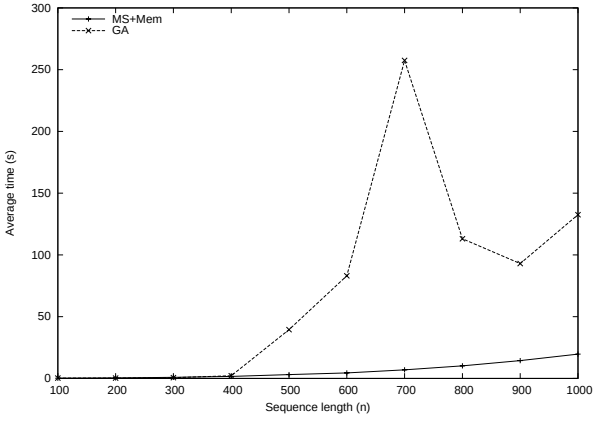
The memory-based multistart heuristic MS+Mem was compared with the tabu search algorithm (TS) in [4], the overlapping windows heuristic (OW) in [3], and the genetic algorithm (GA) in [6]. The numerical results are summarized in Tables 3 and 4, which depict the average similarities and the average computation times in seconds observed for each algorithm over the 40 instances with the same size in group A. The heuristic MS+Mem found much better solutions than the others. The alignments observed for the solutions produced by MS+Mem are systematically higher. The new heuristic MS+Mem is faster than TS and competitive with GA (the results displayed for the overlapping windows heuristic were obtained on a CRAY T3E-900 supercomputer).

Further comparative results for the four algorithms are given in Table 5, in which we give the number of target sequences exactly reconstructed for each algorithm over the 40 instances with the same size in group A. The heuristic MS+Mem was able to reconstruct the 40 original sequences of size 109 and 209, and 39 out of the 40 instances of sizes 309, 409, and 509, corresponding to a total of 197 out of the 200 test instances in group A. The overlapping windows and the tabu search heuristics found, respectively, only 96 and 88 out of the 200 original sequences.

We also compared the new heuristic MS+Mem with the genetic algorithm for the instances in group R. Average similarities and average computation times in seconds are shown in Figure 8. Table 6 depicts the number of target sequences exactly reconstructed by MS+Mem and the genetic algorithm over the 100 instances of each size in group R. Also for the instances in this group, the new heuristic outperformed the genetic algorithm both in terms of solution quality and computation times.




(a) Similarities




(b) Computation times in seconds

**Fig. 8.** Computational results obtained by the heuristic MS+Mem and the genetic algorithm (GA) for the instances in group R

**Table 6.** Target sequences exactly reconstructed for the instances in group R

| | $n$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Algorithm | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
| GA | 70 | 61 | 55 | 37 | 23 | 11 | 9 | 3 | 1 | 2 |
| MS+Mem | 79 | 74 | 83 | 72 | 58 | 52 | 24 | 14 | 11 | 3 |

## 4 Concluding Remarks

We proposed a multistart heuristic for the problem of sequencing by hybridization, based on an intensification strategy that makes use of an adaptive memory. The adaptive memory strategy makes use of a set of elite solutions found along

the search. The choice of the new element to be inserted into the partial solution at each iteration of a greedy randomized construction procedure is based not only on greedy information, but also on frequency information extracted from the memory.

Computational results on test instances generated from human and random DNA sequences have shown that the memory-based strategy is able to significantly improve the performance of a memoryless construction procedure purely based on greedy choices. The memory-based multistart heuristic obtained better results than more involving and time consuming techniques such as tabu search and genetic algorithms, both in terms of solution quality and computation times.

The use of adaptive memory structures that are able to store information about the relative positions of the tasks in elite solutions seems to be particularly suited to scheduling problems in which blocks formed by the same tasks in the same order often appear in the best solutions.

## References

1. R.M. Aiex, M.G.C. Resende, and C.C. Ribeiro. Probability distribution of solution time in GRASP: An experimental investigation. *Journal of Heuristics*, 8:343–373, 2002.
2. D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, and D.L. Wheeler. Genbank: Update. *Nucleic Acids Research*, 32:D23–D26, 2004.
3. J. Blazewicz, P. Formanowicz, F. Guinand, and M. Kasprzak. A heuristic managing errors for DNA sequencing. *Bioinformatics*, 18:652–660, 2002.
4. J. Blazewicz, P. Formanowicz, M. Kasprzak, W. T. Markiewicz, and T. Weglarz. Tabu search for DNA sequencing with false negatives and false positives. *European Journal of Operational Research*, 125:257–265, 2000.
5. J. Blazewicz and M. Kasprzak. Complexity of DNA sequencing by hybridization. *Theoretical Computer Science*, 290:1459–1473, 2003.
6. T.A. Endo. Probabilistic nucleotide assembling method for sequencing by hybridization. *Bioinformatics*, 20:2181–2188, 2004.
7. C. Fleurent and F. Glover. Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. *INFORMS Journal on Computing*, 11:198–204, 1999.
8. P.A. Pevzner. *Computational molecular biology: An algorithmic approach*. MIT Press, 2000.
9. M.S. Waterman. *Introduction to computational biology: Maps, sequences and genomes*. Chapman & Hall, 1995.

# High-Performance Algorithm Engineering for Large-Scale Graph Problems and Computational Biology

David A. Bader*

Electrical and Computer Engineering Department,
University of New Mexico, Albuquerque, NM 87131
dbader@ece.unm.edu

**Abstract.** Many large-scale optimization problems rely on graph theoretic solutions; yet high-performance computing has traditionally focused on regular applications with high degrees of locality. We describe our novel methodology for designing and implementing irregular parallel algorithms that attain significant performance on high-end computer systems. Our results for several fundamental graph theory problems are the first ever to achieve parallel speedups. Specifically, we have demonstrated for the first time that significant parallel speedups are attainable for arbitrary instances of a variety of graph problems and are developing a library of fundamental routines for discrete optimization (especially in computational biology) on shared-memory systems.

Phylogenies derived from gene order data may prove crucial in answering some fundamental questions in biomolecular evolution. High-performance algorithm engineering offers a battery of tools that can reduce, sometimes spectacularly, the running time of existing approaches. We discuss one such such application, GRAPPA, that demonstrated over a billion-fold speedup in running time (on a variety of real and simulated datasets), by combining low-level algorithmic improvements, cache-aware programming, careful performance tuning, and massive parallelism. We show how these techniques are directly applicable to a large variety of problems in computational biology.

## 1 Experimental Parallel Algorithms

We discuss our design and implementation of theoretically-efficient parallel algorithms for combinatorial (irregular) problems that deliver significant speedups on typical configurations of SMPs and SMP clusters and scale gracefully with the number of processors. Problems in genomics, bioinformatics, and computational ecology provide the focus for this research. Our source code is freely-available under the GNU General Public License (GPL) from our web site.

* This work was supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, ITR EIA-01-21377,Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA contract NBCH30390004.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 16–21, 2005.
© Springer-Verlag Berlin Heidelberg 2005

### 1.1 Theoretically- and Practically-Efficient Portable Parallel Algorithms for Irregular Problems

Our research has designed parallel algorithms and produced implementations for primitives and kernels for important operations such as prefix-sum, pointer-jumping, symmetry breaking, and list ranking; for combinatorial problems such as sorting and selection; for parallel graph theoretic algorithms such as spanning tree, minimum spanning tree, graph decomposition, and tree contraction; and for computational genomics such as maximum parsimony (see [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]). Several of these classic graph theoretic problems are notoriously challenging to solve in parallel due to the fine-grained global accesses needed for the sparse and irregular data structures. We have demonstrated theoretically and practically fast implementations that achieve parallel speedup for the first time when compared with the best sequential implementation on commercially available platforms.

## 2 Combinatorial Algorithms for Computational Biology

In the 50 years since the discovery of the structure of DNA, and with new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Many of the newly faced challenges require high-performance computing, either due to the massive-parallelism required by the problem, or the difficult optimization problems that are often combinatoric and NP-hard. Unlike the traditional uses of supercomputers for regular, numerical computing, many problems in biology are irregular in structure, significantly more challenging to parallelize, and integer-based using abstract data structures.

Biologists are in search of biomolecular sequence data, for its comparison with other genomes, and because its structure determines function and leads to the understanding of biochemical pathways, disease prevention and cure, and the mechanisms of life itself. Computational biology has been aided by recent advances in both technology and algorithms; for instance, the ability to sequence short contiguous strings of DNA and from these reconstruct the whole genome and the proliferation of high-speed microarray, gene, and protein chips for the study of gene expression and function determination. These high-throughput techniques have led to an exponential growth of available genomic data.

Algorithms for solving problems from computational biology often require parallel processing techniques due to the data- and compute-intensive nature of the computations. Many problems use polynomial time algorithms (e.g., all-to-all comparisons) but have long running times due to the large number of items in the input; for example, the assembly of an entire genome or the all-to-all comparison of gene sequence data. Other problems are compute-intensive due to their inherent algorithmic complexity, such as protein folding and reconstructing evolutionary histories from molecular data, that are known to be NP-hard (or harder) and often require approximations that are also complex.

## 3 Phylogeny Reconstruction

A phylogeny is a representation of the evolutionary history of a collection of organisms or genes (known as taxa). The basic assumption of process necessary to phylogenetic reconstruction is repeated divergence within species or genes. A phylogenetic reconstruction is usually depicted as a tree, in which modern taxa are depicted at the leaves and ancestral taxa occupy internal nodes, with the edges of the tree denoting evolutionary relationships among the taxa. Reconstructing phylogenies is a major component of modern research programs in biology and medicine (as well as linguistics). Naturally, scientists are interested in phylogenies for the sake of knowledge, but such analyses also have many uses in applied research and in the commercial arena.

Existing phylogenetic reconstruction techniques suffer from serious problems of running time (or, when fast, of accuracy). The problem is particularly serious for large data sets: even though data sets comprised of sequence from a single gene continue to pose challenges (e.g., some analyses are still running after two years of computation on medium-sized clusters), using whole-genome data (such as gene content and gene order) gives rise to even more formidable computational problems, particularly in data sets with large numbers of genes and highly-rearranged genomes.

To date, almost every model of speciation and genomic evolution used in phylogenetic reconstruction has given rise to NP-hard optimization problems. Three major classes of methods are in common use. Heuristics (a natural consequence of the NP-hardness of the problems) run quickly, but may offer no quality guarantees and may not even have a well-defined optimization criterion, such as the popular *neighbor-joining* heuristic [13]. Optimization based on the criterion of *maximum parsimony* (MP) [14] seeks the phylogeny with the least total amount of change needed to explain modern data. Finally, optimization based on the criterion of *maximum likelihood* (ML) [15] seeks the phylogeny that is the most likely to have given rise to the modern data.

Heuristics are fast and often rival the optimization methods in terms of accuracy, at least on datasets of moderate size. Parsimony-based methods may take exponential time, but, at least for DNA and amino acid data, can often be run to completion on datasets of moderate size. Methods based on maximum likelihood are very slow (the point estimation problem alone appears intractable) and thus restricted to very small instances, and also require many more assumptions than parsimony-based methods, but appear capable of outperforming the others in terms of the quality of solutions when these assumptions are met. Both MP- and ML-based analyses are often run with various heuristics to ensure timely termination of the computation, with mostly unquantified effects on the quality of the answers returned.

Thus there is ample scope for the application of high-performance algorithm engineering in the area. As in all scientific computing areas, biologists want to study a particular dataset and are willing to spend months and even years in the process: accurate branch prediction is the main goal. However, since all exact algorithms scale exponentially (or worse, in the case of ML approaches) with the

number of taxa, speed remains a crucial parameter—otherwise few datasets of more than a few dozen taxa could ever be analyzed.

As an illustration, we briefly discuss our experience with a high-performance software suite, GRAPPA (Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms) that we developed, *GRAPPA* extends Sankoff and Blanchette's breakpoint phylogeny algorithm [16] into the more biologically-meaningful inversion phylogeny and provides a highly-optimized code that can make use of distributed- and shared-memory parallel systems (see [17, 18, 19, 20, 21, 22] for details). In [23] we give the first linear-time algorithm and fast implementation for computing inversion distance between two signed permutations. We ran *GRAPPA* on a 512-processor IBM Linux cluster with Myrinet and obtained a 512-fold speed-up (linear speedup with respect to the number of processors): a complete breakpoint analysis (with the more demanding inversion distance used in lieu of breakpoint distance) for the 13 genomes in the Campanulaceae data set ran in less than 1.5 hours in an October 2000 run, for a *million-fold* speedup over the original implementation. Our latest version features significantly improved bounds and new distance correction methods and, on the same dataset, exhibits a speedup factor of *over one billion*. We achieved this speedup through a combination of parallelism and high-performance algorithm engineering. Although such spectacular speedups will not always be realized, we suggest that many algorithmic approaches now in use in the biological, pharmaceutical, and medical communities can benefit tremendously from such an application of high-performance techniques and platforms.

This example indicates the potential of applying high-performance algorithm engineering techniques to applications in computational biology, especially in areas that involve complex optimizations: our reimplementation did not require new algorithms or entirely new techniques, yet achieved gains that turned an impractical approach into a usable one.

## References

1. Bader, D., Illendula, A., Moret, B.M., Weisse-Bernstein, N.: Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In Brodal, G., Frigioni, D., Marchetti-Spaccamela, A., eds.: Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001). Volume 2141 of Lecture Notes in Computer Science., Århus, Denmark, Springer-Verlag (2001) 129–144
2. Bader, D., Moret, B., Sanders, P.: Algorithm engineering for parallel computation. In Fleischer, R., Meineche-Schmidt, E., Moret, B., eds.: Experimental Algorithmics. Volume 2547 of Lecture Notes in Computer Science. Springer-Verlag (2002) 1–23
3. Bader, D., Sreshta, S., Weisse-Bernstein, N.: Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In Sahni, S., Prasanna, V., Shukla, U., eds.: Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002). Volume 2552 of Lecture Notes in Computer Science., Bangalore, India, Springer-Verlag (2002) 63–75

4. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004), Santa Fe, NM (2004)
5. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). Journal of Parallel and Distributed Computing (2004) to appear.
6. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004), Santa Fe, NM (2004)
7. Cong, G., Bader, D.A.: The Euler tour technique and parallel rooted spanning tree. In: Proc. Int'l Conf. on Parallel Processing (ICPP), Montreal, Canada (2004) 448–457
8. Su, M.F., El-Kady, I., Bader, D.A., Lin, S.Y.: A novel FDTD application featuring OpenMP-MPI hybrid parallelization. In: Proc. Int'l Conf. on Parallel Processing (ICPP), Montreal, Canada (2004) 373–379
9. Bader, D., Madduri, K.: A parallel state assignment algorithm for finite state machines. In: Proc. 11th Int'l Conf. on High Performance Computing (HiPC 2004), Bangalore, India, Springer-Verlag (2004)
10. Cong, G., Bader, D.: Lock-free parallel algorithms: An experimental study. In: Proc. 11th Int'l Conf. on High Performance Computing (HiPC 2004), Bangalore, India, Springer-Verlag (2004)
11. Cong, G., Bader, D.: An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM (2004) Submitted for publication.
12. Bader, D., Cong, G., Feo, J.: A comparison of the performance of list ranking and connected components algorithms on SMP and MTA shared-memory systems. Technical report, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM (2004) Submitted for publication.
13. Saitou, N., Nei, M.: The neighbor-joining method: A new method for reconstruction of phylogenetic trees. Molecular Biological and Evolution **4** (1987) 406–425
14. Farris, J.: The logical basis of phylogenetic analysis. In Platnick, N., Funk, V., eds.: Advances in Cladistics. Columbia Univ. Press, New York (1983) 1–36
15. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17** (1981) 368–376
16. Sankoff, D., Blanchette, M.: Multiple genome rearrangement and breakpoint phylogeny. Journal of Computational Biology **5** (1998) 555–570
17. Bader, D., Moret, B., Vawter, L.: Industrial applications of high-performance computing for phylogeny reconstruction. In Siegel, H., ed.: Proc. SPIE Commercial Applications for High-Performance Computing. Volume 4528., Denver, CO, SPIE (2001) 159–168
18. Bader, D., Moret, B.M., Warnow, T., Wyman, S., Yan, M.: High-performance algorithm engineering for gene-order phylogenies. In: DIMACS Workshop on Whole Genome Comparison, Piscataway, NJ, Rutgers University (2001)
19. Moret, B., Bader, D., Warnow, T.: High-performance algorithm engineering for computational phylogenetics. J. Supercomputing **22** (2002) 99–111 Special issue on the best papers from ICCS'01.
20. Moret, B., Wyman, S., Bader, D., Warnow, T., Yan, M.: A new implementation and detailed study of breakpoint analysis. In: Proc. 6th Pacific Symp. Biocomputing (PSB 2001), Hawaii (2001) 583–594

21. Moret, B.M., Bader, D., Warnow, T., Wyman, S., Yan, M.: GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. In: Proc. Botany, Albuquerque, NM (2001)
22. Yan, M.: High Performance Algorithms for Phylogeny Reconstruction with Maximum Parsimony. PhD thesis, Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM (2004)
23. Bader, D., Moret, B., Yan, M.: A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. Journal of Computational Biology **8** (2001) 483–491

# The "Real" Approximation Factor of the MST Heuristic for the Minimum Energy Broadcasting

Michele Flammini[1], Alfredo Navarra[1], and Stephane Perennes[2]

[1] Computer Science Department, University of L'Aquila,
Via Vetoio, loc. Coppito I-67100 L'Aquila, Italy
{flammini, navarra}@di.univaq.it
[2] MASCOTTE project, I3S-CNRS/INRIA/University of Nice,
Route des Lucioles BP 93 F-06902 Sophia Antipolis, France
Stephane.Perennes@sophia.inria.fr

**Abstract.** The paper deals with one of the most studied problems during the last years in the field of wireless communications in Ad-Hoc networks. The problem consists in reducing the total energy consumption of wireless radio stations randomly spread on a given area of interest to perform the basic pattern of communication given by the Broadcast. Recently an almost tight 6.33-approximation of the Minimum Spanning Tree heuristic has been proved [8]. While such a bound is theoretically close to optimum compared to the known lower bound of 6 [10], there is an evident gap with practical experimental results. By extensive experiments, proposing a new technique to generate input instances and supported by theoretical results, we show how the approximation ratio can be actually considered close to 4 for a "real world" set of instances, that is, instances with a number of nodes more representative of practical purposes.

## 1 Introduction

In the context of Ad-Hoc networking, one of the most popular studied problems is the so called *Minimum Energy Broadcast Routing* (MEBR). The problem arises from the requirement of a basic pattern of communication such as the Broadcast. Given a set of radio stations (or nodes) randomly (or suitably) spread on a given area of interest, and specified one of those stations as the source, the problem is to assign the transmission range of each station so as to induce a broadcast communication from the source with a minimum overall power consumption. A communication session can be established through a series of wireless links involving any of the network nodes and therefore Ad-Hoc networks are *multi-hop* networks. To this aim, the nodes have the ability to adjust their transmission power as needed. Thus every node is assigned a transmission range and every node inside this range receives its message. Considering the fact that the nodes operate with a limited supply of energy and given the nature of the operations for which this kind of networks are used, such as military operations or emergency

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 22–31, 2005.
© Springer-Verlag Berlin Heidelberg 2005

disaster relief, a fundamental problem is of assigning transmission ranges in such a way that the total consumed energy is minimum.

According to the mostly used power attenuation model [11, 4], when a node $s$ transmits with power $P_s$, a node $r$ can receive its message if and only if $\frac{P_s}{\|s,r\|^2} > 1$, where $\|s, r\|$ is the Euclidean distance between $s$ and $r$.

Since the MEBR problem is $NP$-hard [3], a lot of effort was devoted to device good approximation algorithms. Several papers progressively reduced the estimate of the approximation ratio of the fundamental Minimum Spanning Tree (MST) heuristic from 40 to 6.33 [3, 6, 10, 4, 8]. Roughly speaking the heuristic computes the directed minimum spanning tree from the given source to the leaves starting from the complete weighted graph obtained from the set of nodes in which weights are the square distances of the endpoints of the edges. For each node, then, the heuristic assigns a power of transmission equal to the weight of the longest outgoing edge.

Even if the 6.33-approximation ratio is almost tight according to the lower bound of 6 [10], there is an evident gap between such a ratio and the experimental results obtained in several papers (see for instance [11, 2, 6, 7, 1, 9]). This suggests to investigate more carefully the possible input instances in order to better understand this phenomenon. The goal is to classify some specific family of instances according to the output of the MST heuristic. The most common method used to randomly generate the input instances has been that of uniformly spreading the nodes inside a given area. In this paper we propose a new method to produce instances in order to maximize the final cost of the MST heuristic. In this way we better catch the intrinsic properties of the problem. Motivated by the obtained experimental studies, we also provide theoretical results that lead to an almost tight 4-approximation ratio for high-density instances of the MEBR problem. The tightness of such ratio is of its own interest since the common intuition was of a much better performance of the MST heuristic on high-density instances. Moreover, such instances are more representative of practical environments since for a small number of nodes exhaustive algorithms can be applied (see for instance the integer linear programming formulation proposed in [6]).

The paper is organized as follows. In the next section we briefly provide some basic definitions and summarize the estimation method proposed in [4] by which an 8-approximation for the MST heuristic arises. That will be useful for the rest of the paper. In Section 3 we formally describe the algorithm to generate suitable instances that maximize the cost of the MST heuristic. In Section 4 we present the obtained experimental results and in Section 5 we present theoretical results that strengthen the experimental ones. Finally, in Section 6, we discuss some conclusive remarks.

## 2 Definitions and Notation

Let us first provide a formal definition of the Minimum Energy Broadcast Routing (MEBR) problem in the 2-dimensional space (see [3, 10, 2] for a more detailed discussion). Given a set of points $S$ in a 2-dimensional Euclidean space that

represents the set of radio stations, let $G_2(S)$ be the complete weighted graph whose nodes are the points of $S$ and in which the weight of each edge $\{x, y\}$ is the power consumption needed for a correct communication between $x$ and $y$, that is $\|x, y\|^2$.

A range assignment for $S$ is a function $r : S \rightarrow \mathbb{R}^+$ such that the range $r(x)$ of a station $x$ denotes the maximal distance from $x$ at which signals can be correctly received. The total cost of a range assignment is then $cost(r) = \sum_{x \in S} r(x)^2$.

A range assignment $r$ for $S$ yields a directed communication graph $G^r = (S, A)$ such that, for each $(x, y) \in S^2$, the directed edge $(x, y)$ belongs to $A$ if and only if $y$ is at distance at most $r(x)$ from $x$. In other words, $(x, y)$ belongs to $A$ if and only if the power emission of $x$ is at least equal to the weight of $\{x, y\}$ in $G^2(S)$. In order to perform the required *minimum energy broadcast* from a given source $s \in S$, $G^r$ must contain a directed spanning tree rooted at $s$ and must have the minimum cost.

One fundamental algorithm, called the MST heuristic [11], is based on the idea of tuning ranges so as to include a spanning tree of minimum cost. More precisely, denoted as $T_2(S)$ a minimum spanning tree of $G_2(S)$ and as $MST(G_2(S))$ its cost, considering $T_2(S)$ rooted at the source station $s$, the heuristic directs the edges of $T_2(S)$ toward the leaves and sets the range $r(x)$ of every internal station $x$ of $T_2(S)$ with $k$ children $x_1, \ldots, x_k$ in such a way that $r(x) = max_{i=1,\ldots,k} \|x, x_i\|^2$. In other words, $r$ is the range assignment of minimum cost inducing the directed tree derived from $T_2(S)$ and it is such that $cost(r) \leq MST(G_2(S))$.

Let us denote by $C_r$ a circle of radius $r$. From [3, 10, 4] it is possible to restrict the study of the performance of the MST heuristic just considering $C_1$ centered at the source as area of interest to locate the radio stations. An 8-approximation is then proved in [4] by assigning a growing circle to each node till all the circles form a unique connected area component. Such an area, denoted by $a(S, \frac{r_{max}}{2})$, is related to the $MST$ cost according to the following equation (see [5, 4]),

$$MST(G_2(S)) = 2 \int_0^{r_{max}} (n(S, r) - 1) r \; dr,$$

where $r_{max}$ is the size of the longest edge contained in $MST(S)$ and $n(S, r)$ is the number of connected components obtained from $S$ associating a circle of radius $r$ to each node[1]. The following bounds are then derived

$$\frac{\pi}{4} MST(S) + \frac{\pi}{4} r_{max}^2 \leq a(S, \frac{r_{max}}{2}) \leq \pi (1 + \frac{r_{max}}{2})^2,$$

hence obtaining

$$MST(S) \leq 4(1 + r_{max}).$$

The 8-approximation then holds by observing that $r_{max} \leq 1$. For $r_{max}$ tending to 0, the approximation ratio of the MST heuristic tends to 4. Studying the

[1] Two nodes belong to the same connected component if and only if the two associated circles are overlapping in at least one point.

results obtained by extensive experiments we are going to show that, in practice, that is, for a considerable number of nodes, such a bound of 4 is almost tight.

## 3 Augmenting Algorithm

It is well-known that the lower bound for the MST heuristic is given by the hexagonal shape presented in [10] where the instance is given by seven nodes that are the center and the vertices of a regular hexagon inscribed in $C_1$ (see Figure 1). On such an instance the MST heuristic cost can be equal to 6 while the optimal solution costs just 1. It is evident that 6 is the maximum cost for instances inside a $C_1$ in which the source is its center and the number of nodes is at most 7. Performing experiments as described in [11, 2, 6, 7, 1, 9], even just throwing seven nodes, in which one of them is fixed to be the center of $C_1$ and the other ones are randomly at uniform distributed inside such a circle, it is really "lucky" to happen that a similar high cost instance appears. Moreover increasing the number of nodes involved in the experiments, on average, the cost of the performed MST decreases.

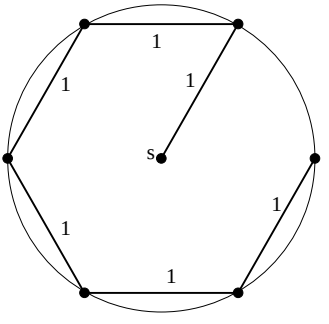



**Fig. 1.** The 6 lower bound for the MST heuristic provided in [10]

In this paper we are interested in maximizing the cost of a possible MST inside $C_1$ considering its center $s$ as the source in order to better understand the actual quality of the performance of the MST heuristic over interesting instances more representative of the real world applications. Roughly speaking, starting from random instances, the maximization is due to slight movements of the nodes according to some useful properties of the MST construction. For instance if we want to increase the cost of an edge of the MST, the easiest idea is to increase the distance of its endpoints. Let us now consider a node $v \neq s$ of a generic instance given in input. We consider the degree of such a node in the undirected tree obtained from the MST heuristic before assigning the directions. Let $N_v = \{v_1, v_2, \ldots, v_k\}$ be the set of the neighbors of $v$ in such a tree. We evaluate the median point $p = (x, y)$ whose coordinates

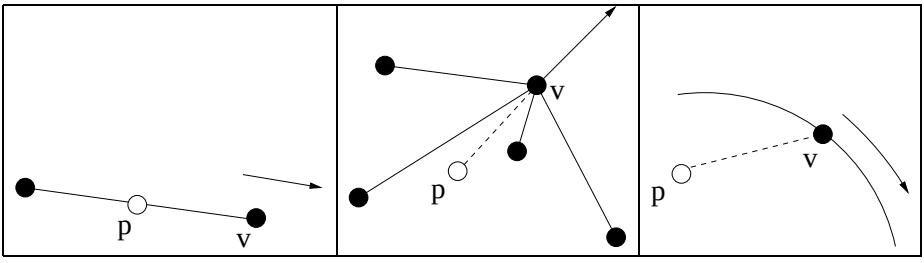



**Fig. 2.** Augmenting the edge costs when a node has one or more neighbors and when it is on the circumference of $C_1$

are given by the average of the corresponding coordinates of the nodes in $N_v$, that is

$$x = \frac{1}{k}\sum_{i=1}^{k} x_{v_i}, \qquad y = \frac{1}{k}\sum_{i=1}^{k} y_{v_i}.$$

The idea is then to move the node $v$ farther from $p$ but, of course, remaining inside the considered circle. In general this should augment the cost of the MST on the edge connecting the node $v$ to the rest of the tree (see Figure 2).

It can also happen that such a movement completely changes the structure of the MST reducing the initial cost. In that case we do not validate the movement. Given an instance, the augmenting algorithm performs this computation for each node twisting over all the nodes but $s$ till no movements are allowed. As we are going to show, the movements depend also by a random parameter *rand*. Therefore, in order to give to a node a "second chance" to move, we can repeat such computations for a fixed number of rounds. Notice that, when a node reaches the border that is the circumference of the circle, the only allowed movement is over such circumference.

A further way to increase the cost of the MST is then to try to delete a node. We choose as candidate the node with highest degree. The idea behind this choice is that the highest degree node could be considered as the intermediary node to connect its neighbors, so removing it, a "big hole" is luckily to appear. On one hand this means that the distances to connect the remaining disjoint subtrees should increase the overall cost. On the other hand, we are creating more space for further movements. After a deletion, the algorithm starts again with the movements. Indeed the deletion can be considered as a movement in which two nodes are overlapping. If the deletion does not increase the cost of the current MST, we do not validate it. In such a case, the next step, will be the deletion of the second highest degree node and so on. The whole procedure is repeated till no movements and no deletions are allowed. Notice that eventually the whole algorithm can be repeated several consecutive times in order to obtain more accurate results.

We now define more precisely the algorithm roughly described above. Let $V = \{s, v_1, v_2, \ldots, v_n\}$ be a set of nodes inside $C_1$ centered in $s$ and let $\epsilon$ be the

step of the movements we allow, that is, the maximum fraction of the distance from the median point $p$ we allow to move the current point $v$.

**Algo**$(s, V, \epsilon)$

$flag1 = 1$; $\ast$ *flag1 determines if there is an allowed movement anymore.*
$flag2 = 1$; $\ast$ *flag2 determines if there is an allowed deletion anymore.*
$N = |V| - 1$; $\ast$ *Number of available nodes for the augmenting methods.*
$i = 1$;
$j = 1$;
Compute the MST over the complete weighted graph $G$ induced by the set of nodes $V$ in which each edge $\{x, y\}$ has weight $\|x, y\|^2$; save its cost in $cost1$;
**while** $flag2 \leq N$ **do**
  **while** $flag1 \leq N$ **do**
    Consider the node $v_i = \{x_i, y_i\}$ and its $k_i$ neighbors,
    $x = \frac{1}{k_i} \sum_{i=1}^{k_i} x_{v_i}$; $y = \frac{1}{k_i} \sum_{i=1}^{k_i} y_{v_i}$; $\ast$ *Coordinates of the median point p.*
    Let $rand$ be a random number in $[0, 1]$;
    **if** $v_i$ is not on the circumference **then**
      Let $v_i'$ be a point inside $C_1$ on the line passing through $v_i$ and $p$ in such a way that $\|v_i, p\| < \|v_i', p\| \leq (1 + \epsilon \cdot rand)\|v_i, p\|$;
    **else**
      Let $v_i'$ be a point on the circumference further from $p$ with respect to $v_i$ such that the arc joining $v_i$ and $v_i'$ has length $\epsilon \cdot rand$;
    **end if**
    Compute the MST over the complete weighted graph induced by the set of nodes $(V \setminus v_i) \cup v_i'$; save its cost in $cost2$;
    **if** $cost2 > cost1$ **then**
      $V = (V \setminus v_i) \cup v_i'$;
      $cost1 = cost2$;
      $flag1 = 1$;
    **else**
      $flag1 = flag1 + 1$; $\ast$ *The movement is not valid.*
    **end if**
    $i = (i + 1) mod\ N$;
  **end while**
  Let $v_j$ be the $j$-th highest degree node of the current MST, compute the MST over the complete weighted graph induced by the set of nodes $V \setminus v_j$; save its cost in $cost2$.
  **if** $cost2 > cost1$ **then**
    $V = V \setminus v_j$;
    $N = N - 1$;
    $cost1 = cost2$;
    $flag1 = 1$;
    $flag2 = 1$;
  **else**
    $flag2 = flag2 + 1$; $\ast$ *The deletion is not valid.*
  **end if**
  $j = (j + 1) mod\ N$;
**end while**

The validated movements (and deletions) imply a monotonic increasing function on the cost of the MST. Since such a cost is bounded by 6.33 [8], the termination of the algorithm is guaranteed. Actually this is accomplished by the minimal constant growth in each computation given by the minimum performable positive number of the working machine. A strategy to speed up the algorithm could be to modify the $if$ condition of lines 17 and 27 by $cost2 > cost1 + c$, hence introducing a further parameter $c$ that fixes the minimal growth at each augmenting step.

## 4 Experimental Results

We run the algorithm over hundreds of instances from 5 up to 100 nodes. Table 1 resumes the average and the maximum costs obtained on random instances as in previous papers and using our augmenting method for $\epsilon$ equal to .5 and .1. We also repeated the execution of the algorithm two consecutive times for each instance.

Compared to the standard random generated instances, the average costs were almost tripled while the maximum almost doubled. The numerical results obtained are very interesting since they show that standard random instances are not so well representative to study the bounds of the MEBR problem. Moreover, as "side effect" of such experiments, another very interesting obtained property is about the topologies obtained in the augmented instances. While for instances till around 15 nodes our method modifies the nodes distribution tending to the well-known hexagon shape of Figure 1, increasing the number of nodes, things become more and more interesting.

In Figure 3 an instance of 100 nodes is given before and after the movements and deletions. What follows from those experiments is an evident regularity on the final obtained instances. As showed in Figure 3, in general, after the augmen-

**Table 1.** The average and the maximum costs obtained on standard random instances and using the previous augmenting algorithm on instances of 5 up to 100 nodes and $\epsilon$ equal to 0.1 and 0.5

| $n$ | Random | | Augmented, $\epsilon = .5$ | | Augmented, $\epsilon = .1$ | |
|---|---|---|---|---|---|---|
| | Average | Max | Average | Max | Average | Max |
| 5 | 1.301 | 2.8752 | 3.6456 | 4 | 3.6276 | 4 |
| 7 | 1.4799 | 2.4793 | 4.5454 | 5.7386 | 4.5606 | 5.8797 |
| 10 | 1.8019 | 3.1231 | 5.2848 | 5.7851 | 5.353 | 5.9187 |
| 15 | 1.8875 | 2.6691 | 4.8648 | 5.4803 | 4.777 | 5.7728 |
| 20 | 1.854 | 2.6187 | 4.2817 | 5.0906 | 4.1316 | 5.1222 |
| 30 | 1.8252 | 2.2328 | 4.137 | 4.45 | 3.991 | 4.1819 |
| 50 | 1.812 | 1,9718 | 3.7319 | 3,8901 | 3.6331 | 3,7598 |
| 100 | 1.6833 | 1.8829 | 3.5673 | 3.7223 | 3.4898 | 3.812 |

tation, nodes look like disposed on some kind of regular grid. This strengthens the lower bound given by the regular hexagon shape.

It is evident that our method considerably increases the average and the maximum cost of the investigated instances. Moreover, the experiments also suggest to consider regular distributions of the nodes in order to obtain maximal cost instances. In the next section we investigate this property hence obtaining an almost tight 4-approximation upper bound for the MST heuristic in the case of high-density distributions.

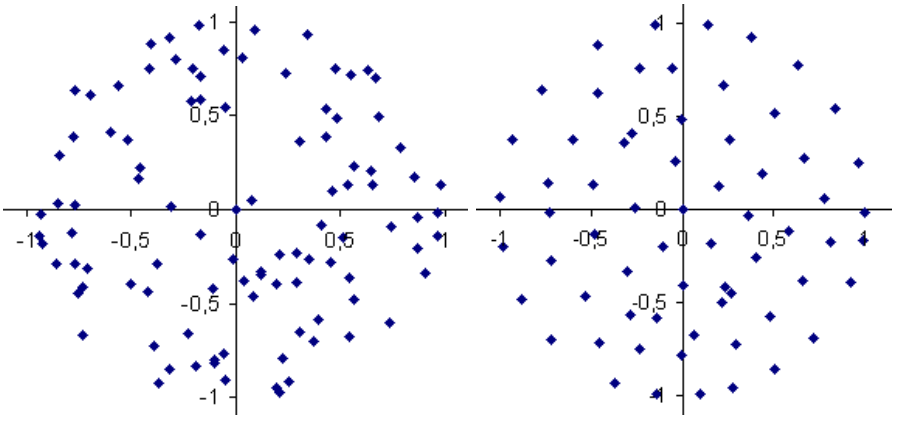



**Fig. 3.** A random instance of 100 nodes before and after the augmenting method. The number of nodes decreased from 100 to 65, while the cost increased from 1.8774 to 3.6809

## 5 High-Density Case

In this section we show that the upper bound of 4 provided in Section 2 for the MST heuristic in the case of $r_{max}$ tending to 0 is almost tight. We provide an example of uniform distribution with high-density of the radio stations in which the cost of the solution returned by the MST heuristic is very close to 4. Actually, this is a significant result, in fact, as already stressed before, it was a common idea, even supported by experimental results, that the MST heuristic is very close to the optimum for the high-density case (see [11, 2, 6, 7, 1, 9]). It is also interesting to notice that the next construction follows directly from the previous experimental results. Such results, in fact, suggest to investigate the case of equidistant nodes in order to increase the cost of the computed MST.

Let us assume an high-density uniform distributions of nodes inside $C_1$ and let the set of nodes $S$ be located on the vertices of a grid composed by equilateral triangles as showed in Figure 4. Roughly speaking, the idea is now to estimate

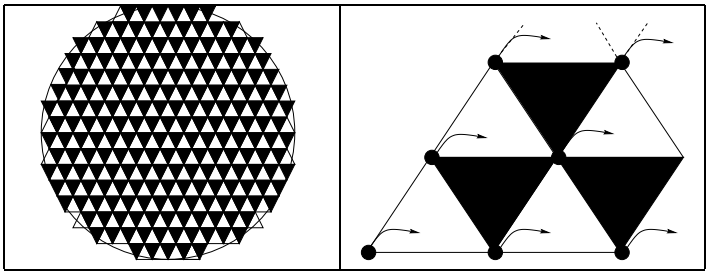


**Fig. 4.** The subdivision of a circle in triangles and the association of each node to a triangle

the cost of the MST heuristic[2] and comparing it with respect to the optimal solution whose cost is upper bounded by the radius of $C_1$ of length 1. Associating a triangle to each node, roughly half of the triangles remain "singles" (the black ones in Figure 4). Since for a given side $l$, the area of an equilateral triangle is equal to $\frac{\sqrt{3}}{4}l^2$, and considering that, by construction, the number of nodes of the MST is equal to the number of its edges plus 1, $\frac{\sqrt{3}}{4}MST(S) \simeq \frac{\pi 1^2}{2}$ and then

$$MST(S) \simeq \frac{2\pi}{\sqrt{3}} > 3.62.$$

The following theorem is then a direct consequence of the above discussions.

**Theorem 1.** *In the* 2*-dimensional Euclidean space, the upper bound on the approximation ratio of the MST heuristic for the Minimum Energy Broadcast Routing problem with high-density distribution of the nodes is between* 3.62 *and* 4.

## 6 Conclusions

We closely examined the MEBR problem by extensive experiments. The main goal was to find special instances in order to maximize the possible cost of the MST heuristic. Motivated by the gap between the theoretical bounds and the values observed by experimental studies, we proposed a new method to generate input instances hence obtaining interesting results. Those experiments, in fact, showed that the usually considered standard random instances are not so well representative for upper bounding the cost of the MST heuristic. Moreover they also suggested how to build expensive instances hence validating the well-known lower bound of 6 for the MEBR problem and the 4 approximation factor in the high-density case.

[2] In the case of regular distribution such as a triangular grid, there exists always an MST composed by a path that visits all the nodes like in Figure 1. Therefore, the maximal cost of the MST heuristic coincides with the cost of the MST.

## References


1. S. Athanassopoulos, I. Caragiannis, C. Kaklamanis, and P. Kanellopoulos. Experimental Comparison of Algorithms for Energy-Efficient Multicasting in Ad Hoc Networks. In *Proceedings of the* $3^{rd}$ *International Conference on Ad-Hoc Networks and Wireless (ADHOC-NOW)*, volume 3158 of *Lecture Notes in Computer Science*, pages 183–196. Springer Verlag, 2004.
2. A. Clementi, G. Huiban, P. Penna, G. Rossi, and Y. C. Verhoeven. On the approximation ratio of the mst-based heuristic for the energy-efficient broadcast problem in static ad-hoc radio networks. In *Proceedings of the* $3^{rd}$ *IEEE IPDPS Workshop on Wireless, Mobile and Ad Hoc Networks (WMAN)*, 2003.
3. A.E.F. Clementi, P. Crescenzi, P. Penna, G. Rossi, and P. Vocca. On the complexity of computing minimum energy consumption broadcast subgraph. In *Proceedings of the* $18^{th}$ *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2010 of *Lecture Notes in Computer Science*, pages 121–131. Springer-Verlag, 2001.
4. M. Flammini, R. Klasing, A. Navarra, and S. Perennes. Improved approximation results for the Minimum Energy Broadcasting Problem. In *Proceedings of ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pages 85–91, 2004.
5. A. M. Frieze and C. J. H. McDiarmid. On Random Minimum Length Spanning Trees. *Combinatorica*, 9:363–374, 1989.
6. R. Klasing, A. Navarra, A. Papadopoulos, and S. Perennes. Adaptive Broadcast Consumption (ABC), a new heuristic and new bounds for the minimum energy broadcast routing problem. In *Proceedings of the* $3^{rd}$ *IFIP-TC6 International Networking Conference*, volume 3042 of *Lecture Notes in Computer Science*, pages 866–877. Springer Verlag, 2004.
7. F. J. O. Martnez, I. Stojmenovic, F. G. Nocetti, and J. S. Gonzalez. Finding Minimum Transmission Radii for Preserving Connectivity and Constructing Minimal Spanning Trees in Ad Hoc and Sensor Networks. In *Proceedings of the* $3^{rd}$ *International Workshop on Experimental and Efficient Algorithms (WEA)*, volume 3059 of *Lecture Notes in Computer Science*, pages 369–382. Springer Verlag, 2004.
8. A. Navarra. Tighter bounds for the Minimum Energy Broadcasting problem. In *Proceedings of the* $3^{rd}$ *International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*, 2005, to appear.
9. P. Penna and C. Ventre. Energy-efficient broadcasting in ad-hoc networks: combining msts with shortest-path trees. In *Proceedings of the* $1^{st}$ *ACM International Workshop on Performance Evaluation of Wireless, Ad Hoc, Sensor and Ubiquitous Networks (PE-WASUN)*, pages 61–68. ACM Press, 2004.
10. P. J. Wan, G. Calinescu, X. Li, and O. Frieder. Minimum energy broadcasting in static ad hoc wireless networks. *Wireless Networks*, 8(6):607–617, 2002. Extended abstract appeared in *Proceedings of the* $20^{th}$ *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (2001).
11. J. E. Wieselthier, G. D. Nguyen, and A. Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *Proceedings of the* $19^{th}$ *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 585–594. IEEE Computer Society, 2000.

# Implementing Minimum Cycle Basis Algorithms

Kurt Mehlhorn and Dimitrios Michail

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{mehlhorn, michail}@mpi-sb.mpg.de

**Abstract.** In this paper we consider the problem of computing a minimum cycle basis of an undirected graph $G = (V, E)$ with $n$ vertices and $m$ edges. We describe an efficient implementation of an $O(m^3 + mn^2 \log n)$ algorithm presented in [1]. For sparse graphs this is the currently best known algorithm. This algorithm's running time can be partitioned into two parts with time $O(m^3)$ and $O(m^2 n + mn^2 \log n)$ respectively. Our experimental findings imply that the true bottleneck of a sophisticated implementation is the $O(m^2 n + mn^2 \log n)$ part. A straightforward implementation would require $\Omega(nm)$ shortest path computations, thus we develop several heuristics in order to get a practical algorithm. Our experiments show that in random graphs our techniques result in a significant speedup.

Based on our experimental observations, we combine the two fundamentally different approaches to compute a minimum cycle basis used in [1, 2] and [3, 4], to obtain a new hybrid algorithm with running time $O(m^2 n^2)$. The hybrid algorithm is very efficient in practice for random dense unweighted graphs.

Finally, we compare these two algorithms with a number of previous implementations for finding a minimum cycle basis in an undirected graph.

## 1 Introduction

Let $G = (V, E)$ be an undirected graph. A *cycle* of $G$ is any subgraph in which each vertex has even degree. Associated with each cycle is an *incidence vector* $x$, indexed on $E$, where $x_e = 1$ if $e$ is an edge of $C$, $x_e = 0$ otherwise. The vector space over $GF(2)$ generated by the incidence vectors of cycles is called the *cycle space* of $G$. It is well-known that this vector space has dimension $N = m - n + \kappa$, where $m$ is the number of edges, $n$ is the number of vertices, and $\kappa$ the number of connected components of $G$. A maximal set of linearly independent cycles is called a *cycle basis*.

The edges of $G$ have non-negative weights. The weight of a cycle is the sum of the weights of its edges. The weight of a cycle basis is the sum of the weights of its cycles. We consider the problem of computing a cycle basis of minimum weight in a graph; we use the abbreviation MCB to refer to a minimum cycle basis.

The problem has been extensively studied, both in its general setting and in special classes of graphs. Its importance lies in its use as a preprocessing step

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 32–43, 2005.
© Springer-Verlag Berlin Heidelberg 2005

in several algorithms. Such algorithms include diverse applications like electrical circuit theory [5], structural engineering [6] and periodic event scheduling [1].

The first polynomial time algorithm for the minimum cycle basis problem was given by Horton [3] with running time $O(m^3n)$. de Pina [1] gave an $O(m^3 + mn^2 \log n)$ algorithm by using a different approach. Golynski and Horton [4] improved Horton's algorithm to $O(m^\omega n)$ by using fast matrix multiplication. It is presently known [7] that $\omega < 2.376$. Recently Berger et al. [8] gave another $O(m^3 + mn^2 \log n)$ algorithm by using similar ideas as de Pina. Finally, Kavitha et al. [2] improved de Pina's algorithm into $O(m^2n + mn^2 \log n)$ again by using fast matrix multiplication. In the same paper a faster $1 + \epsilon$ approximation algorithm, for any $\epsilon > 0$, is presented.

In this paper we report our experimental findings from our implementation of the $O(m^3 + mn^2 \log n)$ algorithm presented in [1]. Our implementation uses LEDA [9]. We develop a set of heuristics which improve the best-case performance of the algorithm while maintaining its asymptotics. Finally, we consider a hybrid algorithm obtained by combining the two different approaches used in [1, 2] and [3, 4] with running time $O(m^2n^2)$, and compare the implementations. The new algorithm is motivated by our need to reduce the cost of the shortest path computations. The resulting algorithm seems to be very efficient in practice for random dense unweighted graphs. Finally, we compare our implementations with previous implementations of minimum cycle basis algorithms [3, 8].

The paper is organized as follows. In Section 2 we briefly describe the algorithms.In Section 2.1 we describe our heuristics and in 2.2 we present our new algorithm. In Section 3 we give and discuss our experimental results.

## 2 Algorithms

Let $G(V, E)$ be an undirected graph with $m$ edges and $n$ vertices. Let $l : E \mapsto \mathbb{R}_{\geq 0}$ be a non-negative length function on the edges. Let $\kappa$ be the number of connected components of $G$ and let $T$ be any spanning forest of $G$. Also let $e_1, \ldots, e_N$ be the edges of $G \setminus T$ in some arbitrary but fixed order. Note that $N = m - n + \kappa$ is exactly the dimension of the cycle space.

The algorithm [1] computes the cycles of an MCB and their *witnesses*. A witness $S$ of a cycle $C$ is a subset of $\{e_1, \ldots, e_N\}$ which will prove that $C$ belongs to the MCB. We view these subsets in terms of their incidence vectors over $\{e_1, \ldots, e_m\}$. Hence, both cycles and witnesses are vectors in the space $\{0, 1\}^m$. $\langle C, S\rangle$ stands for the standard inner product of vectors $C$ and $S$. Since we are at the field $GF(2)$ observe that $\langle C, S\rangle = 1$ if and only if the intersection of the two edge sets has odd cardinality. Finally, adding two vectors $C$ and $S$ in $GF(2)$ is the same as the symmetric difference of the two edge sets. Algorithm 1 gives a full description.

The algorithm in phase $i$ has two parts, one is the computation of the cycle $C_i$ and the second part is the update of the sets $S_j$ for $j > i$. Note that updating the sets $S_j$ for $j > i$ is nothing more than maintaining a basis $\{S_{i+1}, \ldots, S_N\}$ of the subspace orthogonal to $\{C_1, \ldots, C_i\}$.

**Algorithm 1** Construct an MCB

```
Set S_i = {e_i} for all i = 1, ..., N.
for i = 1 to N do
  Find C_i as the shortest cycle in G s.t ⟨C_i, S_i⟩ = 1.
  for j = i + 1 to N do
    if ⟨S_j, C_i⟩ = 1 then
      S_j = S_j + S_i
    end if
  end for
end for
```

*Computing the Cycles.* Given $S_i$, it is easy to compute a shortest cycle $C_i$ such that $\langle C_i, S_i\rangle = 1$ by reducing it to $n$ shortest path computations in an appropriate graph $G_i$. The following construction is well-known.

$G_i$ has two copies $v^+$ and $v^-$ of each vertex $v \in V$. For each edge $e = (u, v) \in E$ do: if $e \notin S_i$, then add edges $(u^+, v^+)$ and $(u^-, v^-)$ to the edge set of $G_i$ and assign their weights to be the same as $e$. If $e \in S_i$, then add edges $(u^+, v^-)$ and $(u^-, v^+)$ to the edge set of $G_i$ and assign their weights to be the same as $e$. $G_i$ can be visualized as 2 levels of $G$ (the + level and the − level). Within each level, we have edges of $E \setminus S_i$. Between the levels we have the edges of $S_i$. Call $G_i$, the *signed* graph.

Any $v^+$ to $v^-$ path $p$ in $G_i$ corresponds to a cycle in $G$ by identifying edges in $G_i$ with their corresponding edges in $G$. If an edge $e \in G$ occurs multiple times we include it if the number of occurrences of $e$ modulo 2 is 1. Because we identify $v^+$ and $v^-$ with $v$, the path in $G$ resulting from $p$ is a cycle $C$. Since we start from a positive vertex and end in a negative one, the cycle has to change sign an odd number of times and therefore uses an odd number of edges from $S_i$. In order to find a shortest cycle, we compute a shortest path from $v^+$ to $v^-$ for all $v \in V$.

*Running Time.* In each phase we have the shortest path computations which take time $O(n(m + n \log n))$ and the update of the sets which take $O(m^2)$ time. We execute $O(m)$ phases and therefore the running time is $O(m^3 + m^2 n + mn^2 \log n)$.

### 2.1 Heuristic Improvements

In this section we present several heuristics which can improve the running time substantially. All heuristics preserve the worst-case time and space bounds.

*Compressed representation (H1)* All vectors (sets $S$ and cycles $C$) which are handled by the algorithm are in $\{0, 1\}^m$. Moreover, any operations performed are normal set operations. This allows us to use a compressed representation where each entry of these vectors is represented by a bit of an integer. This allows us to save up space and at the same time to perform 32 or 64 bitwise operations in parallel.

*Upper bounding the shortest path (H2)* During phase $i$ we might perform up to $n$ shortest path computations in order to compute the shortest cycle $C_i$ with an odd intersection with the set $S_i$. Following similar observations of [10] we can use the shortest path found so far as an upper bound on the shortest path. This is implemented as follows; a node is only added in the priority queue of Dijkstra's implementation if its current distance is not more than our current upper bound.

*Reducing the shortest path computations (H3)* We come to the most important heuristic. In each of the $N$ phases we are performing $n$ shortest path computations. This results to $\Omega(mn)$ shortest path computations.

Let $S = \{e_1, e_2, \ldots, e_k\}$ be a *witness* at some point of the execution. We need to compute the shortest cycle $C$ s.t $\langle C, S\rangle = 1$. We can reduce the number of shortest path computations based on the following observation.

Let $C_{\geq i}$ be the shortest cycle in $G$ s.t $\langle C_{\geq i}, S\rangle = 1$, and $C_{\geq i} \cap \{e_1, \ldots, e_{i-1}\} = \emptyset$, and $e_i \in C_{\geq i}$. Then cycle $C$ can be expressed as $C = \min\limits_{i=1,\ldots,k} C_{\geq i}$. We can compute $C_{\geq i}$ in the following way. We delete edges $\{e_1, \ldots, e_i\}$ from the graph $G$ and the corresponding edges from the signed graph $G_i$. Let $e_i = (v, u) \in G$. Then we compute a shortest path in $G_i$ from $v^+$ to $u^+$. The path computed will have an even number of edges from the set $S$, and together with $e_i$ an odd number. Since we deleted edges $\{e_1, \ldots, e_i\}$ the resulting cycle does not contain any edges from $\{e_1, \ldots, e_{i-1}\}$.

Using the above observation we can compute each cycle in $O(kSP(n, m))$ time when $|S| = k < n$ and in $O(nSP(n, m))$ when $|S| \geq n$. Thus the running time for the cycles computations is equal to $SP(m, n) \cdot \sum_{i=1,\ldots,N} min\{n, |S_i|\}$ where $SP(m, n)$ is the time to compute a single-source shortest path on an undirected weighted graph with $m$ edges and $n$ vertices.

### 2.2 A New Hybrid Algorithm

The first polynomial algorithm [3] developed, did not compute the cycles one by one but instead computed a superset of the MCB and then greedily extracted the MCB by Gaussian elimination. This superset contains $O(mn)$ cycles which are constructed in the following way.

For each vertex $v$ and edge $e = (u, w)$, construct the cycle $C = SP(v, u) + SP(v, w) + (u, w)$ where $SP(a, b)$ is the shortest path from $a$ to $b$. If these two shortest paths do not contain a vertex other than $v$ in common then keep the cycle otherwise discard it. Let us call this set of cycles the *Horton set*. It was shown in [3] that the Horton set always contains an MCB. However, not every MCB is contained in the Horton set.

Based on the above and motivated by the need to reduce the cost of the shortest path computations we developed a new algorithm, which combines the two approaches. That is, compute the Horton set and extract the MCB not by using Gaussian elimination which would take time $O(m^3 n)$ but by using the orthogonal space of the cycle space as we did in Section 2. The Horton set contains an MCB but not necessarily all the cycles that belong to any MCB. We

resolve this difficulty by ensuring uniqueness of the MCB. We ensure uniqueness by ensuring uniqueness of the shortest path distances on the graph (either by perturbation or by lexicographic ordering). After the preprocessing step, every cycle of the MCB will be contained in the Horton set and therefore we can query the superset for the cycles instead of the graph $G$. A succinct description can be found in Algorithm 2.

**Algorithm 2** Hybrid MCB algorithm

```
Ensure uniqueness of shortest path distances of G ( lexicographically or by perturbation).
Construct superset (Horton set) S of MCB.
Set S_i = {e_i} for all i = 1, ..., N.
for i = 1 to N do
  Find C_i as the shortest cycle in S s.t ⟨C_i, S_i⟩ = 1.
  for j = i + 1 to N do
    if ⟨S_j, C_i⟩ = 1 then
      S_j = S_j + S_i
    end if
  end for
end for
```

The above algorithm has worst case running time $O(m^2n^2)$. This is because the Horton set contains at most $mn$ cycles, we need to search for at most $m$ cycles and each cycle contains at most $n$ edges. The important property of this algorithm is that the time to actually compute the cycles is only $O(n^2m)$, which is by a factor of $\frac{m}{n} + \log n$ better than the $O(m^2n + mn^2 \log n)$ time required by Algorithm 1. Together with the experimental observation that in general the linear independence step is not the bottleneck, we actually hope to have developed a very efficient algorithm.

## 3 Experiments

We perform several experiments in order to understand the running time of the algorithms using the previously presented heuristics. In order to understand the speedup obtained, especially from the use of the *H3* heuristic, we study in more detail the cardinalities of the sets $S$ during the algorithm as well as how many operations are required in order to update these sets. We also compare the running times of Algorithms 1 and 2, with previous implementations.

All experiments are done using random sparse and dense graphs. All graphs were constructed using the $G(n; p)$ model, for $p = 4/n, 0.3, 0.5$ and $0.9$. Our implementation uses LEDA [9]. All experiments were performed on a Pentium 1.7Ghz machine with 1 GB of memory, running GNU/Linux. We used the GNU g++ 3.3 compiler with the -O optimization flag. All other implementations, use the boost C++ libraries [11].

### 3.1 Updating $S_i$'s

In this section we present experimental results which suggest that the dominating factor of the running time of Algorithm 1 (at least for random graphs) is not the time needed to update the sets $S$ but the time to compute the cycles.

Note that the time to update the sets is $O(m^3)$ and the time to compute the cycles is $O(m^2n + mn^2 \log n)$, thus on sparse graphs this algorithm has the same running time $O(n^3 \log n)$ as the fastest known. The currently fastest algorithm [2] for the MCB problem has running time $O(m^2n + mn^2 \log n + m^\omega)$; the $m^\omega$ factor is dominated by the $m^2n$ but we present it here in order to understand what type of operations the algorithm performs. This algorithm improves upon [1] w.r.t the time needed to update the sets $S$ by using fast matrix multiplication techniques.

Although fast matrix multiplication can be practical for medium and large sized matrices, our experiments show that the time needed to update the sets $S$ is a small fraction of the time needed to compute the cycles. Figure 1 presents a comparison of the required time to update the sets $S_i$ and to calculate the cycles $C_i$ by using the signed graph for random weighted graphs.

In order to get a better understanding of this fact, we performed several experiments. As it turns out, in practice, the average cardinality of the sets $S$ is much less than $N$ and moreover the number of times we actually perform set updates (if $\langle C_i, S_j \rangle = 1$) is much less than $N(N-1)/2$. Moreover, heuristic *H1* decreases the constant factor of the running time (for updating $S$'s) substantially by performing 32 or 64 operations in parallel. This constant factor

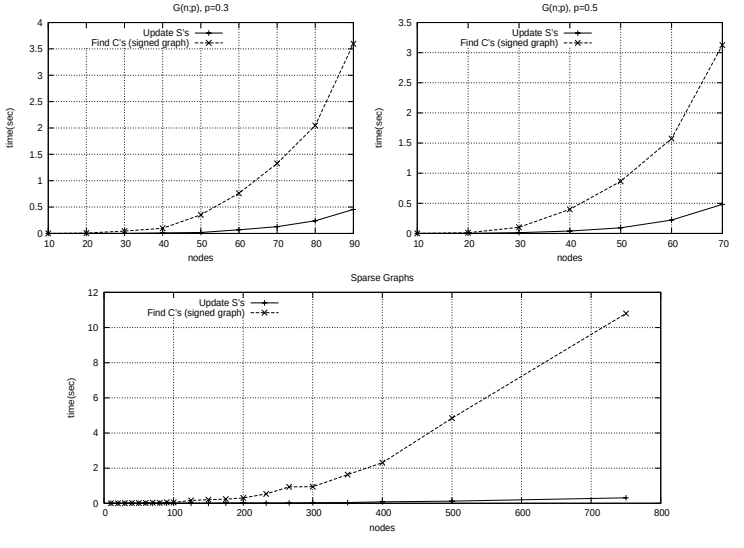



**Fig. 1.** Comparison of the time taken to update the sets $S$ and the time taken to calculate the cycles on random weighted graphs, by Algorithm 1

**Table 1.** Statistics about sets $S$ sizes on sparse random graphs with $p = 4/n$ and dense random graphs for $p = 0.3$ and 0.5. Sets are considered during the whole execution of the algorithm. Column $\#\langle S, C\rangle = 1$ denotes the number of updates performed on the sets $S$. An upper bound on this is $N(N-1)/2$, which we actually use when bounding the algorithm's running time. Note that the average cardinality of $S$ is very small compared to $N$ although the maximum cardinality of some $S$ is in $O(N)$

| $n$ | $m$ | $N$ | $N(N-1)/2$ | $max(\|S\|)$ | $avg(\|S\|)$ | $\#\ \langle S, C\rangle = 1$ |
|---|---|---|---|---|---|---|
| sparse ($m \approx 2n$) | | | | | | |
| 10 | 19 | 10 | 45 | 4 | 2 | 8 |
| 104 | 208 | 108 | 5778 | 44 | 4 | 258 |
| 491 | 981 | 500 | 124750 | 226 | 7 | 2604 |
| 963 | 1925 | 985 | 484620 | 425 | 7 | 5469 |
| 2070 | 4139 | 2105 | 2214460 | 1051 | 13 | 20645 |
| 4441 | 8882 | 4525 | 10235550 | 2218 | 17 | 58186 |
| $p = 0.3$ | | | | | | |
| 10 | 13 | 4 | 6 | 2 | 2 | 2 |
| 25 | 90 | 66 | 2145 | 27 | 3 | 137 |
| 75 | 832 | 758 | 286903 | 370 | 6 | 3707 |
| 150 | 3352 | 3203 | 5128003 | 1535 | 9 | 22239 |
| 200 | 5970 | 5771 | 16649335 | 2849 | 10 | 49066 |
| 300 | 13455 | 13156 | 86533590 | 6398 | 10 | 116084 |
| 500 | 37425 | 36926 | 681746275 | 18688 | 14 | 455620 |
| $p = 0.5$ | | | | | | |
| 10 | 22 | 13 | 78 | 7 | 2 | 14 |
| 25 | 150 | 126 | 7875 | 57 | 4 | 363 |
| 75 | 1387 | 1313 | 861328 | 654 | 6 | 6282 |
| 150 | 5587 | 5438 | 14783203 | 2729 | 9 | 39292 |
| 200 | 9950 | 9751 | 47536125 | 4769 | 11 | 86386 |
| 300 | 22425 | 22126 | 244768875 | 10992 | 13 | 227548 |
| 500 | 62375 | 61876 | 1914288750 | 30983 | 15 | 837864 |

decrease does not concern the shortest path computations. Table 1 summarizes our results.

### 3.2 Number of Shortest Path Computations

Heuristic *H3* improves the best case of the algorithm, while maintaining at the same time the worst case. Instead of $\Omega(nm)$ shortest path computations we hope to perform much less. In Table 2 we study the sizes of the sets $S_i$ for $i = 1, \ldots, N$ used to calculate the cycles for sparse and dense graphs respectively.

In both sparse and dense graphs although the maximum set can have quite large cardinality, the average set size is much less than $n$. Moreover, in sparse graphs every set used has cardinality less than $n$. On dense graphs the sets with cardinality less than $n$ are more than 95% percent. This implies a significant speedup due to the *H3* heuristic.

Figure 2 compares the running times of Algorithm 1 with and without the *H3* heuristic. As can easily be seen the improvement is more than a constant factor.

**Table 2.** Statistics about sets $S_i$ sizes on sparse random graphs with $p = 4/n$ and dense random graphs for $p = 0.3$ and 0.5, at the moment we calculate cycle $C_i$

| n | m | N | $max(\|S_i\|)$ | $\lceil avg(\|S_i\|)\rceil$ | $\|\{S_i : \|S_i\| < n\}\|$ |
|---|---|---|---|---|---|
| sparse ($m \approx 2n$) | | | | | |
| 10 | 19 | 10 | 4 | 2 | 10 |
| 104 | 208 | 108 | 39 | 5 | 108 |
| 491 | 981 | 498 | 246 | 13 | 498 |
| 963 | 1925 | 980 | 414 | 11 | 980 |
| 2070 | 4139 | 2108 | 1036 | 27 | 2108 |
| 4441 | 8882 | 4522 | 1781 | 33 | 4522 |
| $p = 0.3$ | | | | | |
| 10 | 13 | 4 | 2 | 2 | 4 |
| 25 | 90 | 66 | 20 | 4 | 66 |
| 75 | 832 | 758 | 357 | 15 | 721 |
| 150 | 3352 | 3203 | 1534 | 18 | 3133 |
| 200 | 5970 | 5771 | 2822 | 29 | 5635 |
| 300 | 13455 | 13156 | 6607 | 32 | 12968 |
| 500 | 37425 | 36926 | 15965 | 39 | 36580 |
| $p = 0.5$ | | | | | |
| 10 | 22 | 13 | 7 | 3 | 13 |
| 25 | 150 | 126 | 66 | 5 | 121 |
| 75 | 1387 | 1313 | 456 | 10 | 1276 |
| 150 | 5587 | 5438 | 2454 | 19 | 5338 |
| 200 | 9950 | 9751 | 4828 | 28 | 9601 |
| 300 | 22425 | 22126 | 10803 | 33 | 21875 |
| 500 | 62375 | 61876 | 30877 | 38 | 61483 |

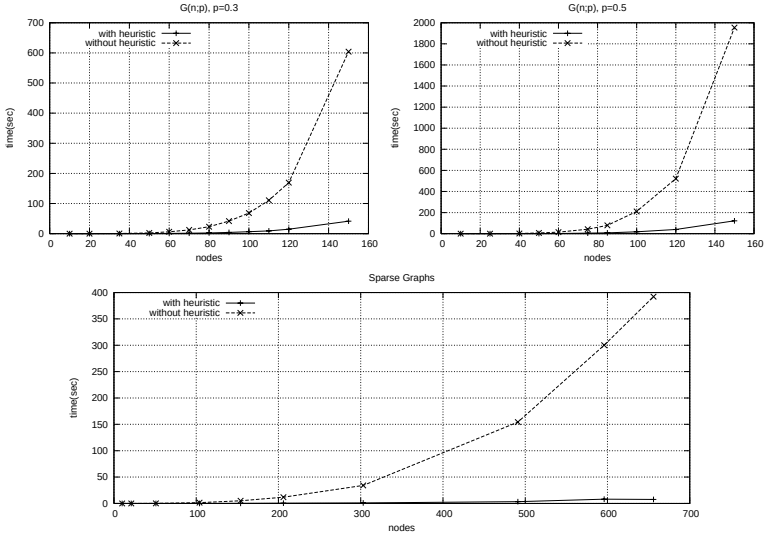



**Fig. 2.** Running times of Algorithm 1 with and without the *H3* heuristic. Without the heuristic the algorithm is forced to perform $\Omega(nm)$ shortest path computations

### 3.3 Running Time

In this section we compare the various implementations for computing a minimum cycle basis. Except for Algorithms 1 (DP) and 2 (HYB) we include in the

comparison two implementations [12, 13] (HOR) of Horton's algorithm with running time $O(m^3n)$ and an implementation [12] (FEAS) of the $O(m^3 + mn^2 \log n)$ algorithm presented in [8]. Algorithms 1 and 2 are implemented with compressed integer sets. Fast matrix multiplication [2, 4] can nicely improve many parts of these implementations with respect to the worst case complexity. We did not experiment with these versions of the algorithms.

The comparison of the running times is performed for three different type of undirected graphs: (a) random sparse graphs, where $m \approx 2n$, (b) random graphs from $G(n;p)$ with different density $p = 0.3, 0.5, 0.9$ and (c) hypercubes. Tests are performed for both weighted and unweighted graphs. In the case of

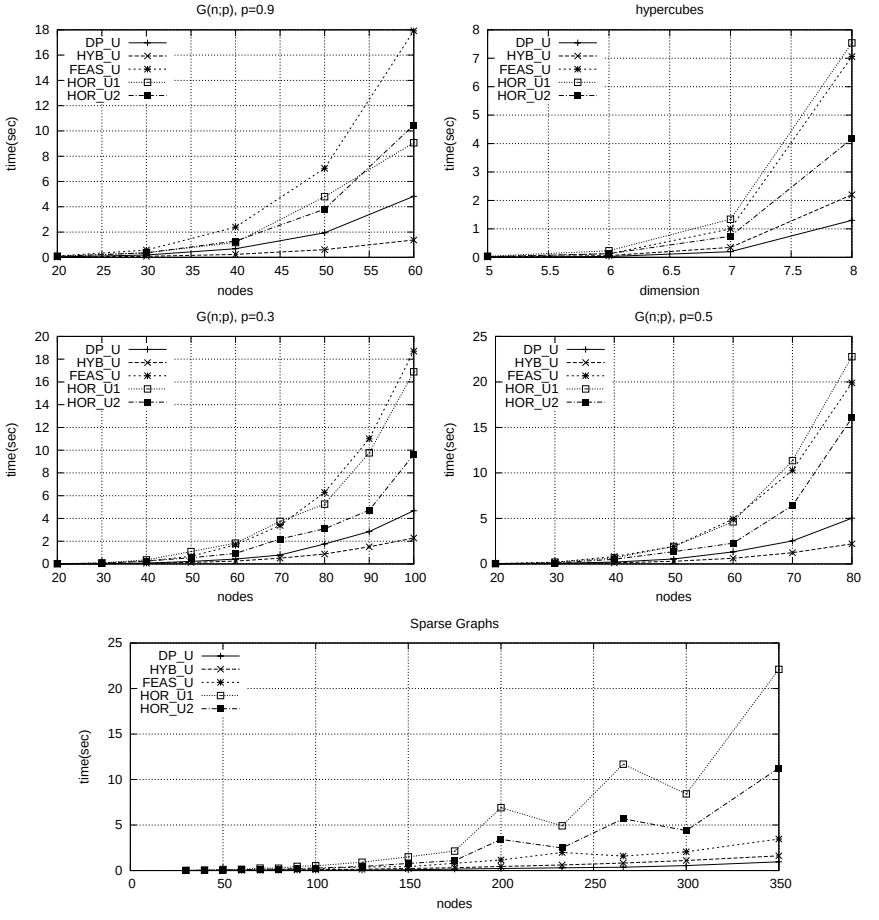



**Fig. 3.** Comparison of various algorithms for random *unweighted* graphs. Algorithm 1 is denoted as *DP_U* and Algorithm 2 as *HYB_U*. *HOR_U1* [12] and *HOR_U2* [13] are two different implementation of Horton's [3] algorithm. *FEAS_U* is an implementation [12] of an $O(m^3)$ algorithm described in [8]

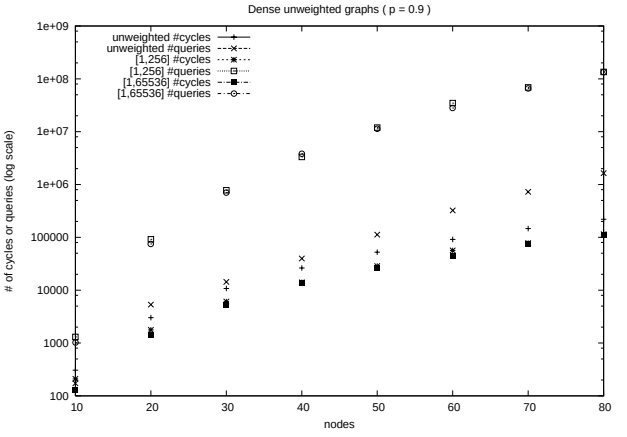



**Fig. 4.** Number of cycles in the Horton set (set with duplicates) and number of queries required in this set (set sorted by cycle weight) in order to extract the MCB for random dense graphs with random weights of different ranges. Each random graph is considered with three different edge weight ranges: (a) unweighted, (b) weights in $[1, 2^8]$, (c) weights in $[1, 2^{16}]$

weighted graphs the weight of an edge is an integer chosen independently at random from the uniform distribution in the range $[0 \dots 2^{16}]$.

Figures 3 and 5 summarize the results of these comparisons. In the case of weighted graphs Algorithm 1 is definitely the winner. On the other hand in the case of dense unweighted graphs Algorithm 2 performs much better. As can be easily observed the differences on the running time of the implementations are rather small for sparse graphs. For dense graphs however, we observe a substantial difference in performance.

*Dense Unweighted Graphs.* In the case of dense unweighted graphs, the hybrid algorithm performs better than the other algorithms. However, even on the exact same graph, the addition of weights changes the performance substantially. This change in performance is not due to the difference in size of the produced Horton set, between the unweighted and the weighted case, but due to the total number of queries that have to be performed in this set.

In the hybrid algorithm before computing the MCB, we sort the cycles of the Horton set. Then for each of the $N$ phases, we *query* the Horton set from the least costly cycle to the most, until we find a cycle with an odd intersection with our current witness $S$. Figure 4 plots for dense graphs the number of cycles in the Horton set and the number of queries required in order to extract the MCB from this set. In the case of unweighted graphs, the number of queries is substantially smaller than in the case of weighted graphs. This is exactly the reason why the hybrid algorithm outperforms the others in unweighted dense graphs.

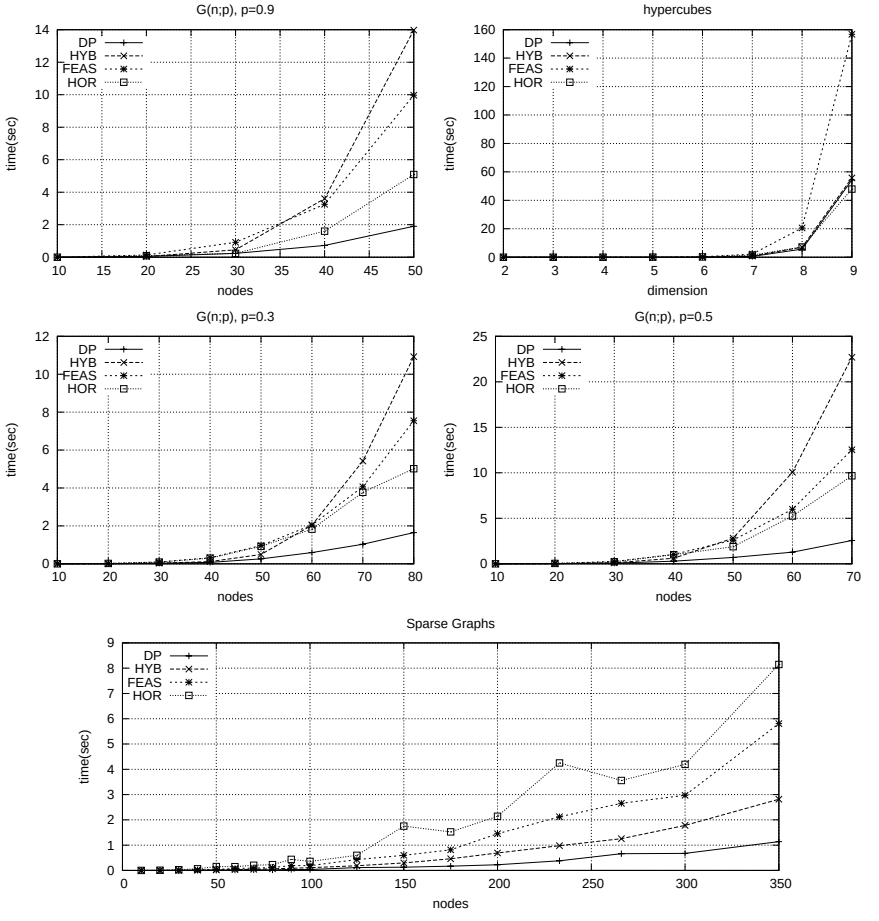



**Fig. 5.** Comparison of various algorithms for random *weighted* graphs. Algorithm 1 is denoted as *DP* and Algorithm 2 as *HYB*. *HOR* [12] is Horton's [3] algorithm. *FEAS* is an implementation [12] of an $O(m^3 + mn^2 \log n)$ algorithm described in [8]

## References

1. de Pina, J.: Applications of Shortest Path Methods. PhD thesis, University of Amsterdam, Netherlands (1995)
2. Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.E.: A faster algorithm for minimum cycle basis of graphs. In: 31st International Colloquium on Automata, Languages and Programming, Finland. (2004) 846–857
3. Horton, J.D.: A polynomial-time algorithm to find a shortest cycle basis of a graph. SIAM Journal of Computing **16** (1987) 359–366
4. Golynski, A., Horton, J.D.: A polynomial time algorithm to find the minimum cycle basis of a regular matroid. In: 8th Scandinavian Workshop on Algorithm Theory. (2002)

5. Chua, L.O., Chen, L.: On optimally sparse cycle and coboundary basis for a linear graph. IEEE Trans. Circuit Theory **CT-20** (1973) 495–503
6. Cassell, A.C., Henderson, J.C., Ramachandran, K.: Cycle bases of minimal measure for the structural analysis of skeletal structures by the flexibility method. Proc. Royal Society of London Series A **350** (1976) 61–70
7. Coppersmith, D., Winograd, S.: Matrix multiplications via arithmetic progressions. Journal of Symb. Comput. **9** (1990) 251–280
8. Berger, F., Gritzmann, P., , de Vries, S.: Minimum cycle basis for network graphs. Algorithmica **40** (2004) 51–62
9. Mehlhorn, K., Naher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
10. Bast, H., Mehlhorn, K., Schäfer, G.: A heuristic for dijkstra's algorithm with many targets and its use in weighted matching algorithms. Algorithmica **36** (2003) 75–88
11. Boost: C++ Libraries. (2001) `http://www.boost.org`.
12. Kreisbasenbibliothek: CyBaL. (2004) `http://www-m9.ma.tum.de/dm/cycles/cybal`.
13. Huber, M.: Implementation of algorithms for sparse cycle bases of graphs. (2002) `http://www-m9.ma.tum.de/dm/cycles/mhuber`.

# Rounding to an Integral Program*

Refael Hassin and Danny Segev

School of Mathematical Sciences,
Tel-Aviv University, Tel-Aviv 69978, Israel
{hassin, segevd}@post.tau.ac.il

**Abstract.** We present a general framework for approximating several NP-hard problems that have two underlying properties in common. First, the problems we consider can be formulated as integer covering programs, possibly with additional side constraints. Second, the number of covering options is restricted in some sense, although this property may be well hidden. Our method is a natural extension of the *threshold rounding* technique.

## 1 Introduction

We present a general framework for approximating several special cases of NP-hard problems that have two underlying properties in common. First, the problems we consider can be formulated as integer covering programs, possibly with additional constraints that control the interaction between the variables. Second, the number of covering options is restricted in some sense, although this property may be well hidden.

Our method is based on the rectangle stabbing algorithm of Gaur, Ibaraki and Krishnamurti [4], and can be viewed as an extension of the *threshold rounding* technique, introduced by Hochbaum [9] for the vertex cover problem. Given an integer programming formulation of the problem, $\min\{c^T x : Ax \geq b, x \in \{0,1\}^n\}$, this approach first relaxes the integrality constraints to obtain the linear program $\min\{c^T x : Ax \geq b, x \in [0,1]^n\}$. The optimal fractional solution $x^*$ to this program is then rounded to an integral one by setting each variable to 1 if its value is at least $\lambda$, and to 0 otherwise, for a threshold parameter $\lambda$.

Since threshold rounding by itself does not guarantee any non-trivial approximation ratio for the problems we study, we strengthen this method as follows. Instead of rounding $x^*$ to an integral solution, we round $x^*$ to an *integral program*. In other words, using $x^*$ and a threshold parameter $\lambda$, we construct a new linear program $\min\{c^T x : A^* x \geq b^*, x \in [0,1]^n\}$ with the following two structural properties:

1. **Feasibility:** Any feasible solution to the new linear program is also feasible to the original program.
2. **Integrality:** The extreme points of the polyhedron $P^* = \{x : A^* x \geq b^*, x \in [0,1]^n\}$ are integral.

It follows that the new linear program can be solved to obtain an integral solution $\hat{x}$ to the original integer program. We prove that the cost of $\hat{x}$ is within factor $\frac{1}{\lambda}$ of optimum by fitting $x^*$ into the polyhedron $P^*$, that is, we show that $\frac{1}{\lambda} x^* \in P^*$.

* Due to space limitations, we defer most proofs to the full version of this paper.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 44–54, 2005.
© Springer-Verlag Berlin Heidelberg 2005

### 1.1 Applications

SET COVER WITH $k$ BLOCKS OF CONSECUTIVE ONES. Let $U = \{e_1, \ldots, e_n\}$ be a ground set of elements, and let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a collection of subsets of $U$, where each subset $S_i$ is associated with a non-negative cost $c_i$. The objective of the set cover problem is to find a minimum cost subcollection $\mathcal{S}' \subseteq \mathcal{S}$ that covers all elements. We consider a special case of set cover, in which there is a known order $\mathcal{O}$ on $\{1, \ldots, m\}$ such that the subsets covering each element $e \in U$ form at most $k$ "blocks" in $S_{\mathcal{O}(1)}, \ldots, S_{\mathcal{O}(m)}$. More precisely, there are indices $l_1^e \leq r_1^e \leq \cdots \leq l_k^e \leq r_k^e$ such that the collection of subsets to which $e$ belongs is $\bigcup_{t=1}^{k} \bigcup_{i=l_t^e}^{r_t^e} \{S_{\mathcal{O}(i)}\}$.

MULTI-RADIUS COVER. Let $G = (V, E)$ be a graph with a non-negative edge length $l_{u,v}$ for every $(u, v) \in E$. The vertices of $G$ represent locations at which transmission stations are positioned, and each edge of $G$ represents a continuum of demand points to which we should transmit. A station located at $v$ is associated with a set of allowed transmission radii $R_v = \{r_0^v, \ldots, r_{k_v}^v\}$, $0 = r_0^v < \cdots < r_{k_v}^v$, where the cost of transmitting to radius $r_i^v$ is $b_{v,i}$. Without loss of generality, $0 = b_{v,0} < \cdots < b_{v,k_v}$ for every $v \in V$. The multi-radius cover problem asks to determine for each station a transmission radius, such that for each edge $(u, v) \in E$ the sum of the radii in $u$ and $v$ is at least $l_{u,v}$, and such that the total cost is minimized.

RECTANGLE STABBING. Let $R = \{r_1, \ldots, r_n\}$ be a set of axis-parallel rectangles, and let $H$ and $V$ be finite sets of horizontal and vertical lines, respectively, where each line $l$ has a non-negative weight $w(l)$. The objective of the rectangle stabbing problem is to find a minimum weight subset of lines in $H \cup V$ that intersects all rectangles in $R$.

GROUP CUT ON A PATH. Let $P = (V, E)$ be a path, in which each edge $e \in E$ is associated with a non-negative cost $c_e$, and let $G_1, \ldots, G_k$ be $k$ *groups*, where each group is a set of at least two vertices. A group $G_i$ is *separated by the set of edges* $F \subseteq E$ if there is a representative $v_i \in G_i$ such that no vertex in $G_i \setminus \{v_i\}$ belongs to the connected component of $P - F$ that contains $v_i$. The objective of the group cut on a path problem (GCP) is to find a minimum cost set of edges that separates all groups. We consider two special cases of this problem: The case where the vertices of each group appear consecutively on the path, and the case where the cardinality of each group is at most $d$.

FEASIBLE CUT. Let $G = (V, E)$ be a graph with a non-negative cost $c_e$ for every $e \in E$. In addition, let $S_1, \ldots, S_k$ be a collection of $k$ *commodities*, where each commodity is a set of vertices, and let $v^* \in V$. A cut $(X, \bar{X})$ is *feasible* if it separates $v^*$ from at least one vertex in each commodity, that is, $v^* \in X$ and $S_i \cap \bar{X} \neq \emptyset$ for every $i = 1, \ldots, k$. The feasible cut problem asks to find a minimum cost feasible cut. We consider a special case of this problem, in which each commodity contains at most $d$ vertices.

### 1.2 Related Work

The multi-radius cover problem, which is a generalization of the vertex cover problem, was suggested by Hassin and Segev [7]. They presented two LP-based algorithms that

achieve an approximation ratio of 2. The first algorithm is based on an extension of threshold rounding, and the second is an efficient primal-dual algorithm that exploits the special structure of the problem.

Hassin and Megiddo [6] proved that the rectangle stabbing problem is NP-hard, and presented a 2-approximation algorithm for the special case in which all rectangles are translates of the same rectangle. Gaur, Ibaraki and Krishnamurti [4] developed a 2-approximation algorithm for the general case. Kovaleva and Spieksma [11] recently studied several variants of rectangle stabbing.

The group cut on a path problem was introduced by Hassin and Segev [8]. They showed that this problem is at least as hard to approximate as set cover, and presented a greedy $2H_k$-approximation algorithm. They also proved that group cut on a path is polynomial time solvable when the cardinality of each group is at most 3, but at least as hard to approximate as vertex cover when the bound on cardinality is 4.

The feasible cut problem was first studied by Yu and Cheriyan [12]. They proved that the problem is NP-hard, and provided an LP-rounding algorithm with an approximation ratio of 2 when each commodity contains two vertices. This special case was also considered by Bertsimas, Teo and Vohra [2], who presented a 2-approximation algorithm using dependent rounding. Hochbaum [10] proved that the linear program in [12] has an optimal solution consisting of half integrals, and obtained a faster 2-approximation algorithm.

### 1.3 Our Results

Our main result in Section 2 is a $k$-approximation algorithm for set cover instances in which the subsets covering each element form at most $k$ blocks in $S_{\mathcal{O}(1)}, \ldots, S_{\mathcal{O}(m)}$, where $\mathcal{O}$ is some known order. We remark that for every $k \geq 2$, Goldberg, Golumbic, Kaplan and Shamir [5] proved that the problem of recognizing whether such an order exists is NP-complete. Our result generalizes those of Bar-Yehuda and Even [1] and Hochbaum [9], whose algorithms guarantee a $k$-approximation when each block contains a single subset. Our algorithm identifies a collection of special blocks using the optimal fractional solution to the natural LP-relaxation of set cover. It then constructs a new integral program, in which the objective is to cover all elements, under the restriction that subsets must be chosen from all special blocks.

We proceed to show that this algorithm can be used to provide a 2-approximation for multi-radius cover. Using the indexing scheme suggested by Hassin and Segev [7], we present a new set cover formulation of this problem, and describe an order on the subsets, such that the subsets covering each element form at most two blocks. In addition, we show that through our set cover algorithm we can obtain new insight into the rectangle stabbing algorithm of Gaur, Ibaraki and Krishnamurti [4].

In Section 3 we consider a special case of GCP, in which the vertices of each group appear consecutively on the path, and show that this problem is at least as hard to approximate as vertex cover. We then present a 3-approximation algorithm, that is surprisingly based on an *incorrect* integer program. This program uses an objective function according to which we might pay the cost of each edge more than once. However, we prove that the cost of an optimal solution to this program is at most that of an optimal solution to the original problem. Moreover, we show how to obtain a 3-approximation

to the new program, and translate it to a set of edges that separates all groups without increasing the cost.

We also study another special case of GCP, where the cardinality of each group is at most $d$, for which we provide a $(d-2)$-approximation algorithm. This algorithm is based on an integer programming formulation with constraints that depend on the cardinality of each group[1]. We remark that since Hassin and Segev [8] proved that the case $d = 4$ is at least as hard as vertex cover, any approximation ratio better than $d-2$ would improve the best known results regarding vertex cover.

In Section 4 we present a $d$-approximation algorithm for a special case of the feasible cut problem, in which the cardinality of each commodity is at most $d$. This result improves previously known algorithms in two ways, as it is not restricted to commodities of exactly two vertices and is also very easy to analyze. Our algorithm uses the optimal fractional solution to an LP-relaxation that was originally suggested by Yu and Cheriyan [12], and identifies a non-empty subset of each commodity to be separated from $v^*$. Using these subsets, it then defines a new integral program, that can be interpreted as an extension of the well-known MINIMUM $s$-$t$ CUT problem.

## 2 Set Cover with *k* Blocks of Consecutive Ones

In this section we present a $k$-approximation algorithm for the special case of set cover in which there is a known order $\mathcal{O}$ on $\{1,\ldots,m\}$ such that the subsets covering each element $e \in U$ form at most $k$ blocks in $S_{\mathcal{O}(1)},\ldots,S_{\mathcal{O}(m)}$. We also show that this algorithm can be used to provide a 2-approximation for the multi-radius cover problem, and to obtain new insight into the rectangle stabbing algorithm of Gaur, Ibaraki and Krishnamurti [4]. To avoid confusion, we refer to the latter as the GIK algorithm.

### 2.1 The Algorithm

To simplify the presentation, we assume that the subsets $S_1,\ldots,S_m$ are indexed according to $\mathcal{O}$ in advance. In addition, for each element $e \in U$ we denote by $l_1^e \le r_1^e \le \cdots \le l_k^e \le r_k^e$ the endpoints of the blocks that cover $e$, that is, the collection of subsets that contain $e$ is $\bigcup_{t=1}^{k}\bigcup_{i=l_t^e}^{r_t^e}\{S_i\}$. Using this notation, the set cover problem can be formulated as an integer program by:

$$
\begin{array}{lllll}
(SC) & \min & \sum_{i=1}^{m} c_i x_i & & \\
 & \text{s.t.} & (1) \quad \sum_{t=1}^{k}\sum_{i=l_t^e}^{r_t^e} x_i \ge 1 & & \forall\, e \in U \\
 & & (2) \quad x_i \in \{0,1\} & & \forall\, i = 1,\ldots,m
\end{array}
$$

In this formulation, the variable $x_i$ indicates whether the subset $S_i$ is picked for the cover, and constraint $(1)$ ensures that for each element $e \in U$ we pick at least one

[1] Although our algorithm follows the general framework of rounding to an integral program, we defer its description to the full version of this paper.

subset that covers it. The LP-relaxation of this integer program, $(SC_f)$, is obtained by replacing the integrality constraint (2) with $x_i \geq 0$, as the upper bound on $x_i$ is redundant.

We first solve the linear program $(SC_f)$ to obtain an optimal fractional solution $x^*$. Next, we use this solution to identify for each element $e \in U$ a special set of blocks. Specifically, we apply threshold rounding to define for each $e \in U$ the set $I_e^* = \left\{ t : \sum_{i=l_t^e}^{r_t^e} x_i^* \geq \frac{1}{k} \right\}$. Based on these sets, we construct a new linear program

$$
\begin{array}{lllll}
(SC_f^*) & \min & \sum_{i=1}^{m} c_i x_i & & \\
& \text{s.t.} & (1) \quad \sum_{i=l_t^e}^{r_t^e} x_i \geq 1 & & \forall\, e \in U,\, t \in I_e^* \\
& & (2) \quad x_i \geq 0 & & \forall\, i = 1, \ldots, m
\end{array}
$$

and solve it to obtain an optimal solution $\hat{x}$.

In Lemma 1 we show that every feasible solution to $(SC_f^*)$ is also a feasible solution to $(SC_f)$. We also observe that $\hat{x}$ is an extreme point of an integral polyhedron, and therefore it is indeed a feasible solution to $(SC)$. In Theorem 2 we show that $x^*$ can be fitted into $(SC_f^*)$ when it is scaled by a factor of $k$. It follows that the cost of $\hat{x}$ is at most $k$ times the cost of $x^*$, which is a lower bound on the cost of any solution to the set cover problem.

**Lemma 1.** *$\hat{x}$ is a feasible solution to $(SC)$.*

*Proof.* We first show that every feasible solution to $(SC_f^*)$ is also a feasible solution to $(SC_f)$. Let $x'$ be a feasible solution to $(SC_f^*)$. As $x'$ is non-negative, it remains to prove $\sum_{t=1}^{k} \sum_{i=l_t^e}^{r_t^e} x_i' \geq 1$ for every $e \in U$. Consider some element $e$. Since $x^*$ is a feasible solution to $(SC_f)$, $\sum_{t=1}^{k} \sum_{i=l_t^e}^{r_t^e} x_i^* \geq 1$, and there is an index $s$ for which $\sum_{i=l_s^e}^{r_s^e} x_i^* \geq \frac{1}{k}$. It follows that $s \in I_e^*$ and the linear program $(SC_f^*)$ contains the constraint $\sum_{i=l_s^e}^{r_s^e} x_i \geq 1$. Therefore, $\sum_{t=1}^{k} \sum_{i=l_t^e}^{r_t^e} x_i' \geq \sum_{i=l_s^e}^{r_s^e} x_i' \geq 1$.

In addition, the rows of the coefficient matrix in $(SC_f^*)$ have the interval property, that is, each row contains a single interval of consecutive 1's. Such a matrix is totally unimodular, and the extreme points of the set of feasible solutions to $(SC_f^*)$ are integral. □

**Theorem 2.** *The cost of $\hat{x}$ is at most $k \cdot \mathrm{OPT}(SC_f)$.*

*Proof.* To bound the cost of $\hat{x}$, we claim that $kx^*$ is feasible for $(SC_f^*)$. Consider an element $e \in U$ and an index $s \in I_e^*$. Then $\sum_{i=l_s^e}^{r_s^e} kx_i^* = k \sum_{i=l_s^e}^{r_s^e} x_i^* \geq 1$, where the last inequality holds since $\sum_{i=l_s^e}^{r_s^e} x_i^* \geq \frac{1}{k}$. We conclude that

$$
\sum_{i=1}^{m} c_i \hat{x}_i \leq \sum_{i=1}^{m} c_i (k x_i^*) = k \sum_{i=1}^{m} c_i x_i^* = k \cdot \mathrm{OPT}(SC_f) \ .
$$

□

Two remarks are in place. First, the above analysis shows that an alternative way to construct the sets $I_e^*$ is to pick for each such set an arbitrary index $t$ for which $\sum_{i=l_t^e}^{r_t^e} x_i^* \geq \frac{1}{k}$. Second, our algorithm can be easily adapted to obtain a similar approximation ratio for the MULTICOVER problem. In this generalization of set cover, each element $e \in U$ has an integer coverage requirement $r_e$, and we are allowed to pick multiple copies of the subsets in $\mathcal{S}$.

### 2.2 Application: Multi-Radius Cover

We first formulate multi-radius cover as a set cover problem. Our formulation is based on the observation that when we transmit to radius $r_i^v$ from the station located at $v$, we cover all demand points on the edges adjacent to $v$ that are within distance of at most $r_i^v$ from $v$. Therefore, from each $u \in N(v)$ we must transmit to radius at least $l_{u,v} - r_i^v$. This enables us to hide the actual lengths of edges and radii by using the indexing scheme suggested in [7] as follows.

For every $v \in V$, $i = 0, \ldots, k_v$ and $u \in N(v)$, we define $\mathbb{I}_u^v(i) = \min\{j : r_j^u \geq l_{u,v} - r_i^v\}$ if there exists some $0 \leq j \leq k_u$ such that $r_j^u \geq l_{u,v} - r_i^v$, and $\mathbb{I}_u^v(i) = \infty$ otherwise. Note that if we transmit from $v$ to radius $r_i^v$, we must transmit from $u$ to radius at least $r^u_{\mathbb{I}_u^v(i)}$. In addition, when $\mathbb{I}_u^v(i) = \infty$ we cannot transmit from $v$ to radius $r_i^v$. Using this notation, multi-radius cover can be formulated as set cover by:

$$
\begin{array}{lllll}
(MRC) & \min & \displaystyle\sum_{v \in V} \sum_{i=0}^{k_v} b_{v,i} x_{v,i} & & \\
& \text{s.t.} & (1) \quad \displaystyle\sum_{i=0}^{k_v} x_{v,i} \geq 1 & & \forall\, v \in V \\
& & (2) \quad \displaystyle\sum_{j=\mathbb{I}_u^v(i)}^{k_u} x_{u,j} + \sum_{j=i+1}^{k_v} x_{v,j} \geq 1 & & \forall\, v \in V,\, i = 0, \ldots, k_v,\ u \in N(v) \\
& & (3) \quad x_{v,i} \in \{0,1\} & & \forall\, v \in V,\, i = 0, \ldots, k_v
\end{array}
$$

The variable $x_{v,i}$ indicates whether we transmit to radius $r_i^v$ from $v$. Constraint $(1)$ ensures that we choose at least one radius for each vertex. Constraint $(2)$ ensures that we either transmit from $v$ to radius at least $r_{i+1}^v$ or transmit from $u \in N(v)$ to radius at least $r^u_{\mathbb{I}_u^v(i)}$.

Each row of the coefficient matrix in $(MRC)$ contains at most two blocks according to the following order of the variables:

$$\mathcal{O} = x_{v_1,0}, x_{v_1,1}, \ldots, x_{v_1,k_{v_1}}, \ldots, x_{v_n,0}, x_{v_n,1}, \ldots, x_{v_n,k_{v_n}} \ .$$

Therefore, using the algorithm for set cover with blocks, we construct a feasible solution whose cost is at most $2 \cdot \mathrm{OPT}(MRC_f)$, where $(MRC_f)$ is the LP-relaxation of $(MRC)$.

### 2.3 Application: Rectangle Stabbing

Let $H = \{h_1, \ldots, h_{|H|}\}$ and $V = \{v_1, \ldots, v_{|V|}\}$ be the sets of horizontal and vertical lines, respectively. For each rectangle $r_k \in R$, we denote by $H_k$ and $V_k$ the subsets of

horizontal and vertical lines that intersect $r_k$. The rectangle stabbing problem can be formulated as an integer program by:

$$
\begin{array}{lllll}
(RS) & \min & \sum_{h_i \in H} w(h_i)x_i + \sum_{v_j \in V} w(v_j)y_j & & \\
& \text{s.t.} & (1) \quad \sum_{h_i \in H_k} x_i + \sum_{v_j \in V_k} y_j \geq 1 & & \forall\, k = 1, \ldots, n \\
& & (2) \quad x_i,\, y_j \in \{0, 1\} & & \forall\, i = 1, \ldots, |H|,\, j = 1, \ldots, |V|
\end{array}
$$

The variable $x_i$ indicates whether the horizontal line $h_i$ is chosen, and the variable $y_j$ indicates whether the vertical line $v_j$ is chosen. Constraint $(1)$ ensures that for each rectangle $r_k$ we choose at least one line that intersects it. We denote by $(RS_f)$ the LP-relaxation of $(RS)$, in which the integrality constraint $(2)$ is replaced with $x_i \geq 0$ and $y_j \geq 0$.

The GIK algorithm can be summarized as follows. We first solve the linear program $(RS_f)$ to obtain an optimal fractional solution $(x^*, y^*)$. Using this solution, we define two subsets of rectangles, $R_H = \{r_k \in R : \sum_{h_i \in H_k} x_i^* \geq \frac{1}{2}\}$ and $R_V = \{r_k \in R : \sum_{v_j \in V_k} y_j^* \geq \frac{1}{2}\}$. We now solve to optimality the problem of covering all rectangles in $R_H$ using horizontal lines and the problem of covering all rectangles in $R_V$ using vertical lines. These problems can be formulated as linear programs that have integral optimal solutions:

$$
\begin{array}{lllll}
(RS_H) & \min & \sum_{h_i \in H} w(h_i)x_i & & \\
& \text{s.t.} & (1) \quad \sum_{h_i \in H_k} x_i \geq 1 & & \forall\, r_k \in R_H \\
& & (2) \quad x_i \geq 0 & & \forall\, i = 1, \ldots, |H| \\
 & & & & \\
(RS_V) & \min & \sum_{v_j \in V} w(v_j)y_j & & \\
& \text{s.t.} & (1) \quad \sum_{v_j \in V_k} y_j \geq 1 & & \forall\, r_k \in R_V \\
& & (2) \quad y_j \geq 0 & & \forall\, j = 1, \ldots, |V|
\end{array}
$$

We show that the GIK algorithm is a special case of the set cover with blocks algorithm, that exploits additional structure of the problem. Each row of the coefficient matrix in $(RS)$ contains at most two blocks according to the order

$$\mathcal{O} = x_1, x_2, \ldots, x_{|H|}, y_1, y_2, \ldots, y_{|V|}$$

of the variables, where we assume that the lines in $H$ and $V$ are indexed in increasing order of coordinates. In addition, each block is contained either in $x_1, \ldots, x_{|H|}$ or in $y_1, \ldots, y_{|V|}$.

Given $(RS)$, the set cover algorithm uses the optimal fractional solution $(x^*, y^*)$ to identify for each rectangle $r_k$ at least one block from which a line will be subsequently chosen. The rule applied by the algorithm guarantees that a block $\{x_i : h_i \in H_k\}$ is chosen if and only if $r_k \in R_H$ in the GIK algorithm, and similarly, a block $\{y_j : v_j \in V_k\}$ is chosen if and only if $r_k \in R_V$. This observation implies that in the second stage the algorithm constructs the linear program

$$
\begin{array}{lll}
(RS_f^*) & \min & \sum_{h_i \in H} w(h_i)x_i + \sum_{v_j \in V} w(v_j)y_j \\
& \text{s.t.} & (1) \quad \sum_{h_i \in H_k} x_i \geq 1 \qquad \forall\, r_k \in R_H \\
& & (2) \quad \sum_{v_j \in V_k} y_j \geq 1 \qquad \forall\, r_k \in R_V \\
& & (3) \quad x_i,\, y_j \geq 0 \qquad \forall\, i = 1, \ldots, |H|,\, j = 1, \ldots, |V|
\end{array}
$$

and returns its optimal solution, which is integral. However, this linear program is separable with respect to $x$ and $y$. Moreover, $\mathrm{OPT}(RS_f^*) = \mathrm{OPT}(RS_H) + \mathrm{OPT}(RS_V)$, since $(RS_f^*)$ decomposes exactly to $(RS_H)$ and $(RS_V)$.

We remark that the set cover with blocks algorithm can be used to obtain a $d$-approximation for the problem of stabbing rectangles in $\mathbb{R}^d$ using hyperplanes. In addition, a $cd$-approximation can be obtained for the problem of stabbing compact sets with at most $c$ connected components in $\mathbb{R}^d$.

## 3 Group Cut with Consecutive Groups

In what follows we consider the case where the vertices of each group appear consecutively on the path. We assume that the left-to-right order of the vertices on $P$ is given by $v_1, \ldots, v_n$, and denote by $[v_i, v_j]$ the subpath connecting $v_i$ and $v_j$.

We first discuss the hardness of approximating GCP with consecutive groups, and prove that this problem is at least as hard to approximate as vertex cover. We then present a 3-approximation algorithm, that is surprisingly based on an incorrect integer program.

### 3.1 Hardness Results

In Lemma 3 we describe an approximation preserving reduction from the vertex cover problem to GCP with consecutive groups. It follows that hardness results regarding vertex cover extend to this special case of GCP, and in particular it is NP-hard to approximate the latter problem to within any factor smaller than 1.3606 [3].

**Lemma 3.** *A polynomial time $\alpha$-approximation algorithm for GCP with consecutive groups would imply a polynomial time $\alpha$-approximation algorithm for vertex cover.*

*Proof.* Given a vertex cover instance $I$, with a graph $G = (V, E)$ whose set of vertices is $V = \{u_1, \ldots, u_n\}$, we construct an instance $\rho(I)$ of GCP with consecutive groups as follows. For each vertex $u_i \in V$ there is a corresponding edge $e_i$ with unit cost, where the edges $e_1, \ldots, e_n$ are vertex-disjoint. We connect these edges to a path in increasing order of indices, using intermediate edges with cost $M = \alpha n + 1$. For each edge $(u_i, u_j) \in E$, $i < j$, we define the group $G_{ij}$ to be the sequence of vertices that begins at the left endpoint of $e_i$ and terminates at the right endpoint of $e_j$.

Let $S^* \subseteq V$ be a minimum cardinality vertex cover in $G$. We show how to find in polynomial time a vertex cover in $G$ with cardinality at most $\alpha|S^*|$, given a polynomial time $\alpha$-approximation algorithm for GCP with consecutive groups.

Since $S^*$ is a vertex cover, the set of edges $\{e_i : u_i \in S^*\}$ separates either the leftmost vertex or the rightmost vertex in each group $G_{ij}$ from the other vertices in

that group. It follows that $\mathrm{OPT}(\rho(I)) \leq |S^*|$, and we can find in polynomial time a set of edges $F$ that separates all groups such that $c(F) \leq \alpha|S^*|$. We claim that $S = \{u_i : e_i \in F\}$ is a vertex cover in $G$. Consider some edge $(u_i, u_j) \in E$. Clearly, $F$ cannot separate any vertex in the interior of $G_{ij}$ from the other vertices in $G_{ij}$, or otherwise it contains an edge with cost $M = \alpha n + 1$ and $c(F) > \alpha|S^*|$. Therefore, $F \cap \{e_i, e_j\} \neq \emptyset$ and $S$ contains at least one of the vertices $u_i$ and $u_j$. In addition, $|S| \leq \alpha|S^*|$ since $|S| = |F| = c(F)$. □

### 3.2 A 3-Approximation Algorithm

Let $L_i$ and $R_i$ be the indices of the leftmost and rightmost vertices of the group $G_i$, respectively. We formulate GCP with consecutive groups as an integer program using two types of variables. For $j = 1, \ldots, n-1$, the variable $x_j$ indicates whether we disconnect the edge $(v_j, v_{j+1})$. For $j = 2, \ldots, n-1$, the variable $y_j$ indicates whether we disconnect both $(v_{j-1}, v_j)$ and $(v_j, v_{j+1})$. Consider the following integer program:

$$
\begin{array}{lllll}
(GCP) & \min & \sum_{j=1}^{n-1} c_{j,j+1} x_j + \sum_{j=2}^{n-1} (c_{j-1,j} + c_{j,j+1}) y_j & & \\
& \text{s.t.} & (1) \quad x_{L_i} + x_{R_i - 1} + \sum_{j=L_i}^{R_i} y_j \geq 1 & & \forall\, i = 1, \ldots, k \\
& & (2) \quad x_j,\, y_j \in \{0, 1\} & & \forall\, j = 1, \ldots, n
\end{array}
$$

Clearly, constraint $(1)$ ensures that the collection of edges we should pick, according to the interpretation of the variables, separates all groups[2].

It appears as if we made a mistake by choosing the objective function in $(GCP)$. This follows from the observation that a single edge $(v_j, v_{j+1})$ can play three roles simultaneously: When $x_j = 1$, it separates $v_j$ as a leftmost vertex or $v_{j+1}$ as a rightmost vertex; when $y_j = 1$, along with $(v_{j-1}, v_j)$ it separates $v_j$ as a middle vertex; when $y_{j+1} = 1$, along with $(v_{j+1}, v_{j+2})$ it separates $v_{j+1}$ as a middle vertex. Therefore, by separately considering each group $G_i$ and adjusting the variables in $(GCP)$ according to their roles, we might end up paying the cost of each edge several times.

Let $F^*$ be a minimum cost set of edges that separates all groups. In Lemma 4 we resolve the problem described above, by suggesting a way to distribute the cost of $F^*$ between the variables in $(GCP)$, such that we obtain a feasible solution with an identical cost.

**Lemma 4.** *There is a feasible solution to $(GCP)$ whose cost is at most $c(F^*)$.*

An important observation is that $(GCP)$ is an integer programming formulation of a certain set cover problem, in which we are required to cover the groups $G_1, \ldots, G_k$ using single edges and pairs of adjacent edges. We now use the set cover with blocks algorithm, not before we notice that the subsets covering each group form at most three blocks according to the following order of the variables:

$$\mathcal{O} = x_1, x_2, \ldots, x_{n-1}, y_2, y_3, \ldots, y_{n-1} \ .$$

[2] For groups $G_i$ such that $L_i = 1$, we begin the summation $\sum_{j=L_i}^{R_i} y_j$ at $j = 2$. In addition, when $R_i = n$ we end this summation at $j = n - 1$.

By Theorem 2 and Lemma 4, we obtain a feasible solution $(\hat{x}, \hat{y})$ whose cost is at most $3c(F^*)$. This solution can be translated to a set of edges that separates all groups without increasing its cost, since now the fact that in $(GCP)$ we might pay the cost of each edge several times works in our favor.

**Theorem 5.** *There is a 3-approximation algorithm for GCP with consecutive groups.*

## 4 Feasible Cut

In this section we present a $d$-approximation algorithm for a special case of the feasible cut problem, in which the cardinality of each commodity is at most $d$. This algorithm is based on an LP-relaxation that was originally suggested by Yu and Cheriyan [12].

The feasible cut problem can be formulated as an integer program by:

$$
\begin{array}{lllll}
(FC) & \min & \sum_{(u,v)\in E} c_{u,v}x_{u,v} & & \\
& \text{s.t.} & (1) \quad y_{v^*} = 0 & & \\
& & (2) \quad \sum_{v\in S_i} y_v \geq 1 & & \forall\, i = 1,\ldots,k \\
& & (3) \quad x_{u,v} \geq y_u - y_v & & \\
& & \qquad\; x_{u,v} \geq y_v - y_u & & \forall\, (u,v) \in E \\
& & (4) \quad y_v,\, x_{u,v} \in \{0,1\} & & \forall\, v \in V,\, (u,v) \in E
\end{array}
$$

In this formulation, the variable $y_v$ indicates whether the vertex $v$ belongs to the side of the cut that does not contain $v^*$, and the variable $x_{u,v}$ indicates whether the edge $(u, v)$ crosses the cut. Constraint $(2)$ ensures that we separate from $v^*$ at least one vertex from each commodity. Constraint $(3)$ ensures that $x_{u,v} = 1$ when $(u, v)$ crosses the cut. Let $(FC_f)$ be the LP-relaxation of $(FC)$, in which constraint $(4)$ is replaced with $y_v \geq 0$ and $x_{u,v} \geq 0$.

Let $(x^*, y^*)$ be an optimal fractional solution to $(FC_f)$. We determine in advance a subset of vertices $V_i^* \subseteq S_i$ to be separated from $v^*$. These are vertices a large fraction of which is separated from $v^*$ in $y^*$, $V_i^* = \left\{v \in S_i : y_v^* \geq \frac{1}{d}\right\}$. We now construct a new linear program,

$$
\begin{array}{lllll}
(FC_f^*) & \min & \sum_{(u,v)\in E} c_{u,v}x_{u,v} & & \\
& \text{s.t.} & (1) \quad y_{v^*} = 0 & & \\
& & (2) \quad y_v \geq 1 & & \forall\, v \in \bigcup_{i=1}^{k} V_i^* \\
& & (3) \quad x_{u,v} \geq y_u - y_v & & \\
& & \qquad\; x_{u,v} \geq y_v - y_u & & \forall\, (u,v) \in E \\
& & (4) \quad y_v,\, x_{u,v} \geq 0 & & \forall\, v \in V,\, (u,v) \in E
\end{array}
$$

and solve it to obtain an optimal solution $(\hat{x}, \hat{y})$. Without loss of generality, we assume that $\hat{y}_v \leq 1$ for every $v \in V$, since this property can be achieved without increasing the cost of the solution.

Since $y_v \geq 1$ in constraint $(2)$ can be replaced by $y_v = 1$, $(FC_f^*)$ is the LP-relaxation of the problem of finding a minimum cut that separates $v^*$ and $\bigcup_{i=1}^{k} V_i^*$.

Therefore, $(\hat{x}, \hat{y})$ is integral. In Theorem 6 we show that this solution is indeed a feasible cut, with cost of at most $d$ times the cost of the optimal solution to $(FC_f)$.

**Theorem 6.** *$(\hat{x}, \hat{y})$ is a feasible solution to $(FC)$, and its cost is at most $d \cdot \mathrm{OPT}(FC_f)$.*

## References


1. R. Bar-Yehuda and S. Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.
2. D. Bertsimas, C. P. Teo, and R. Vohra. On dependent randomized rounding algorithms. *Operations Research Letters*, 24:105–114, 1999.
3. I. Dinur and S. Safra. The importance of being biased. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 33–42, 2002.
4. D. R. Gaur, T. Ibaraki, and R. Krishnamurti. Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem. *Journal of Algorithms*, 43:138–152, 2002.
5. P. W. Goldberg, M. C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *Journal of Computational Biology*, 2:139–152, 1995.
6. R. Hassin and N. Megiddo. Approximation algorithms for hitting objects with straight lines. *Discrete Applied Mathematics*, 30:29–42, 1991.
7. R. Hassin and D. Segev. The multi-radius cover problem, 2004.
8. R. Hassin and D. Segev. The set cover with pairs problem, 2005.
9. D. S. Hochbaum. Approximation algorithms for the set covering and vertex cover problems. *SIAM Journal on Computing*, 11:555–556, 1982.
10. D. S. Hochbaum. Instant recognition of half integrality and 2-approximations. In *Proceedings of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 99–110, 1998.
11. S. Kovaleva and F. C. R. Spieksma. Approximation of rectangle stabbing and interval stabbing problems. In *Proceedings of the 12th Annual European Symposium on Algorithms*, pages 426–435, 2004.
12. B. Yu and J. Cheriyan. Approximation algorithms for feasible cut and multicut problems. In *Proceedings of the 3rd Annual European Symposion on Algorithms*, pages 394–408, 1995.

# Rectangle Covers Revisited Computationally

L. Heinrich-Litan[1] and M.E. Lübbecke[2]

[1] Technische Universität Braunschweig,
Institut für Mathematische Optimierung,
Pockelsstraße 14, D-38106 Braunschweig, Germany
litan@tu-bs.de
[2] Technische Universität Berlin,
Institut für Mathematik, Sekr. MA 6-1, Straße des
17. Juni 136, D-10623 Berlin, Germany
m.luebbecke@math.tu-berlin.de

**Abstract.** We consider the problem of covering an orthogonal polygon with a minimum number of axis-parallel rectangles from a computational point of view. We propose an integer program which is the first general approach to obtain provably optimal solutions to this well-studied $\mathcal{NP}$-hard problem. It applies to common variants like covering only the corners or the boundary of the polygon, and also to the weighted case. In experiments it turns out that the linear programming relaxation is extremely tight, and rounding a fractional solution is an immediate high quality heuristic. We obtain excellent experimental results for polygons originating from VLSI design, fax data sheets, black and white images, and for random instances. Making use of the dual linear program, we propose a stronger lower bound on the optimum, namely the cardinality of a fractional stable set. We outline ideas how to make use of this bound in primal-dual based algorithms. We give partial results which make us believe that our proposals have a strong potential to settle the main open problem in the area: To find a constant factor approximation algorithm for the rectangle cover problem.

## 1 Introduction

A polygon with all edges either horizontal or vertical is called *orthogonal*. Given an orthogonal polygon $P$, the *rectangle cover problem* is to find a minimum number of possibly overlapping axis-parallel rectangles whose union is exactly $P$. In computational geometry, this problem received considerable attention in the past 25 years, in particular with respect to its complexity and approximability in a number of variants. Still, the intriguing main open question [5] is:

> Is there a constant factor approximation algorithm for the rectangle cover problem?

We do not answer this question now, but we offer a different and new kind of reply, which is "computationally, yes". In fact, we provide a fresh experimental

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 55–66, 2005.
© Springer-Verlag Berlin Heidelberg 2005

view, the first of its kind, on the problem which has applications in the fabrication of masks in the design of DNA chip arrays [11], in VLSI design, and in data compression, in particular in image compression.

*Previous work.* Customarily, one thinks of the polygon $P$ as a union of finitely many (combinatorial) pixels, sometimes also called a polyomino. The polygon $P$ can be associated with a visibility graph $G$ [15, 17, 18, 20]: The vertex set of $G$ is the set of pixels of $P$ and two vertices are adjacent in $G$ if and only if their associated pixels can be covered by a common rectangle. Rectangles correspond to cliques in $G$. That is a set of vertices, any two of which are adjacent. Let $\theta$ denote the number of rectangles in an optimal cover. An obvious lower bound on $\theta$ is the size $\alpha$ of a maximum stable set in $G$, also called maximum independent set. This is a set of pixels, no two of which are contained in a common rectangle. In the literature one also finds the notion of an antirectangle set.

Chvátal originally conjectured that $\alpha = \theta$, and this is true for convex polygons [6] and a number of special cases. Szemerédi gave an example with $\theta \neq \alpha$, see Figure 1. Intimately related to the initially stated open question, Erdős then asked whether $\theta/\alpha$ was bounded by a constant. In [6] an example is mentioned with $\theta/\alpha \geq 21/17 - \varepsilon$, however, this example cannot be reconstructed from [6], and thus cannot be verified. The best proven bound is $\theta/\alpha \geq 8/7$.

For polygons with holes and even for those without holes (also called simple) the rectangle cover problem is $\mathcal{NP}$-hard [16, 7] and Max$\mathcal{SNP}$-hard [4], that is, there is no polynomial time approximation scheme. The best approximation algorithms known achieve a factor of $O(\sqrt{\log n})$ for general polygons [1] and a factor of 2 for simple polygons [8], where $n$ is the number of edges of the polygon. Because of the problem's hardness quite some research efforts have gone into finding polynomially solvable special cases; we mention only covering with squares [2, 14] and polygons in general position [5]. Interestingly, there is a polynomial time algorithm for partitioning a polygon into non-overlapping rectangles [19]. However, a polygon similar to Fig. 3 shows that an optimal partition size may exceed an optimal cover size by more than constant factor, so this does not lead to an approximation.

*Our Contributions.* Despite its theoretical hardness, we demonstrate the rectangle cover problem to be computationally very tractable, in particular by studying an integer programming formulation of the problem. Doing this, we are the first to offer an exact (of course non-polynomial time) algorithm to obtain provably optimal solutions, and we are the first to introduce linear/integer programming techniques in this problem area. Based on a fractional solution to the (dual of the) linear programming relaxation we propose a stronger lower bound on the optimum cover size which we call the *fractional* stable set size. In fact, this new lower bound motivates us to pursue previously unexplored research directions to find a constant factor approximation algorithm. These are the celebrated primal-dual scheme [9], rounding a fractional solution, and a dual fitting algorithm [21]. We are optimistic that our research will actually contribute to a positive answer to the initially stated long standing open question, and due to space limitations

we only sketch some partial results and promising ideas. A fruitful contribution of our work is a number of open questions it spawns.

*Preliminaries.* Since we are dealing with a combinatorial problem, we identify $P$ with its set of combinatorial pixels. This way we write $p \in P$ to state that pixel $p$ is contained in polygon $P$. Let $R$ denote the set of all rectangles in $P$. It is important that we only count rectangles and do not consider areas. Thus, it is no loss of generality to restrict attention to inclusionwise maximal rectangles. We will do so in the following without further reference. The number of these rectangles can still be quadratic in the number $n$ of edges of $P$ [8], see also Fig. 2.

## 2 An Integer Program

Interpreting rectangles as cliques in $G$ we can make use of the standard integer programming formulation for the minimum clique cover problem in graphs [20]. A binary variable $x_r$ indicates whether rectangle $r \in R$ is chosen in the cover or not. For every pixel $p \in P$ at least one rectangle which covers $p$ has to be picked, and the number of picked rectangles has to be minimized:

$$\theta = \min \sum_{r \in R} x_r \tag{1}$$

$$\text{s. t.} \sum_{r \in R : r \ni p} x_r \geq 1 \qquad p \in P \tag{2}$$

$$x_r \in \{0,1\} \quad r \in R \tag{3}$$

This integer program (which we call the *primal* program) allows us to optimally solve any given instance of our problem, and we will do so in our experiments. When we replace (3) by $x_r \geq 0, r \in R$ (3′), we obtain the associated linear programming (LP) relaxation. There is no need to explicitly require $x_r \leq 1$, $r \in R$, since we are minimizing. We call the optimal objective function value of the LP relaxation the *fractional cover size* of $P$ and denote it by $\bar{\theta}$. Clearly, it holds that $\bar{\theta} \leq \theta$. In general, no polynomial time algorithm is known to compute the fractional clique cover number of a graph, that is, for solving this linear program [20]. In our case, however, the number of variables and constraints is polynomial in $n$, in fact quadratic, due to the fact that we work with maximal rectangles only. Therefore, the fractional cover size $\bar{\theta}$ can be computed in polynomial time.

This integer program immediately generalizes to the weighted rectangle cover problem, where rectangles need not have unit cost. It is straightforward, and it does not increase the complexity, to restrict the coverage requirement to particular features of the polygon like the corners or the boundary—two well-studied variants [4] for which no exact algorithm was known. It is also no coincidence that a formal dualization of our program leads to a formulation for the dual problem of finding a maximum stable set. A binary variable $y_p$, $p \in P$, reflects whether a pixel is chosen in the stable set or not. We have to require that no rectangle contains more than one of the chosen pixels, and we maximize the number of chosen pixels. We call this the dual integer program:

$$\alpha = \max \sum_{p \in P} y_p \tag{4}$$

$$\text{s. t.} \sum_{p \in P : p \in r} y_p \leq 1 \qquad r \in R \tag{5}$$

$$y_p \in \{0,1\} \quad p \in P \tag{6}$$

Again, when replacing (6) by $y_p \geq 0, p \in P$ (6′), we obtain the associated LP relaxation. We call its optimal objective function value $\bar{\alpha}$ the *fractional stable set size* of $P$. We refer to a feasible solution to the dual as a *fractional stable set*. It holds that $\bar{\alpha} \geq \alpha$. By strong linear programming duality we have $\bar{\alpha} = \bar{\theta}$. We stress again the fact that we distinguish between the (primal and dual) *integer* programs which solve the problems exactly, and their respective *continuous* linear programming relaxations, which give bounds. In general, optimal solutions to both linear programs (1)–(3′) and (4)–(6′) are fractional. However, using an interesting link to graph theory, in the case that $G$ is perfect [10], optimal solutions are automatically integer because of a strong duality between the *integer* programs [20]. This link was established already early, see e.g., [3,17,18], and our linear programs give optimal integer covers in polynomial time for this important class of polygons with $\alpha = \theta$.

### 2.1 About Fractional Solutions

Our computational experiments fuel our intuition; therefore we discuss some observations first. In linear programming based approximation algorithms the objective function value of a primal or dual fractional solution is used as a lower bound on the integer optimum. The more we learn about such fractional solutions the more tools we may have to analyze the problem's approximability.

*General Observations.* The linear relaxations (1)–(3′) and (4)–(6′) appear to be easily solvable to optimality in a few seconds on a standard PC. The vast majority of variables already assumes an integer value. A mere rounding of the remaining fractional variables typically gives an optimal or near-optimal integer solution (e.g., instance `night` is a bad example with "only" 95% integer values, but the rounded solution is optimal). For smaller random polygons the LP optimal solution is very often already integer; and this is an excellent quality practical heuristic, though memory expensive for very large instances.

*Odd Holes.* Figure 1 (left) shows Szemerédi's counterexample to the $\alpha = \theta$ conjecture. The 5 rectangles indicated by the shaded parts have to be in any cover. In the remaining parts of the polygon, there are 5 pixels which induce an odd-length cycle $C$ ("odd hole") in the visibility graph $G$. To cover these pixels, at least 3 rectangles are needed, implying $\theta \geq 8$. On the other hand, at most 2 of these pixels can be independent, that is, $\alpha \leq 7$. The odd hole $C$ is precisely the reason why $G$ is not perfect in this example. Figure 1 (right) shows that $C$ is encoded in the optimal fractional solution as well: Exactly the variables corresponding to edges of $C$ assume a value of 0.5. The same figure shows an

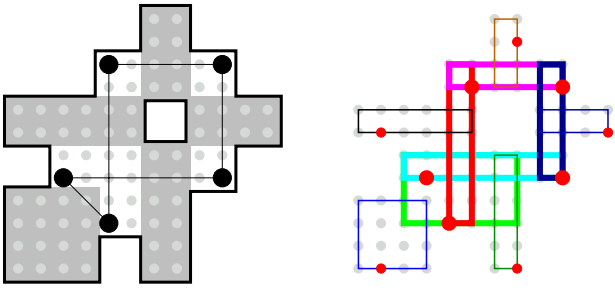


**Fig. 1.** The original counterexample to $\alpha = \theta$ by Szemerédi and (to the right) an optimal fractional cover. Thicker lines (points) indicate rectangles (pixels) which are picked to the extent of 0.5

optimal fractional stable set. Pixels corresponding to vertices of $C$ assume a value of 0.5 (drawn fatter in the figure). That is, $\bar{\alpha} = \bar{\theta} = 7.5$. This immediately suggests to strengthen the LP relaxation.

**Lemma 1.** *For any induced odd cycle $C$ with $|C| \geq 5$, the inequality $\sum_{r \in C} x_r \geq \lceil |C|/2 \rceil$ is valid for (1)–(3), where $r \in C$ denotes the rectangles corresponding to the edges of $C$.*

The graph theoretic complements of odd holes are called odd *antiholes*. A graph is not perfect either if it contains an induced odd antihole. However, we can prove that there is no way of representing even the simplest non-trivial antihole with 7 vertices in a rectangle visibility graph. Odd holes are therefore the only reason for imperfection. Unfortunately still, from our experiments, arbitrary fractions are possible, not only halves, and simply rounding a fractional solution does not give a constant factor approximation, as discussed next.

*High Coverage.* We define the coverage of a pixel $p$ as the number of rectangles which contain $p$. For the classical set cover problem, rounding up an optimal fractional solution gives an $f$-approximate cover, where $f$ is the maximum coverage of any element. In general, a pixel can have more than constant coverage; even worse, almost *no* pixel may have constant coverage; even in an optimal cover of a simple polygon in general position pixels may have high coverage (see Fig. 2). Unlike in the general set cover case, high coverage is no prediction about the fractions in an optimal LP solution: In Fig. 2 there are no fractional variables, the solution is integer. The fractional (indeed integer) optimal solution to this simple example has a remarkable property. Every rectangle in the optimal cover contains pixels of low coverage. More precisely, the following holds.

**Lemma 2.** *In an optimal cover $\mathcal{C}$, every rectangle $r \in \mathcal{C}$ contains a pixel which is uniquely covered by $r$.*

This can be easily seen since otherwise $\mathcal{C} \setminus \{r\}$ would be a cover, contradicting the optimality of $\mathcal{C}$. We call these uniquely covered pixels *private*. It is no coincidence that the pixels in a maximal stable set are private. It is natural to ask

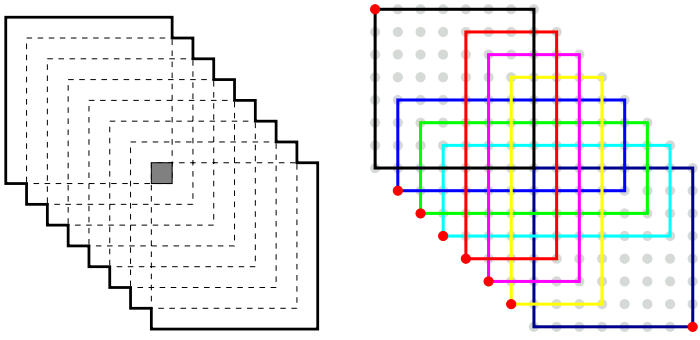


**Fig. 2.** Left: The shaded center pixel is covered by *any* maximal rectangle; almost all pixels have non-constant coverage. In an optimal cover, the coverage of the center pixel is linear in the cover size. The right figure schematically shows a minimal cover and a maximal stable set

(since an answer immediately turns LP rounding into a constant factor approximation algorithm): What are the characteristics of polygons where every pixel has only constant coverage? What kind of polygons have "many" pixels with "low" coverage? How can we exploit Lemma 2? These questions are intimately related to the next section.

## 3 LP Based Approximation

There are more elaborate linear programming based approaches to constant factor approximation algorithms. They can be used as analytical tools to theoretically sustain our excellent computational results.

### 3.1 Primal-Dual Scheme

The primal-dual scheme [9] builds on relaxing the well-known complementary slackness optimality conditions [20] in linear programming. The general scheme iteratively improves an initially infeasible integer primal solution, that is, a set of rectangles, to finally obtain a feasible cover. The improvement step is guided by a feasible fractional dual solution, that is a *fractional stable set*, which is improved in alternation with the primal solution. The relaxed complementary slackness conditions contain the key information. In our case they read

$$x_r > 0 \;\Rightarrow\; \frac{1}{d} \leq \sum_{p \in P : p \in r} y_p \qquad r \in R \tag{7}$$

for some constant $d$, and

$$y_p > 0 \;\Rightarrow\; \sum_{r \in R : r \ni p} x_r \leq c \qquad p \in P \tag{8}$$

for some constant $c$. First note that if a possibly infeasible primal integer solution is maintained, $x_r > 0$ means $x_r = 1$. An interpretation of condition (7) is that every rectangle in the constructed cover must cover at least $1/d$ pixels from the fractional stable set. Condition (8) states that a pixel in the fractional stable set must not be contained in more than $c$ rectangles (regardless of whether in the cover or not).

We found two cases where we can compute a cover and a fractional stable set simultaneously such that the two conditions hold. *Thin* polygons, as unions of width 1 or height 1 rectangles, are a class of polygons amenable to LP rounding and the primal-dual scheme: Since no pixel is covered by more than two rectangles this gives a 2-approximation. More generally, polygons of bounded width (every pixel contains a boundary pixel in its "neighborhood") are a new non-trivial class which allows a constant factor approximation.

### 3.2 Dual Fitting

Since $\alpha \leq \theta$ the former natural approach to approximation algorithms was to construct a large stable set usable as a good lower bound [8]. Since $\alpha \leq \bar{\alpha}$ we propose to use the stronger bound provided by a *fractional* stable set. Our *dual fitting* approach [21] is to simultaneously construct a cover $\mathcal{C} \subseteq R$ and an *pseudo stable set* $\mathcal{S} \subseteq P$ of pixels with $|\mathcal{C}| \leq |\mathcal{S}|$ (we say that $\mathcal{S}$ *pays* for $\mathcal{C}$). "Pseudo" refers to allowing a constant number $c$ of pixels in a rectangle, that is, we relax (5) to $\sum_{p \in P : p \in r} y_p \leq c$. From this constraint we see that picking each pixel in $\mathcal{S}$ to the extent of $1/c$ (which is a division of all $y_p$ variables' values by $c$) gives a feasible fractional solution to our dual linear program. A cover with these properties has a cost of

$$|\mathcal{C}| \leq |\mathcal{S}| \leq c \cdot \bar{\alpha} = c \cdot \bar{\theta} \leq c \cdot \theta \ , \tag{9}$$

that is, it would yield a $c$-approximation. Actually, one does not have to require that $\mathcal{S}$ pays for the full cover but $\frac{1}{d}|\mathcal{C}| \leq |\mathcal{S}|$ for a constant $d$ suffices, which would imply a $(c \cdot d)$-approximation. This paying for a constant fraction of the primal solution only is a new proposal in the context of dual fitting. Here again, the question is how to *guarantee* our conditions in general. From a computational point of view, we obtain encouraging results which suggest that our proposal can be developed into a proven constant factor approximation. In the next section we sketch some ideas how this can be done.

## 4 Towards a Constant Factor Approximation

### 4.1 Obligatory Rectangles and Greedy

For set cover, the greedy algorithm yields the best possible approximation factor of $O(\log n)$. The strategy is to iteratively pick a rectangle which covers the most yet uncovered pixels. One expects that for our particular problem, the performance guarantee can be improved. Computationally, we answer strictly

in the affirmative. Again, our contribution is the dual point of view. It is our aim to design an algorithm which is based on the dual fitting idea of Section 3.2, and we mainly have to say how to construct a feasible dual fractional solution.

We use some terminology from [11]. Certain rectangles have to be in any cover. A *prime* rectangle contains a pixel which is not contained in any other rectangle. Such a pixel is called a *leaf*. Every cover must contain all prime rectangles. For a given pixel $p$ we may extend horizontally and vertically until we hit the boundary; the rectangular area $R(p)$ defined by the corresponding edges at the boundary is called the *extended* rectangle of $p$. $R(p)$ might *not* be entirely contained in the polygon but if so, it is a prime rectangle [11]. Moreover, let $\mathcal{C}' \subseteq \mathcal{C}$ be a subset of some optimal cover $\mathcal{C}$. If there is a rectangle $r$ which contains $(P \setminus \mathcal{C}') \cap R(p)$ for some extended rectangle $R(p)$, then there is an optimal cover which contains $\mathcal{C}'$ and $r$ [11]. In this context, let us call rectangle $r$ *quasi-prime* and pixel $p$ a *quasi-leaf*. The algorithm we use to compute a cover is a slight extension of [11], but we will provide a new interpretation, and more importantly, a dual counterpart:

Quasi-Greedy

1. pick all prime rectangles
2. pick a maximal set of quasi-prime rectangles
3. cover the remaining pixels with the greedy algorithm
4. remove redundant rectangles ("pruning")

It has not been observed before that a set of leafs and quasi-leafs forms a stable set. This leads to the idea to compute a pseudo stable set containing a maximal set of leafs and quasi-leafs. Thus, in order to build a pseudo stable set we check for every rectangle in the greedy cover whether it contains

1. a leaf
2. a quasi-leaf
3. a corner pixel

The first positive test gives a pixel which we add to the pseudo stable set. A *corner pixel* is a corner of a rectangle which is private and a corner of the polygon. We already observed that pixels from steps 1 and 2 are independent. Furthermore, any rectangle obviously contains at most 4 corner pixels, and since corner pixels are private, actually at most 2 of them. By our previous considerations, this would imply a 2-approximation if the constructed pseudo stable set would pay for the whole cover. In general, we found this not to be true. We have constructed examples which suggest that one cannot guarantee that a constant fraction of the cover has been paid for. To achieve this latter goal one has to add more pixels to the pseudo stable set. To this end we extend the above test and also check for every rectangle in the cover whether it contains

4. a border pixel.

A *border pixel* $p$ is private and adjacent to a non-polygon pixel $\bar{p}$ (the outer face or a hole). The row (or column) of pixels which contains $p$, which is adjacent

to $\bar{p}$, and which extends to the left and the right (to the top and the bottom) until some non-polygon pixel is hit *must not be adjacent* to a different hole (or the outer face) *other* than the hole (or the outer face) the pixel $\bar{p}$ corresponds to. Also these pixels have a natural motivation.

Let us furthermore remark that after the pruning step in Quasi-Greedy, every rectangle in the cover contains a private pixel (Lem. 2). This pixel is an intuitive candidate to become a pixel in a pseudo stable set. This set would actually pay for the whole cover. However, it is not clear whether one can control how many pixels of this set can be contained in the same rectangle.

### 4.2 Using Boundary Covers

There is a simple 4-approximation algorithm for covering the boundary of an orthogonal polygon [4]. In this context a natural question arises: Can we always find an interior cover whose size is bounded from above by a constant multiple of the size $\theta_{\text{boundary}}$ of an optimal boundary cover? The answer is "no". Our counterexample in Fig. 3 shows that there is an $O(\sqrt{n})$-cover of the boundary of the polygon in the left figure with maximal horizontal and vertical strips. But the optimal interior cover needs $\Theta(n)$ rectangles since the white uncovered pixels in the right figure are independent. Nevertheless, the latter observation is actually very encouraging. We conjecture that one can find an interior cover of size less than $c_1 \cdot \theta_{\text{boundary}} + c_2 \cdot \alpha$ where $c_1$ are $c_2$ are appropriate constants. This would imply a constant factor approximation for the rectangle cover problem.

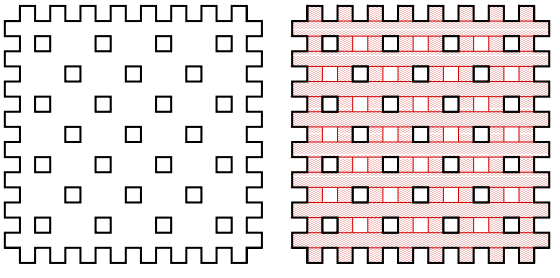


**Fig. 3.** A boundary cover may leave a non-constant fraction of pixels uncovered

### 4.3 Quasi-Prime Rectangles and Breaking Holes

There is a large class of polygons (e.g., polygons resulting from typical oligonucleotide masks [11]) where the optimal cover is found after the first two steps of the Quasi-Greedy algorithm in Section 4.1. Then the cover consists of only prime and quasi-prime rectangles. This is of course in general not the case (see Fig. 1). Now, consider the set $\mathcal{U}$ of pixels remained uncovered after step 2. We can prove that there is an induced cycle (a hole) in $G$ whose vertices correspond to a subset of $\mathcal{U}$. Covering each second edge of this hole extends the previous partial cover. We call this covering step to "break a hole". A straightforward

algorithm is the following: while the polygon is uncovered, iteratively pick a maximal set of quasi-prime rectangles, then find a hole and break it. We can iteratively extend also the partial pseudo stable set. The quasi-prime rectangles are paid for by quasi-leafs, which form a stable set. The rectangles which break an even (odd) hole can all (but one) be paid for by a stable set, too.

We have experimented with related and extended ideas based on the observations sketched in Sections 4.2 and 4.3 and obtained encouraging results. These methods and their approximation potential are currently under investigation.

## 5 Computational Experience

We experimented with small polygons occurring in VLSI mask design (instances `VLSI*`), a set of standard fax images[1] (instances `ccitt*`), and several black and white images (instances `marbles`, `mickey`, ...). Further, we have two strategies to construct random polygons. The first is to eliminate a varying fraction of single pixels uniformly from a square of size up to $750 \times 750$ pixels. The second is a union of uniformly placed rectangles of random sizes.

**Table 1.** Results for the primal and dual linear/integer programs. For each instance we list its size in pixels, its number of pixels (as a fraction), and its number of rectangles. For the dual and the primal programs (in that order) we give the optimal linear and integer program objective function values. The 'LP gap' is the relative gap between linear and integer program. Notice that instances `mickey` and `night` do not have a fractional optimal solution with 'nice' fractions

| | instance characteristics | | | dual (stable set size) | | | primal (cover size) | | |
|---|---|---|---|---|---|---|---|---|---|
| Instance | size | density | rectangles | opt. LP | opt. IP | **LP gap** | opt. LP | opt. IP | **LP gap** |
| `VLSI1` | 68×35 | 50.25% | 45 | 43.000 | 43 | 0.000% | 43.000 | 43 | 0.000% |
| `VLSI2` | 3841×298 | 95.34% | 16694 | 4222.667 | 4221 | 0.039% | 4222.667 | 4224 | 0.032% |
| `VLSI3` | 148×135 | 45.09% | 78 | 71.000 | 71 | 0.000% | 71.000 | 71 | 0.000% |
| `VLSI5` | 6836×1104 | 55.17% | 192358 | 77231.167 | 77227 | 0.005% | 77231.167 | 77234 | 0.004% |
| `ccitt1` | 2376×1728 | 3.79% | 27389 | 14377.000 | 14377 | 0.000% | 14377.000 | 14377 | 0.000% |
| `ccitt2` | 2376×1728 | 4.49% | 30427 | 7422.000 | 7422 | 0.000% | 7422.000 | 7422 | 0.000% |
| `ccitt3` | 2376×1728 | 8.21% | 40625 | 21085.000 | 21085 | 0.000% | 21085.000 | 21086 | 0.005% |
| `ccitt4` | 2376×1728 | 12.41% | 101930 | 56901.000 | 56901 | 0.000% | 56901.000 | 56901 | 0.000% |
| `ccitt5` | 2376×1728 | 7.74% | 46773 | 24738.500 | 24738 | 0.002% | 24738.500 | 24739 | 0.002% |
| `ccitt6` | 2376×1728 | 5.04% | 30639 | 12013.000 | 12013 | 0.000% | 12013.000 | 12014 | 0.008% |
| `ccitt7` | 2376×1728 | 8.69% | 85569 | 52502.500 | 52502 | 0.001% | 52502.500 | 52508 | 0.010% |
| `ccitt8` | 2376×1728 | 43.02% | 41492 | 14024.500 | 14022 | 0.018% | 14024.500 | 14025 | 0.004% |
| `marbles` | 1152×813 | 63.49% | 56354 | 44235.000 | 44235 | 0.000% | 44235.000 | 44235 | 0.000% |
| `mickey` | 334×280 | 75.13% | 17530 | 9129.345 | 9127 | 0.026% | 9129.345 | 9132 | 0.029% |
| `day` | 480×640 | 64.63% | 45553 | 32191.000 | 32190 | 0.000% | 32191.000 | 32192 | 0.003% |
| `night` | 480×640 | 96.02% | 17648 | 7940.985 | 7938 | 0.038% | 7940.985 | 7943 | 0.025% |

The extremely small integrality gaps listed in Tab. 1 and experienced for thousands of random polygons (not listed here) are a strong vote for our integer programming approach. On the downside of it, integer programs for industrial

[1] Available at `http://www.cs.waikato.ac.nz/~singlis/ccitt.html`

**Table 2.** Details for the Quasi-Greedy algorithm of Section 4.1. We compare the optimal cover size against ours (they differ by only 3–7%). The following columns list the number of prime and quasi-prime rectangles, and those picked by the greedy step. Then, the number of corner and border pixels in the constructed quasi stable set $\mathcal{S}$ is given (the number of (quasi-)leafs equals the number of (quasi-)primes). Finally, we state the maximal number of pixels of $\mathcal{S}$ in some rectangle, and the fraction of the cover size for which $\mathcal{S}$ pays

| Instance | optimum cover size | prime | quasi-prime | greedy | corner | border | max pixels | pays for |
|---|---|---|---|---|---|---|---|---|
| VLSI1 | 43 | 43 | 41 | 2 | 0 | 0 | 1 | 100.00% |
| VLSI2 | 4224 | 4701 | 1587 | 203 | 2911 | 1105 | 1279 | 4 | 88.79% |
| VLSI3 | 71 | 71 | 71 | 0 | 0 | 0 | 0 | 1 | 100.00% |
| ccitt1 | 14377 | 14457 | 10685 | 2099 | 1673 | 1632 | 28 | 2 | 99.91% |
| ccitt2 | 7422 | 7617 | 3587 | 409 | 3621 | 3574 | 29 | 3 | 99.76% |
| ccitt3 | 21086 | 21259 | 15691 | 2020 | 3548 | 3427 | 86 | 3 | 99.84% |
| ccitt4 | 56901 | 57262 | 42358 | 8605 | 6299 | 6110 | 59 | 2 | 99.77% |
| ccitt5 | 24739 | 24911 | 18529 | 2985 | 3397 | 3259 | 98 | 2 | 99.84% |
| ccitt6 | 12014 | 12132 | 8256 | 1049 | 2827 | 2764 | 35 | 2 | 99.77% |
| ccitt7 | 52508 | 52599 | 39230 | 10842 | 2525 | 2448 | 56 | 2 | 99.96% |
| ccitt8 | 14025 | 14303 | 7840 | 1353 | 5110 | 5023 | 54 | 3 | 99.77% |
| marbles | 56354 | 44235 | 43548 | 687 | 0 | 0 | 0 | 1 | 100.00% |
| mickey | 9132 | 9523 | 5582 | 690 | 3251 | 528 | 1593 | 3 | 88.13% |
| day | 32192 | 32431 | 26308 | 3777 | 2346 | 749 | 900 | 4 | 97.85% |
| night | 7943 | 8384 | 4014 | 501 | 3869 | 762 | 1810 | 4 | 84.53% |

size polygons, e.g., from VLSI design are extremely large. The generation of the integer programs consumes much more time than solving them which takes typically only a few seconds using the standard solver CPLEX [13]. As a remedy we propose a column generation approach, that is, a dynamic generation of the variables of the linear program. This enables us to attack larger instances.

For random instances the relation between the different objective function values is very similar to Tab. 1 and is not reported separately in this abstract. The excellent performance of the Quasi-Greedy algorithm can be seen in Tab. 2. We remark that we never observed more than 4 pixels of a pseudo stable set in a rectangle, and the pseudo stable set pays for significantly more than 50% of the cover size. This supports that Quasi-Greedy could be an 8-approximation algorithm for the rectangle cover problem (see Section 4.1).

## 6 Conclusions

It is common that theory is complemented by computational experience. In this paper we did the reverse: We found promising research directions by a careful study of computational experiments. Finally, we propose:

*Restatement of Erdős' Question.* Is it true that *both*, the integrality gap of our primal and that of our dual integer program are bounded by a constant? The example in Fig. 1 places lower bounds on these gaps of $\theta/\bar{\theta} \geq 16/15$ and $\bar{\alpha}/\alpha \geq 15/14$, implying the already known bound $\theta/\alpha \geq 8/7$. We conjecture that these gaps are in fact tight. Originally, we set out to find an answer to Erdős' question. We conclude with an answer in the affirmative, at least computationally.

## Acknowledgments

We thank Sándor Fekete for fruitful discussions and Ulrich Brenner for providing us with polygon data from the mask fabrication process in VLSI design.

## References

1. V.S. Anil Kumar and H. Ramesh. Covering rectilinear polygons with axis-parallel rectangles. *SIAM J. Comput.*, 32(6):1509–1541, 2003.
2. L.J. Aupperle, H.E. Conn, J.M. Keil, and J. O'Rourke. Covering orthogonal polygons with squares. In *Proc. 26th Allerton Conf. Commun. Control Comput.*, pages 97–106, 1988.
3. C. Berge, C.C. Chen, V. Chvátal, and C.S. Seow. Combinatorial properties of polyominoes. *Combinatorica*, 1:217–224, 1981.
4. P. Berman and B. DasGupta. Complexities of efficient solutions of rectilinear polygon cover problems. *Algorithmica*, 17(4):331–356, 1997.
5. M. Bern and D. Eppstein. Approximation algorithms for geometric problems. In Hochbaum [12], chapter 8, pages 296–345.
6. S. Chaiken, D.J. Kleitman, M. Saks, and J. Shearer. Covering regions by rectangles. *SIAM J. Algebraic Discrete Methods*, 2:394–410, 1981.
7. J.C. Culberson and R.A. Reckhow. Covering polygons is hard. *J. Algorithms*, 17:2–44, 1994.
8. D.S. Franzblau. Performance guarantees on a sweep-line heuristic for covering rectilinear polygons with rectangles. *SIAM J. Discrete Math.*, 2(3):307–321, 1989.
9. M.X. Goemans and D.P. Williamson. The primal-dual method for approximation algorithms and it application to network design problems. In Hochbaum [12], chapter 4.
10. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
11. S. Hannenhalli, E. Hubell, R. Lipshutz, and P.A. Pevzner. Combinatorial algorithms for design of DNA arrays. *Adv. Biochem. Eng. Biotechnol.*, 77:1–19, 2002.
12. D.S. Hochbaum, editor. *Approximation Algorithms for $\mathcal{NP}$-Hard Problems*. PWS Publishing Co., Boston, MA, 1996.
13. ILOG Inc., CPLEX Division. *CPLEX 9.0 User's Manual*, 2004.
14. C. Levcopoulos and J. Gudmundsson. Approximation algorithms for covering polygons with squares and similar problems. In *Proceedings of RANDOM'97*, volume 1269 of *Lect. Notes Comput. Sci.*, pages 27–41, Berlin, 1997. Springer.
15. F. Maire. Polyominos and perfect graphs. *Inform. Process. Lett.*, 50(2):57–61, 1994.
16. W.J. Masek. Some NP-complete set covering problems. Unpublished manuscript, MIT, 1979.
17. R. Motwani, A. Raghunathan, and H. Saran. Covering orthogonal polygons with star polygons: The perfect graph approach. *J. Comput. System Sci.*, 40:19–48, 1989.
18. R. Motwani, A. Raghunathan, and H. Saran. Perfect graphs and orthogonally convex covers. *SIAM J. Discrete Math.*, 2:371–392, 1989.
19. T. Ohtsuki. Minimum dissection of rectilinear regions. In *Proc. 1982 IEEE Symp. on Circuits and Systems, Rome*, pages 1210–1213, 1982.
20. A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin, 2003.
21. V.V. Vazirani. *Approximation Algorithms*. Springer, Berlin, 2001.

# Don't Compare Averages

Holger Bast and Ingmar Weber

Max-Planck-Institut für Informatik,
Saarbrücken, Germany
{bast, iweber}@mpi-sb.mpg.de

**Abstract.** We point out that for two sets of measurements, it can happen that the average of one set is larger than the average of the other set on one scale, but becomes smaller after a non-linear monotone transformation of the individual measurements. We show that the inclusion of error bars is no safeguard against this phenomenon. We give a theorem, however, that limits the amount of "reversal" that can occur; as a by-product we get two non-standard one-sided tail estimates for arbitrary random variables which may be of independent interest. Our findings suggest that in the not infrequent situation where more than one cost measure makes sense, there is no alternative other than to explicitly compare averages for each of them, much unlike what is common practice.

## 1 Introduction

Fig. 1 shows us a typical performance statistic as we find it in many papers.

For the sake of concreteness, let us assume that the two graphs pertain to two different numerical algorithms and that it was measured how large the numbers get in the internal computations. More precisely, the number of bits needed to represent the largest integer were measured, and each point in the graph is actually an average taken over a number of problem instances. The fewer bits,

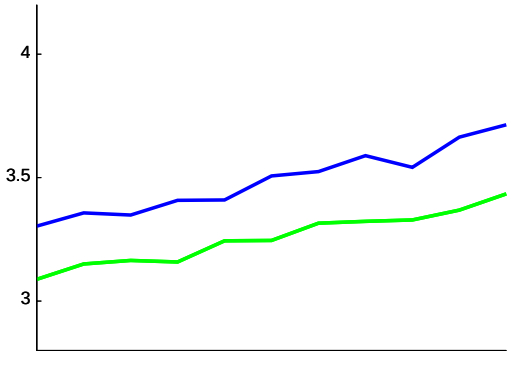



**Fig. 1.** The light gray algorithm is clearly better than the dark gray one . . .

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 67–76, 2005.
© Springer-Verlag Berlin Heidelberg 2005

the better of course. Along the $x$-axis the input size is varied. The message conveyed by the figure is clear: the "light gray" algorithm performs consistently, that is for all considered problem sizes, about 10% better than the "dark gray" algorithm.

Now the cost measure is somewhat arbitrary in the sense that we might as well have chosen to record the largest integer used and not the number of bits used to represent it, that is, to consider costs $2^c$ instead of costs $c$. What graph do we expect then? Well, if on some instance the one algorithm needs 3 bits and the other 4 bits, the modified costs would be $2^3 = 8$ versus $2^4 = 16$, that is, not surprisingly the gap between the two becomes larger. Now let us take a look at the graph for the same data but with the modified cost measure.

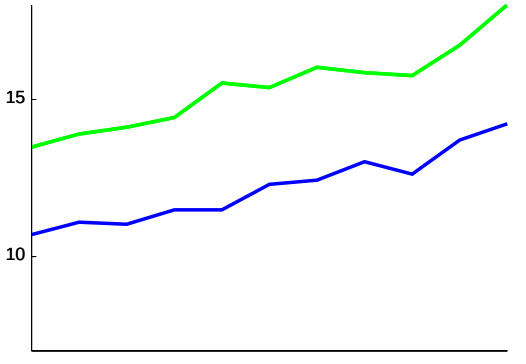



**Fig. 2.** . . . or isn't it?

Indeed, the gap has increased (from 10% to about 30%), but moreover, *the order of the two graphs has changed!* How is that possible?

There is, of course, nothing wrong with the figures, which are from authentic data; details are given in Appendix A. The reason for the reversal is that for two random variables $X$ and $Y$, $\mathbf{E}X \leq \mathbf{E}Y$ does *not*, in general, imply that for an (even strictly) increasing function $f$, $\mathbf{E}f(X) \leq \mathbf{E}f(Y)$. For a simple counterexample, consider two runs of our two algorithms above, where the first algorithm required once 1 and once 5 bits, and the second algorithm required 4 bits twice. Then clearly, on average the first algorithm required *one bit less*. Considering the second cost measure, the first algorithm on average required numbers up to $(2^1 + 2^5)/2 = 17$, which is *one more* than the $(2^4 + 2^4)/2 = 16$ required by the second algorithm.

Alternative cost measures are actually quite frequent: to assess the quality of a language model, for example, both *cross-entropy* ($c$) and *perplexity* ($2^c$) are equally meaningful and both used frequently [1]. Example publications with comparison graphs (or tables) of the very same kind as in Fig. 1 and 2 are [2] [3] [4] [1]. To give a concrete numerical example also, one of these papers in one of their graphs states average perplexities of $\approx 3200$ and $\approx 2900$ for two competing

methods. This appears to indicate a solid 10%-improvement of the one method over the other, but first, note that the difference of the logarithms is a mere 1%, and second, these perplexities would also result if, for example, the cross-entropies were normally distributed with a mean and standard deviation of 11.4 and 0.6, respectively, for the apparently superior method, and 11.3 and 1.0 for the apparently inferior method; see Appendix A for the calculations concerning the normal distribution.

But language modeling is just one prominent example. Another frequent scenario is that one (basic) algorithm is used as a subroutine in another (more complex) algorithm in such a way that the complexity of the latter depends on the complexity of the former via a non-linear, for example quadratic, function $f$. Then, of course, an average complexity of $c$ of the basic algorithm does not simply translate to an average complexity of $f(c)$ of the more complex one. But isn't it very tempting to assume that a subroutine with an improved average complexity will at least improve the program that uses it? Well, but that is just not necessarily true.

Now it is (or at least should be) common practice when plotting averages to also provide so-called *error bars*, indicating some average deviation from the average. The following theorem, which is the main result of this paper, says that the "bands" formed by such error bars at least cannot be reversed completely, that is, without intersection, by any monotone transformation $f$. As is also stated in the theorem, however, the obvious strengthenings of this statement do *not* hold: for example, it can very well happen that the bands do not intersect in one measure, yet the means reverse in another measure. The theorem is stated in terms of expected *absolute deviations* $\delta X = \mathbf{E}|X - \mathbf{E}X|$ and $\delta Y = \mathbf{E}|Y - \mathbf{E}Y|$, which are never more than the standard deviation; see Fact 1 further down.

**Theorem 1.** *For any two random variables $X$ and $Y$, and for any function $f$ that is strictly increasing, we have*

$$\mathbf{E}X + \delta X \le \mathbf{E}Y - \delta Y \implies \mathbf{E}f(X) - \delta f(X) \le \mathbf{E}f(Y) + \delta f(Y) \ .$$

*This result cannot be strengthened in the sense that if we drop any one of $\delta X$, $\delta Y$, $\delta f(X)$, or $\delta f(Y)$ to obtain weaker conditions, we can find a counter-example to the statement.*

The proof for Theorem 1, which we give in the following Sect. 2, is elementary but not obvious. Indeed, on their way the authors switched several times between striving for a proof, and being close to finding a counterexample. In Sect. 3, we give an alternative, more elegant proof in terms of the median. The first proof is more direct, however, while the second proof owes its elegance and brevity to the insight gained from the first; this is why we give the two proofs in that order.

To establish Theorem 1, we will derive two non-standard one-sided tail estimates for general random variables, namely for $a > 0$,

$$\mathbf{Pr}(X \ge \mathbf{E}X + a) \le \delta X/(2a);$$
$$\mathbf{Pr}(X \le \mathbf{E}X - a) \le \delta X/(2a) \ .$$

These bounds, which are reminiscent of but incomparable to the one-sided version of the Chebyshev inequality (cf. Appendix B), seem to be little known and may be of independent interest.

## 2 Proof of the Main Theorem

All the proofs we give in this paper are for continuous random variables. In all cases it will be obvious how to modify the proofs to work for the discrete case by replacing integrals by sums. For a random variable $X$, we write $\mathbf{E}X$ for its expected value (mean), $\sigma X$ for its standard deviation, that is $\sqrt{\mathbf{E}\left(|X - \mathbf{E}X|^2\right)}$, and $\delta X$ for the mean absolute deviation $\mathbf{E}|X - \mathbf{E}X|$. We will throughout assume that these entities exist. The following simple fact relates the two deviation measures.

**Fact 1.** *For every random variable $X$, it holds that $\delta X \leq \sigma X$.*

*Proof.* By Jensen's inequality, $(\delta X)^2 = \left(\mathbf{E}|X - \mathbf{E}X|\right)^2 \leq \mathbf{E}\left(|X - \mathbf{E}X|^2\right) = (\sigma X)^2$. □

Generally, this inequality will be strict. To get a feeling for the difference, check that for a normal distribution $N(\mu, \sigma)$ we have $\delta = \sqrt{2/\pi}\,\sigma \approx 0.8\,\sigma$ and for an exponential distribution $\mathrm{Exp}(\lambda)$ we have $\delta = 2/e\,\sigma = 2/(e\,\lambda) \approx 0.7\,\sigma$.

As a consequence of Fact 1 all our results still hold if we replace $\delta$ by $\sigma$, that is, we will be proving the stronger form of all results.

We first prove the following non-standard tail estimates, which might be of independent interest. There is a one-sided version of Chebyshev's inequality [5] which looks similar to Lemma 1 below, but the two are incomparable: Lemma 1 is stronger for deviations up to at least $\sigma X$, while the Chebyshev tail bounds are stronger for large deviations; see Appendix B.

**Lemma 1.** *For any random variable $X$ and for every $a > 0$, it holds that*

*(a)* $\mathbf{Pr}(X \geq \mathbf{E}X + a) \leq \delta X/(2a)$ ;
*(b)* $\mathbf{Pr}(X \leq \mathbf{E}X - a) \leq \delta X/(2a)$ .

*Proof.* Since $\delta X$ is invariant under shifting $X$ by a constant, we may assume without loss of generality that $\mathbf{E}X = 0$.

Then, with $\varphi$ denoting the density function pertaining to $X$,

$$0 = \mathbf{E}X = \int_{-\infty}^{0} t \cdot \varphi(t)\,dt + \int_{0}^{\infty} t \cdot \varphi(t)\,dt$$

$$\delta X = \int_{-\infty}^{0} (-t) \cdot \varphi(t)\,dt + \int_{0}^{\infty} t \cdot \varphi(t)\,dt \ .$$

Adding the two equations gives us

$$\delta X = 2 \cdot \int_{0}^{\infty} t \cdot \varphi(t)\,dt$$

$$\begin{aligned} &\geq 2 \cdot \int_a^\infty t \cdot \varphi(t)\, dt \\ &\geq 2a \cdot \int_a^\infty \varphi(t)\, dt \\ &= 2a \cdot \mathbf{Pr}(X \geq a) \ , \end{aligned}$$

and hence $\mathbf{Pr}(X \geq a) \leq \delta X/(2a)$, which establishes (a). The proof for (b) is analogous. □

Armed with Lemma 1 we can now establish a relation between $f(\mathbf{E}X)$ and $\mathbf{E}f(X)$ for a monotone function $f$.

**Lemma 2.** *For any random variable $X$, and for any function $f$ that is strictly increasing, it holds that*

*(a)* $\mathbf{E}f(X) - \delta f(X) \leq f(\mathbf{E}X + \delta X)$ ;
*(b)* $\mathbf{E}f(X) + \delta f(X) \geq f(\mathbf{E}X - \delta X)$ .

*Proof.* Let $a = \mathbf{E}f(X) - f(\mathbf{E}X + \delta X)$. If $a \leq 0$, there is nothing to show for (a), otherwise two applications of the previous Lemma 1 give us

$$\begin{aligned} 1/2 &\leq \mathbf{Pr}(X \leq \mathbf{E}X + \delta X) \\ &= \mathbf{Pr}(f(X) \leq f(\mathbf{E}X + \delta X)) \\ &= \mathbf{Pr}(f(X) \leq \mathbf{E}f(X) - a) \\ &\leq \delta f(X)/(2a) \ , \end{aligned}$$

and hence $\mathbf{E}f(X) - f(\mathbf{E}X + \delta X) = a \leq \delta f(X)$, which is exactly part (a) of the lemma. The proof of part (b) is analogous. More generally, we could in fact get that for any $t$,

$$f(t) - \frac{\delta f(X)}{2\,\mathbf{Pr}(X \geq t)} \ \leq \ \mathbf{E}f(X) \ \leq \ f(t) + \frac{\delta f(X)}{2\,\mathbf{Pr}(X \leq t)} \ . \qquad \square$$

Theorem 1 is now only two application of Lemma 2 away. Let $\mathbf{E}X + \delta X \leq \mathbf{E}Y - \delta Y$, like in the theorem, that is, the "bands" formed by the error bars do not intersect. Then

$$\begin{aligned} \mathbf{E}f(X) - \delta f(X) &\leq f(\mathbf{E}X + \delta X) \\ &\leq f(\mathbf{E}Y - \delta Y) \\ &\leq \mathbf{E}f(Y) + \delta f(Y) \ , \end{aligned}$$

where the first inequality is by part (a) of Lemma 2, the second inequality follows from the monotonicity of $f$, and the third inequality is by part (b) of Lemma 2. This finishes the proof of our main theorem.

## 3 The Median

There is an elegant alternative proof of Lemma 2 in terms of the median.

**Fact 2.** *For any random variable $X$ and any strictly monotone function $f$ we have $\mathbf{m}f(X) = f(\mathbf{m}X)$. In the discrete case the medians can be chosen to have this property.*

*Proof.* Simply observe that for any $a$ we have $\mathbf{Pr}(X \leq a) = \mathbf{Pr}(f(X) \leq f(a))$. Here we do require the strict monotonicity. □

**Fact 3.** *For any random variable $X$, the median $\mathbf{m}X$ deviates from the mean $\mathbf{E}X$ by at most $\delta X$, i.e. $\mathbf{m}X \in [\mathbf{E}X - \delta X, \mathbf{E}X + \delta X]$.*

*Remark.* This also establishes the (weaker) fact that for any random variable $X$, the median $\mathbf{m}X$ always lies in the interval $[\mathbf{E}X - \sigma X, \mathbf{E}X + \sigma X]$, which is mentioned in the literature [6], but, according to a small survey of ours, seems to be little known among theoretical computer scientists. When the distribution of $X$ is unimodal, the difference between the mean and the median can even be bounded by $\sqrt{3/5} \cdot \sigma$ [7]. By what is shown below, we may in that case replace $\delta$ by $\sqrt{3/5} \cdot \sigma$ in Theorem 1.

*Proof.* Fact 3 is an immediate consequence of Lemma 1 by noting that (for continuous random variables) $\mathbf{Pr}(X \leq \mathbf{m}X) = \mathbf{Pr}(X \geq \mathbf{m}X) = 1/2$ and taking $a = \delta X$. Alternatively, we could mimic the proof of that lemma. □

These two simple facts are the heart and soul underlying Theorem 1 in the sense that the two inequalities of Lemma 2 now have the following very short and elegant alternative proofs:

$$\begin{aligned} \mathbf{E}f(X) - \delta f(X) \;&\leq\; \mathbf{m}f(X) \;=\; f(\mathbf{m}X) \;\leq\; f(\mathbf{E}X + \delta X) \\ \mathbf{E}f(X) + \delta f(X) \;&\geq\; \mathbf{m}f(X) \;=\; f(\mathbf{m}X) \;\geq\; f(\mathbf{E}X - \delta X) \end{aligned}$$

where the inequalities follow from Fact 3 and the monotonicity of $f$, and the equalities are just restatements of Fact 2.

Given Theorem 1 and Fact 2, the question arises whether not the median should generally be preferred over the mean when looking for an "average" value?

One strong argument that speaks against the median is the following. By the (weak) law of large numbers, the average over a large number of independent trials will be close to the mean, not to the median. In fact, by exactly the kind of considerations given in our introduction, the order of the medians could be the opposite of the order of the averages, which would be deceptive when in practice there were indeed a large number of independent runs of the algorithm.

A pragmatic argument is that the mean can be computed much easier: the values to be averaged over can simply be summed up without a need to keep them in memory. For the median, it is known that such a memoryless computation does not exist [8]; even approximations have to use a non-constant number of intermediate variables, and the respective algorithms are far from being simple [9].

## 4 Relaxations of the Main Theorem

In this section, we show that the result from the previous section cannot be relaxed in any obvious way, as stated in Theorem 1.

We try to find examples which are realistic in the sense that the $f$ is well-behaved and the distributions are simple. We do so to emphasize that all conditions are also of practical relevance. First, observe that if the function $f$ is strictly increasing it also has a strictly increasing inverse function $f^{-1}$. This allows us to halve the number of cases we have to consider from four to two: any counter-example for the case where $\delta X$ is dropped from the statement of Theorem 1, gives a counter-example for the case where $\delta f(Y)$ is dropped, by replacing $f(X) \to U$ and $f(Y) \to V$, where $U$ and $V$ are also random variables, and the same symmetry relates $\delta Y$ to $\delta f(X)$.

To prove that we cannot drop the $\delta X$ (and hence neither the $\delta f(y)$) from the statement of Theorem 1, we consider an example where $Y$ is constant. Then we find an example of a distribution for $X$ and a strictly increasing function $f$ such that

$$\begin{aligned} \mathbf{E}X < Y \qquad & \text{and} \\ \mathbf{E}f(X) - \delta f(X) > f(Y) \ . & \end{aligned}$$

The obvious thing works: We let $X$ have a two-point distribution with points $x_1 < Y$ and $x_2 > Y$ and consider a function which is convex, e.g. $f(x) = e^x$. For this setting we try to solve the system

$$\begin{aligned} p_1\, x_1 + p_2\, x_2 < Y & \\ p_1\, f(x_1) + p_2\, f(x_2) - 2\, p_1\, p_2\ (f(x_2) - f(x_1)) < f(Y) \ . & \end{aligned} \tag{1}$$

It becomes slightly easier to spot solutions to this if we write $p_1 = \frac{1}{2} - \delta$ and $p_2 = \frac{1}{2} + \delta$. Then (1) becomes

$$2\, p_1\, f(x_1)\, (1 + \delta) + 2\, p_2 f(x_2)\, \delta < f(Y) \ . \tag{2}$$

Thus as long as $\delta > 0$ and $f$ increases 'fast enough' in the region between $Y$ and $x_2$ we can always construct a simple counter-example as $f(x_2) >> f(Y)$. For example, take $Y = 2$, $p_1 = \frac{1}{4}$, $p_2 = \frac{3}{4}$, $x_1 = -2$, $x_2 = 3$. Similarly, we can find a two point counter-example for the case without the $\delta Y$ by considering a logarithmic function. One such example consists of a constant $X = 1$, $p_1 = \frac{3}{4}$, $p_2 = \frac{1}{4}$, $y_1 = .5$, $y_2 = 3$ and $f(x) = \log(x)$.

If we restrict ourselves, as we have done, to the case where only one of $X$ and $Y$ is random we see from Jensen's inequality that we must indeed consider examples with the curvatures as chosen above. Otherwise, it would be impossible to find a counter-example.

The same examples still work if we allow $Y$ to have a small degree of variation.

## 5 Conclusions

Theorem 1 ensures that when conclusions are drawn *only* when the error bands do not intersect, there will at least never be contradictions from the angle of different measurement scales. The bad news is that, even when the error bands do not intersect in one scale, in general nothing can be inferred about the order of the averages after a monotone transformation.

Obviously, when two sets of measurements are completely separated in the sense that the largest measurement of one set is smaller than the smallest measurement of the other set, then no monotone transformation can reverse the order of the averages. Beyond that, however, there does not seem to be any less restrictive natural precondition, which most datasets would fulfill and under which average reversal provably cannot occur.

What *can* be proven is that for two random variables $X$ and $Y$, if $0 \leq \mathbf{E}(X - \mathbf{E}X)^k \leq \mathbf{E}(Y - \mathbf{E}Y)^k$ for all $k \in \mathbb{N}$, then for a monotonously increasing function $f$, with all derivatives also monotone (as is the case for any monomial $x \mapsto x^k$ with $k \in \mathbb{N}$, and for any exponential $x \mapsto b^x$ with $b > 1$), indeed $\mathbf{E}(X) \leq \mathbf{E}(Y) \Rightarrow \mathbf{E}f(X) \leq \mathbf{E}f(Y)$. Unfortunately, this precondition is neither practical to check nor generally fulfilled. For example, consider two random variables with different exponential distributions, both mirrored around the mean: then one random variable will have smaller mean *and* variance than the other, yet its third central moment (which is negative), will be larger.

The bottom line of our findings is that, in case there is an option, there is no alternative other than to explicitly provide a comparison in each cost measure that is of interest. Anyway, it should be clear that even in one fixed cost measure, an average comparison alone does not say much: it is well known (see [10] for an easy-to-read account) that even when the error bands do not intersect, the apparent order of the averages is statistically not significant. Comparing averages can be a very deceptive thing. Hence our title.

## References

1. Manning, C.D., Schütze, H.: Foundations of statistical natural language processing. MIT Press (1999)
2. Teh, Y.W., Jordan, M.I., Beal, M.J., Blei, D.M.: Sharing clusters among related groups: Hierarchical dirichlet processes. In: Proceedings of the Advances in Neural Information Processings Systems Conference (NIPS'04), MIT Press (2004)
3. Lavrenko, V., Croft, W.B.: Relevance based language models. In: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'01), ACM Press (2001) 120–127
4. Mori, S., Nagao, M.: A stochastic language model using dependency and its improvement by word clustering. In: Proceedings of the 17th international conference on Computational linguistics (COLING'98), Association for Computational Linguistics (1998) 898–904
5. Grimmett, G., Stirzaker, D.: Probability and Random Processes. Oxford University Press (1992)

6. Siegel, A.: Median bounds and their application. Journal of Algorithms **38** (2001) 184–236
7. Basu, S., Dasgupta, A.: The mean, median and mode of unimodal distributions: A characterization. Theory of Probability and its Applications **41** (1997) 210–223
8. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science **12** (1980) 315–323
9. Manku, G.S., Rajagopalan, S., Lindsay, B.G.: Approximate medians and other quantiles in one pass and with limited memory. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98). (1998) 426–435
10. Motulsky, H.: The link between error bars and statistical significance. (`http://www.graphpad.com/articles/errorbars.htm`)

## A The Example from the Introduction

Each point in the figures in the introduction was computed as the average over points distributed as $Z = z_0 + \mathrm{Exp}(\lambda)$, where $\mathrm{Exp}(\lambda)$ denotes the exponential distribution with mean $1/\lambda$ (density is $\varphi(t) = \lambda\, e^{-\lambda t}$; variance is $1/\lambda^2$).

For the mean of $2^Z$, or more generally, $e^{\kappa Z}$, we have that

$$\begin{aligned}\mathbf{E}e^{\kappa Z} &= \int_0^\infty e^{\kappa(z_0+t)}\lambda e^{-\lambda t}\,dt \\ &= e^{\kappa z_0}\cdot\lambda/(\lambda-\kappa) \\ &\approx e^{\kappa(z_0+1/(\lambda-\kappa))} \\ &= e^{\kappa(z_0+1/\lambda+\kappa/(\lambda(\lambda-\kappa)))}.\end{aligned}$$

For the figures in the introduction, we chose $z_0$ so that the means for each curve would lie on a slightly perturbed line. For the light gray curve, we chose $\lambda = 1$, for the dark gray curve we chose $\lambda = 2$. For example, for $X = 3 + \mathrm{Exp}(1)$ and $Y = 5 + \mathrm{Exp}(2)$, we then have

$$\begin{aligned}\mathbf{E}X &= 3 + 1/1 = 4 \\ \mathbf{E}Y &= 5 + 1/2 = 5.5,\end{aligned}$$

and for $\kappa = 3/4$ (then $e^\kappa \approx 2$),

$$\begin{aligned}\mathbf{E}e^{\kappa X} &\approx e^{\kappa(3+1.0+3.0)} \approx 2^{7.0} \\ \mathbf{E}e^{\kappa Y} &\approx e^{\kappa(4+0.5+0.3)} \approx 2^{5.8}.\end{aligned}$$

Observe that in this setting we need $\kappa < \lambda_1$ and $\kappa < \lambda_2$ to ensure that both $\mathbf{E}e^{\kappa X}$ and $\mathbf{E}e^{\kappa Y}$ exist.

One objection against the exponential distribution might be that its exponentiation is too heavy-tailed in the sense that not all its moments exist. However, the same calculations as above can also be carried out for two, say, normal distributions, which are free from this taint. Let $Z = \mathrm{N}(z_0, \sigma)$, that is, $Z$ has a

normal distribution with mean $z_0$ and standard deviation $\sigma$. A straightforward calculation shows that the mean of $e^{\kappa Z}$, which obeys a lognormal distribution, is given by

$$\begin{aligned}\mathbf{E}e^{\kappa Z} &= \int_{-\infty}^{\infty} e^{\kappa t} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(t-z_0)^2/(2\sigma^2)}\, dt \\ &= e^{\kappa z_0 + \kappa^2\sigma^2/2}.\end{aligned}$$

Taking, for example, $X = \mathrm{N}(4, 1.5)$ and $Y = \mathrm{N}(4.5, 1.0)$ and $\kappa = 1$, we then get the following reversed means after exponentiation:

$$\begin{aligned}\mathbf{E}e^{\kappa X} &= e^{\kappa 4 + \kappa^2 1.5^2/2} = e^{5.125} \\ \mathbf{E}e^{\kappa Y} &= e^{\kappa 4.5 + \kappa^2/2} = e^{5}.\end{aligned}$$

## B One-Sided Chebyshev Bounds

For the sake of completeness, we state the one-sided version of Chebyshev's inequality, which looks similar to Lemma 1 in Sect. 2. As mentioned in that section, Lemma 1 is stronger for deviations up to at least $\sigma X$, while the lemma below is stronger for large deviations.

**Lemma 3.** *For any random variable $X$ and for every $a > 0$, it holds that*

*(a)* $\mathbf{Pr}(X \geq \mathbf{E}X + a) \leq \frac{(\sigma X)^2}{a^2 + (\sigma X)^2}$;
*(b)* $\mathbf{Pr}(X \leq \mathbf{E}X - a) \leq \frac{(\sigma X)^2}{a^2 + (\sigma X)^2}$.

*Proof.* See, for example, [5]. The main idea is to write $\mathbf{Pr}(X \geq \mathbf{E}X + a) = \mathbf{Pr}((X - \mathbf{E}X + c)^2 \geq (a + c)^2)$, then apply Markov's inequality and determine that $c$ which gives the best bound; similarly for (b). □

# Experimental Results for Stackelberg Scheduling Strategies*

A.C. Kaporis[1], L.M. Kirousis[1,2], E.I. Politopoulou[1,2], and P.G. Spirakis[1,2]

[1] Department of Computer Engineering and Informatics,
University of Patras, Greece
{kaporis, kirousis, politop, spirakis}@ceid.upatras.gr
[2] Research Academic Computer Technology Institute,
P.O Box 1122, 26110 Patras, Greece
{politop, spirakis}@cti.gr

**Abstract.** In large scale networks users often behave selfishly trying to minimize their routing cost. Modelling this as a noncooperative game, may yield a *Nash* equilibrium with unboundedly poor network performance. To measure this inefficacy, the *Coordination Ratio* or *Price of Anarchy* (*PoA*) was introduced. It equals the ratio of the cost induced by the worst Nash equilibrium, to the corresponding one induced by the overall optimum assignment of the jobs to the network. On improving the *PoA* of a given network, a series of papers model this selfish behavior as a *Stackelberg* or *Leader-Followers* game.

We consider random tuples of machines, with either linear or M/M/1 latency functions, and *PoA* at least a *tuning parameter c*. We validate a variant (`NLS`) of the *Largest Latency First* (`LLF`) Leader's strategy on tuples with $PoA \geq c$. `NLS` experimentally improves on `LLF` for systems with inherently high *PoA*, where the Leader is *constrained* to control low portion $\alpha$ of jobs. This suggests even better performance for systems with arbitrary *PoA*. Also, we bounded experimentally the least Leader's portion $\alpha_0$ needed to induce optimum cost. Unexpectedly, as parameter $c$ increases the corresponding $\alpha_0$ decreases, for M/M/1 latency functions. All these are implemented in an extensive *Matlab* toolbox.

## 1 Introduction

We consider the problem of system resource allocation [28]. This problem is one of the basic problems in system management, though systems today have high availability of bandwidth and computational power.

* The 2nd and 4th author are partially supported by Future and Emerging Technologies programme of the EU under EU contract 001907 "*Dynamically Evolving, Large Scale Information Systems (DELIS)*". The 1st, 2nd and 3rd author are partially supported by European Social Fund (ESF), Operational Program for Educational and Vacational Training II (EPEAEK II), and particularly *PYTHAGORAS*.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 77–88, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Such systems are large scale networks, for example broadband [27], wireless [8] and peer to peer networks[7] or Internet. The users have the ability to select their own route to their destination with little or no limitation [4, 20, 9]. Since the users are instinctively selfish, they may use the right of path selection and may select a route that maximizes their profit. This selfish routing behavior can be characterized by a fixed state, which in game theory is called *Nash Equilibrium* [15, 18]. In this context, the interested reader can find much of theoretic work in [15, 12, 14, 9, 11, 19, 20, 21].

However, Nash Equilibrium may lead a system to suboptimal behavior. As a measure of how worse is the Nash equilibrium compared to the overall system's optimum, the notion of *coordination ratio* was introduced in [12, 14]. Their work have been extended and improved (*price of anarchy* here is another equivalent notion) in [14, 24, 3, 22, 23, 6, 4].

Selfish behavior can be modeled by a *non-cooperative game*. Such a game could impose strategies that might induce an equilibrium closer to the overall optimum. These strategies are formulated through pricing mechanisms[5], algorithmic mechanisms[16, 17] and network design[25, 10]. The network administrator or designer can define prices, rules or even construct the network, in such a way that induces near optimal performance when the users selfishly use the system.

Particulary interesting is the approach where the network manager takes part to the non-cooperative game. The manager has the ability to control centrally a part of the system resources, while the rest resources are used by the selfish users. This approach has been studied through *Stackelberg* or *Leader-Follower* games [2, 23, 9, 11, 26]. The advantage of this approach is that it might be easier to be deployed in large scale networks. This can be so, since there is no need to add extra components to the network or, to exchange information between the users of the network.

Let us concentrate on the setting of this approach. The simplified system consists of a set of machines with load depended latency functions and a flow of jobs with rate $r$. The manager controls a fraction $\alpha$ of the flow, and assigns it to machines in a way that the induced cost by the users is near or equals the overall optimal. An interesting issue investigated in [23, 9], is how should the manager assign the flow he controls into the system, as to induce optimal cost by the selfish users. For the case of linear load functions, in [23] was presented a polynomial algorithm (LLF) of computing a strategy with cost at most $\frac{4}{3+\alpha}$ times the overall optimum one, where $\alpha$ is the fraction of the rate that the manager controls. Korilis et al [9] has initiated this game theoretic approach and investigated on the necessary conditions such that the manager's assignment induces the optimum performance on a system with M/M/1 latency functions.

### 1.1 Motivation and Contribution

Our work is motivated by the work in [1, 23, 9]. We consider a simple modification of the algorithm *Largest Latency First* (LLF) [23] called *New Leader Strategy*

(NLS). Experiments suggest that NLS has better performance in competitive systems of machines, that is, systems with high value of price of anarchy $PoA$. Also, it has good performance in cases where the Leader may be constrained to use a small portion $\alpha$ of flow. Notice that $PoA \leq 4/3$ for linear latency functions. Furthermore, a highly nontrivial algorithm presented in [1] slightly improves over LLF for the case of linear latency functions. Then, despite its simplicity, our heuristic has comparatively good performance.

Additionally, we conducted thousands random tuples of machines, with either linear or M/M/1 latency functions. We experimentally tried to compute an upper bound $\alpha_0$ for the least possible portion of flow that a Leader needs to induce overall optimal behavior. We have considered tuples of machines with M/M/1 latency functions such that their price of anarchy is at least a parameter $c$. Surprisingly, as parameter $c$ increases (resulting to more competitive systems of machines), the average value of $\alpha_0$ *decreases*.

## 2 Improved Stackelberg Strategies

### 2.1 Model - Stackelberg Strategies

For this study the model and notation of [23] is used. We consider a set $M$ of $m$ machines, each with a latency function $\ell(\cdot) \geq 0$ continuous and nondecreasing, that measures the load depended time that is required to complete a job. Jobs are assigned to $M$ in a finite and positive rate $r$. Let the $m$-vector $X \in \mathcal{R}_+^m$ denote the assignment of jobs to the machines in $M$ such that $\sum_{i=1}^m x_i = r$. The latency of machine $i$ with load $x_i$ is $\ell_i(x_i)$ and incurs cost $x_i\ell_i(x_i)$, convex on $x_i$. This instance is annotated $(M, r)$. The *Cost* of an assignment $X \in \mathcal{R}_+^m$ on the $(M, r)$ instance is $C(X) = \sum_{i=1}^m x_i\ell_i(x_i)$, measuring system's performance. The minimum cost is incurred by a unique assignment $O \in \mathcal{R}_+^m$, called the *Optimum* assignment. The assignment $N \in \mathcal{R}_+^m$ defines a *Nash equilibrium*, if no user can find a loaded machine with lower latency than any other loaded machine. That is, all machines $i$ with load $n_i > 0$ have the same latency $L$ while any machine $j$ with load $n_j = 0$ has latency $L_j \geq L$. According to the Definition 2.1 in [23]:

**Definition 1.** *An assignment $N \in \mathcal{R}_+^m$ to machines $M$ is at Nash equilibrium (or is a Nash assignment) if whenever $i, j \in M$ with $n_i > 0$, $\ell_i(n_i) \leq \ell_j(n_j)$.*

The Nash assignment $N$ causes cost $C(N)$ commonly referred to as *Social Cost* [15, 12, 14, 9, 11, 19, 20, 21]. The social cost $C(N)$ is higher than the optimal one $C(O)$, leading to a degradation in system performance. The last is quantified via the *Coordination Ratio*[12, 14, 3] or *Price of Anarchy (PoA)* [24], i.e. the worst-possible ratio between the social cost and optimal cost: $PoA = \frac{C(N)}{C(O)}$, and the goal is to minimize $PoA$ [1] To do so, a *hierarchical non cooperative Leader-Follower* or *Stackelberg game* is used [2, 23, 9, 11, 26]. In such a game, there is a

[1] Notice that in a general setting may exist a set $A$ of Nash equilibria, then $PoA$ is defined with respect to worst one, i.e. $PoA = \max_{N \in A} \frac{C(N)}{C(O)}$.

set $M$ of machines, jobs with flow rate $r$ and a distinguished player or *Leader* who is responsible for assigning centrally an $\alpha$ portion of the rate $r$ to the system so as to decrease the total social cost of the system. The rest of the players, called *Followers* are assigning selfishly the remaining $(1-\alpha)r$ flow in order to minimize their individual cost. This instance is called *Stackelberg instance* and is annotated by $(M, r, \alpha)$. The Leader assigns $S \in \mathcal{R}_+^m$ to $M$ and the Followers react, inducing an assignment in Nash equilibrium. We give the same definition for an *induced assignment at Nash Equilibrium* or *induced Nash assignment* as in Definition 2.7 of [23].

**Definition 2.** *Let $S \in \mathcal{R}_+^m$ be a Leader assignment for a Stackelberg instance $(M, r, \alpha)$ where machine $i \in M$ has latency function $\ell_i$, and let $\widetilde{\ell}_i(x_i) = \ell_i(s_i + x_i)$ for each $i \in M$. An equilibrium induced by $S$ is an assignment $T \in \mathcal{R}_+^m$ at Nash equilibrium for the instance $(M, (1-a)r)$ with respect to latency function $\widetilde{\ell}$. We then say that $S + T$ is an assignment induced by $S$ for $(M, r, \alpha)$.*

The goal is achieved if $C(S+T) \simeq C(O)$.

We consider here two types of latency functions: linear and M/M/1. Linear latency functions have the form $\ell_i(x_i) = a_i x_i + b_i,\ i \in M,\ X \in \mathcal{R}_+^m$ and it holds $PoA \leq \frac{4}{3}$. M/M/1 latency functions have the form $\ell_i(x_i) = \frac{1}{u_i - x_i},\ i \in M,\ X \in \mathcal{R}_+^m$ and it holds $PoA \leq \frac{1}{2}\left(1 + \sqrt{\frac{u_{min}}{u_{min} - R_{max}}}\right)$, where $u_{min}$ is the smallest allowable machine capacity and $R_{max}$ is the largest allowable traffic rate.

Finally, to tune the competitiveness of a particular system $M$, we define the parameter $c$ as a lower bound of its $PoA$. Thus, systems with highly valued parameter $c$ are particularly competitive.

### 2.2 Algorithm NLS

**Algorithm:** NLS$(M, r, \alpha)$
**Input:** Machines $M = \{M_1, \ldots, M_m\}$, flow $r$, and portion $\alpha \in [0, 1]$
**Output:** An assignment of the load $\alpha r$ to the machines in $M$.
**begin:**
  Compute the global Optimum assignment $O = \langle o_1, \ldots, o_m \rangle$ of flow $r$ on $M$;
  Compute the Nash assignment $N = \langle n_1, \ldots, n_m \rangle$ of the flow $(1-\alpha)r$ on $M$;
  Let $M^* = \{M_i \in M \mid n_i = 0\}$;
  **If** $\sum_{\{i:M_i \in M^*\}} o_i \geq \alpha r$ **then** assign local optimally the flow $\alpha r$ on $M^*$;
    **else** assign the flow $\alpha r$ on $M$ according to LLF;
  **end if**;
**end;**

Notice that it is possible to heuristically compute an even larger subset $M^*$ unaffected by the Followers, allowing us to assign to it a even larger portion $\alpha' > \alpha$ of flow.

In [23] it was presented the *Large Latency First* (LLF) Stackelberg strategy for a *Leader* that controls a portion $\alpha$ of the total flow $r$ of jobs, to be scheduled to

a system of machines $M$. For the case of machines with linear latency functions, it was demonstrated that the induced scheduling cost is at most $\frac{4}{3+\alpha}$ of the optimum one.

We present and validate experimentally the *New Leader Strategy* (`NLS`). Our motivation was a system of machines presented in [23], end of page 17. In that example, the set of machines is $M = \{M_1, M_2, M_3\}$ with corresponding latency functions $\ell_1(x) = x, \ell_2(x) = x + 1, \ell_3(x) = x + 1$. `LLF` at first computes the optimum assignment $O = \langle o_1 = \frac{4}{6}, o_2 = \frac{1}{6}, o_3 = \frac{1}{6}\rangle$, of the total flow $r = 1$ to the given set of machines $M$. On each machine $i$, load $o_i$ incurs latency value $\ell_i(o_i),\ i = 1, \ldots, 3$. Then, `LLF` indexes the machines, from lower to higher latency values, computed at the corresponding optimum load. In this example, the initial indexing remains unchanged, since: $\ell_1(o_1) \leq \ell_2(o_2) \leq \ell_3(o_3)$. In the sequel, it computes a Stackelberg scheduling strategy $S = \langle s_1, s_2, s_3\rangle$ for the Leader as follows. `LLF` schedules the flow $\alpha r$ that Leader controls, filling each machine $i$ up to its optimum load $o_i$, proceeding in a "largest latency first" fashion. At first, machine 3 is assigned a flow at most its optimum load $o_3$. If $\alpha r - o_3 > 0$, then machine 2 is assigned a flow at most its optimum load $o_2$. Finally, if $\alpha r - o_3 - o_2 > 0$, then machine 1 receives at most its optimum load $o_1$. Notice that all selfish followers prefer the first machine, i.e the Nash assignment is $N = \langle n_1 = 1, n_2 = 0, n_3 = 0\rangle$, since the total flow equals $r = 1$. Provided that no Follower affects the load assignment $S$ of the Leader to the subset of machines $M' = \{2, 3\}$, a crucial observation is that strategy $S$ computed by `LLF` is not always optimal. It is optimal only in the case that the portion $\alpha$ of Leader equals: $\alpha = \frac{o_2+o_3}{r}$. In other words, the assignment of the Leader would be optimal if its flow was enough to fill all machines 2 and 3 up to their corresponding optimal loads $o_2, o_3$. Taking advantage of this, a better Stackelberg strategy is: $S' = \langle s'_1 = 0,\ s'_2 = o^*_2,\ s'_3 = o^*_3\rangle$, where $o^*_2$ and $o^*_3$ are the corresponding *local* optimum loads, of the flow $\alpha r$ that a Leader controls, on the subset of the machines $\{2, 3\}$ which are not appealing for the Followers.

To illustrate this, consider any $\alpha < o_2 + o_3 = \frac{1}{6} + \frac{1}{6}$, for example let $\alpha = \frac{1}{6}$. Then `LLF` computes the Stackelberg strategy $S = \langle 0, 0, \frac{1}{6}\rangle$, inducing the Nash assignment $N = \langle \frac{5}{6}, 0, \frac{1}{6}\rangle$ with cost $C_S = \left(\frac{5}{6}\right)^2 + \left(1 + \frac{1}{6}\right)\frac{1}{6} = \frac{8}{9}$. However, the *local* optimum assignment of the flow $\alpha = \frac{1}{6}$ to machines 2 and 3 is $S' = \langle 0, \frac{1}{12}, \frac{1}{12}\rangle$. This induces the Nash assignment $N' = \langle \frac{5}{6}, \frac{1}{12}, \frac{1}{12}\rangle$ with cost $C_{S'} = \left(\frac{5}{6}\right)^2 + \left(1 + \frac{1}{12}\right)\frac{1}{6} = \frac{7}{8} < \frac{8}{9}$.

We propose algorithm `NLS` that takes advantage all these issues discussed above. Intuitively, it tries to compute a *maximal* subset $M^* \subseteq M = \{1, \ldots, m\}$ of machines not appealing to the selfish users. This subset $M^* \subseteq M$ consists of exactly those machines that receive no flow by the Nash assignment of $(1 - \alpha)r$ flow on $M$. Then it assigns the portion $\alpha r$ of a Leader local optimally on $M^*$. The empirical performance of `NLS` is presented in Section 4, in Figures 2 and 4.

## 3 Algorithm OpTop

### 3.1 Description

In [23] (also see the important results in [9] for the case of M/M/1 latency functions) it was possed the important question:

> "Compute the minimum flow of jobs that a Leader may play according to a Stackelberg scheduling strategy to a system of machines, in a way that the selfish play of the Followers leads the system to its optimum cost."

In this section, we investigate this issue experimentally for the case of machines with linear latency functions. Algorithm OpTop below (based on features of LLF), tries to control a minimal portion $\alpha$ of the total flow $r$ of jobs. It schedules this flow to a system of $m$ machines, in a way that the selfish play of the Followers drives the system to its optimum cost. Intuitively, OpTop tries to find a small subset of machines that have the following stabilizing properties:

- The selfish play of the Followers will not affect the flow $\alpha r$ assigned by the Leader optimally to these machines.
- The selfish play of the Followers of the remaining $(1-\alpha)r$ flow on the remaining machines will drive the system to its optimum cost.

**Algorithm: OpTop** $(M,\ r,\ r_0)$
**Input:** Machines $M = \{M_1, \ldots, M_m\}$, flow $r$, initial flow $r_0$
**Output:** A portion $\alpha$ of flow rate $r_0$
**begin:**
  Compute the Nash assignment $N := \langle n_1, \ldots, n_m \rangle$ of flow $r$ on machines $M$;
  Compute the Optimum assignment $O := \langle o_1, \ldots, o_m \rangle$ of flow $r$ on machines $M$;
  **If** $(N \equiv O)$ **return** $(r_0 - r)/r_0$;
    **else** $(M, r) \leftarrow$ Simplify$(M, r, N, O)$;
    **return** OpTop$(M, r, r_0)$;
  **end if;**
**end;**
**Procedure: Simplify**$(M, r, N, O)$
**Input:** Machines $M = \{M_1, \ldots, M_m\}$, flow $r$
  Nash assignment $N := \langle n_1, \ldots, n_m \rangle$
  Optimum assignment $O := \langle o_1, \ldots, o_m \rangle$
**Output:** Set of machines $M$, Flow $r$
**begin:**
  **for** $i = 1$ **to** size$(M)$ **do:**
    **If** $o_i \geq n_i$ **then**
      $r \leftarrow r - o_i$;
      $M \leftarrow M \backslash \{M_i\}$;
    **end if;**
  **end for;**
**end;**

The key features of OpTop are presented with the help of Figures 1a, 1b, 1c. The corresponding Nash and Optimum assignments to these machines are de-

noted as: $N = \langle n_1, \ldots, n_5 \rangle$, such that $\sum_{i=1}^{5} n_i = r$, $O = \langle o_1, \ldots, o_5 \rangle$, such that $\sum_{i=1}^{5} o_i = r$.

**Definition 3.** *Machine $i$ is called* over-loaded *(or* under-loaded*) if $n_i > o_i$ (or $n_i < o_i$). Machine $i$ is called optimum-loaded if $n_i = o_i, i = 1, \ldots, m$.*

Initially, the algorithm assigns to all under-loaded machines in Figure 1a their optimum load. That is, it assigns optimum load $o_4$ and $o_5$ to machines 4 and 5 in Figure 1b. Then the players selfishly assign the remaining $r - o_4 - o_5$ flow to the system of 5 machines. Observe that in the induced Nash assignment, none of the machines 4 and 5 receives flow. That is, machines 4 and 5 have been stabilized to their optimum load, *irrespectively* of the selfish behavior of the Followers, see also Theorem 1.

A crucial point is that we can remove machines 4 and 5 from consideration and run recursively `OpTop` on the simplified system of machines. In other words, the induced game now is equivalent to scheduling the remaining $r - o_4 - o_5$ flow to the remaining machines 1, 2 and 3, see also Lemma 1.

In the sequel, in the simplified game, now machine 3 has become under-loaded and 2 optimum-loaded, while 1 remains over-loaded, see Figure 1b. In the same fashion as in the original game, `OpTop` assigns load $o_3$ to machine 3. Happily, the induced selfish scheduling of the remaining $r - o_3 - o_4 - o_5$ flow yields the overall optimum assignment for the system. That is, the remaining $r - o_3 - o_4 - o_5$ flow, when scheduled selfishly by the Followers, ignores machines 3, 4 and 5 (they assign no load to these machines) while their selfish behavior assigns induced Nash load $n'_i = o_i$ to each machine $i = 1, 2$, see Figure 1c.

In this example, algorithm `OpTop` needed to control a portion $\alpha_0 = \frac{o_3 + o_4 + o_5}{r}$, of the total flow $r$ of jobs, in a way that the final induced load to each machine $i$ equals its optimum value $o_i$, $i = 1, \ldots, 5$. `OpTop`'s objective is to impose the overall optimum by controlling the least possible portion $\alpha_0$. The cornerstone for the stability of the load assigned by `OpTop` to any machine is Theorem 1. Intuitively, this theorem says that `OpTop` raises the latency of proper machines

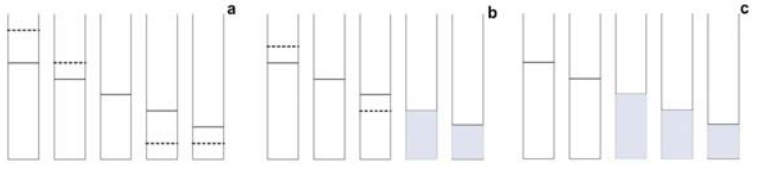



**Fig. 1.** A dashed (or solid) line indicates the Nash (or Optimum) load $n_i$ (or $o_i$) assigned to each machine $i = 1, \ldots, 5$. (a) Machines 1 and 2 (4 and 5) are over(under)-loaded while 3 is optimum-loaded. Then `OpTop` will assign load $o_4$ and $o_5$ to machines 4 and 5. (b) Now machines 4 and 5 received load $o_4$ and $o_5$ by `OpTop`. In the induced Nash assignment, machines 1 (3) become over(under)-loaded while 2 becomes optimum-loaded.(c) Finally, `OpTop` assigns load $o_3$ to machine 2. In the induced Nash assignment, machines 1 and 2 become optimum-loaded

sufficiently high, making them not appealing to selfish players, while retaining their respective optimum load.

**Theorem 1.** *Consider $m$ machines with latency functions $\ell_j(x) = a_j x + b_j,\ j = 1, \ldots, m$. Let the Nash assignment $N = \langle n_1, \ldots, n_m \rangle$ of the total load $r$ on the $m$ machines. Suppose that for a Stackelberg strategy $S = \langle s_1, \ldots, s_m \rangle$ we have either $s_j \geq n_j$ or $s_j = 0, j = 1, \ldots, m$. Then for the induced Nash assignment $T = \langle t_1, \ldots, t_m \rangle$ of the remaining load $r - \sum_{i=1}^{m} s_i$ we have that $t_j = 0$ for each machine $j$ such that $s_j \geq n_j,\ j = 1, \ldots, m$.*

*Proof.* By assigning load $s_j \geq n_j$ to machine $j$ then for any induced load $t_j \geq 0$ to it, its latency is now increased up to $\widetilde{\ell}_j(t_j) = a_j t_j + \ell_j(s_j) \geq \ell_j(s_j) \geq \ell_j(n_j),\ j = 1, \ldots, m$. Since the induced Nash assignment $T$ assigns total load $r - \sum_{i=1}^{m} s_i \leq \sum_{\{i:s_i=0\}} n_i$, its is not now possible for any machine $j$ with $s_j \geq n_j$ to become appealing to the selfish users, $j = 1, \ldots, m$.

Theorem 1 is valid for arbitrary increasing latency functions. Interestingly, a similar (monotonicity) argument can be found in [13]. Another difficulty for the evolution of the algorithm, is to describe the selfish play of the users in the remaining machines. To this end, Lemma 1 is helpful.

**Lemma 1.** *Let a set of machines $M = \{M_1, \ldots, M_m\}$ and the Nash assignment $N = \langle n_1, \ldots, n_m \rangle$ of the total load $r$ on these machines. Fix a Stackelberg strategy $S = \langle s_1, \ldots, s_m \rangle$ such that either $s_j \geq n_j$ or $s_j = 0, j = 1, \ldots, m$. Then the initial game is equivalent to scheduling total flow: $r - \sum_{i=1}^{m} s_i$, to the subset $M' \subseteq M$ of machines: $M' = M \backslash \{M_i :\ s_i \geq n_i\},\ i = 1, \ldots, m$.*

*Proof.* It follows from Theorem 1.

Lemma 1 allows us to run recursively `OpTop` on the simplified game on the remaining machines. The empirical performance of `OpTop` is presented in Section 4, in Figures 3 and 5.

## 4 Experimental Validation of the Algorithms

All experiments presented in this section are performed using the package Matlab [29]. An extensive toolbox was created for manipulating large systems of machines for both linear and M/M/1 latency functions. All the routines of computing the Optimum and Nash assignments, the `LLF` and `NLS` strategies are also implemented in the same Toolbox [30].

Here we present results for 5-tuples of random machines for both linear and M/M/1 latency functions. Similar results were observed for $k$-tuples with $k \geq 5$. For total flow $r$, machine $i$ receives a portion or flow $x_i$ which incurs latency $\ell(x_i) = a_i x_i + b_i,\ i = 1, \ldots, 5$, where $a_i, b_i$ are random numbers in $[0,\ r]$ and $\sum_{i=1}^{5} x_i = r$. The corresponding random M/M/1 latency functions are $\ell(x_i) = \frac{1}{u_i - x_i},\ i = 1, \ldots, 5$. We created many large collections of 5 tuples of machines,

where each such collection satisfies a predetermined value of the *parameter* $c \leq \frac{4}{3}$ (recall $c$ is a lower bound of the price of anarchy value $PoA$). That is, for each generated random 5-tuple of machines, we compute the ratio of the cost of the Nash assignment to corresponding optimum one, and we store the 5-tuple to the appropriate $c$-valued collection of tuples. Intutively, a collection of tuples with particularly high value of $c$ consists of highly competitive machines and the role of the Leader is important.

### 4.1 Linear Latency Functions

**Comparing `NLS` to `LLF`.** We know that `LLF` strategy induce a Follower assignment that drives the PoA to $\frac{4}{3+\alpha}$. We are interested in finding out how much better the `NLS` strategy does in comparison to `LLF` strategy. In other words we are interested in measuring the ratio $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}}$. The worst case would be when this ratio is 1, which means that the `NLS` strategy is the same as the `LLF` strategy. This phenomenon is expected since `NLS` is based on `LLF` but we are interested in finding out the how much similar is `NLS` to `LLF`. Based on intuition, we expected that in instances with higher values of $PoA$ our strategy will do better than `LLF`. This will be the case with even lower $\alpha$, since we may manipulate more machines in the subset $M^*$ which is described in the pseudo code of `NLS`. This intuition was confirmed by the experiments, as it is shown in Figure 2. Both diagrams present the percentage of machines that had $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}} < 1$. What is remarkable is that `NLS` does better when the parameter $c$ of the machine instances is increased from 1 up to 1.2. Then the corresponding portion of machines that had better performance using `NLS` is increased from 33% up to 62% of the instances.

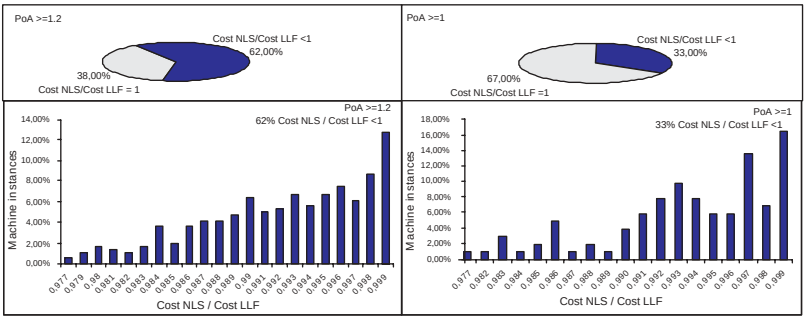



**Fig. 2.** Linear load functions: $\frac{CostNLS}{CostLLF}$ for $PoA \geq 1.2$ and for $PoA \geq 1$

We conjecture that the reason for this phenomenon is that systems with high $PoA$ usually overload 1 or 2 machines, while the rest ones remain idle. Therefore, the $\alpha r$ flow assigned *local* optimally by the Leader to the subset of the idle machines remains unaffected.

Another interesting observation was that `NLS` does better than `LLF` for small $\alpha$. For the instances with $PoA \geq 1$ the `NLS` strategy is better than `LLF` strategy

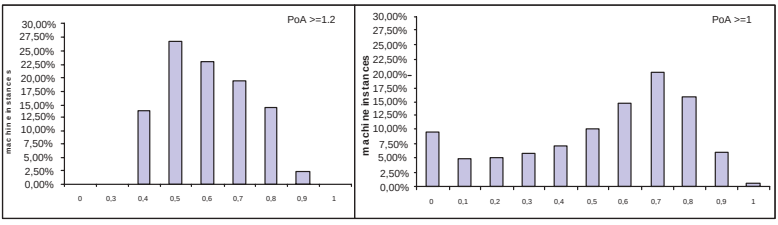



**Fig. 3.** Linear latency functions: The $\alpha_0$ computed by `OpTop` to reach the overall optimum for $PoA \geq 1.2$ and for $PoA \geq 1$

for average $\alpha = 0.132$ while for instances with $PoA \geq 1.2$ the average $\alpha$ is higher and has the value $\alpha$=0.313.

Finally, the average $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}}$ for $PoA \geq 1$ is 0.995 while the $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}}$ for $PoA \geq 1.2$ is 0.991.

**Results for `OpTop`.** The algorithm `OpTop` that we presented in Section 3, computes an upper bound to the amount $\alpha r$ of flow that the Leader must posses in order to induce optimal assignment. That is, we are interested in computing an upper bound to the *minimum flow* $\alpha_0$ that the Leader must control to guarantee overall optimum performance induced by the Followers selfish play. In Figure 3, $x$-axis presents the portion $\alpha_0$ needed to induce the overall optimum, while $y$-axis presents the corresponding percentage of 5-tuples of machines.

The results of our experiments on the set of machine instances are presented in Figure 3 below. In instances where $PoA \geq 1$ the portion $\alpha_0$ of load flow the Leader has to control ranges in $\alpha_0 \in [0,\ 0.9]$ and its average value is $\alpha_0 = 0.5$.

Also in Figure 3, as $PoA$'s lower bound increases up to 1.2, the range of $\alpha_0$ the Leader has to control also increases, that is $\alpha_0 \in [0.4,\ 0.9]$. In this case its average value is $\alpha_0 = 0.6$.

### 4.2 Results for M/M/1 Latency Functions

For M/M/1 latency functions, (i.e. of the form $\frac{1}{u-x}$) the results are similar. The $PoA$ of the systems with such load functions is not that different from the linear load functions. As we can see the `NLS` strategy does better for systems with an increased lower bound (parameter $c$) of $PoA$.

Once more, in Figure 4 we can see that `NLS` does better when the parameter $c$ of the machine instances is increased from 1 up to 1.2. Then the corresponding portion of machines that had better performance using `NLS` is increased from 19% up to 43% of the instances. Furthermore, in the same figure, we see that the average $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}}$ for $PoA \geq 1$ is 0.992 while the $\frac{Cost\texttt{NLS}}{Cost\texttt{LLF}}$ for $PoA \geq 1.2$ is 0.988.

The results of our experiments for `OpTop` on the set of machine instances are presented in Figure 5 below. In instances where $PoA \geq 1$ the portion $\alpha_0$ of flow the Leader has to control to induce the overall optimum ranges in $\alpha_0 \in [0.2,\ 0.9]$ and its average value is $\alpha_0 = 0.57$. Also in this figure, as $PoA$'s lower bound increases up to 1.2, the range of $\alpha_0$ the Leader has to control is in $\alpha_0 \in [0.2,\ 1]$.

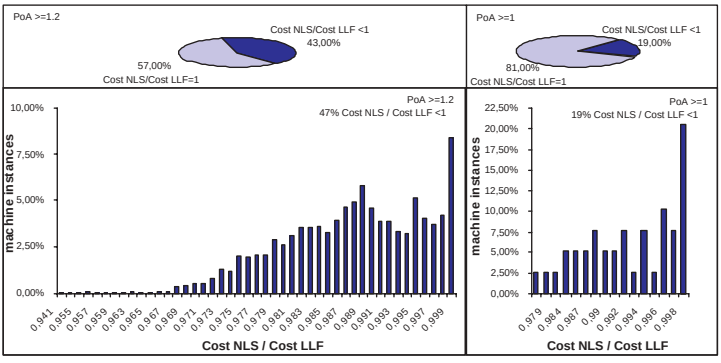



**Fig. 4.** M/M/1 latency functions: $\frac{CostNLS}{CostLLF}$ for $PoA \geq 1.2$ and for $PoA \geq 1$

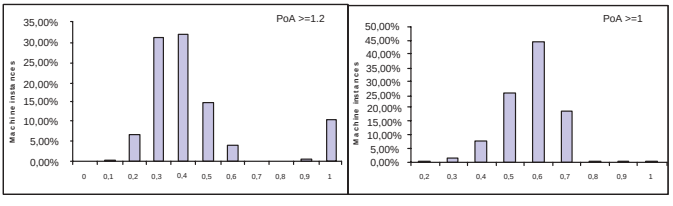



**Fig. 5.** M/M/1 latency functions: The $\alpha_0$ computed by `OpTop` to reach the overall optimum for $PoA \geq 1.2$ and for $PoA \geq 1$

Rather unexpectedly, in this case its average value has been reduced to $\alpha_0 = 0.44$. Further work will focus on machine instances with arbitrary latency functions, where the $PoA$ is greater or even unbounded and the results are expected to be more interesting than those of the linear load functions and M/M/1 functions.

## Acknowledgements

We thank the anonymous referees for their comments that substantially improved the presentation of our ideas.

## References

1. V. S. Anil Kumar, Madhav V. Marathe. Improved Results for Stackelberg Scheduling Strategies. *In Proc. ICALP '02*, pp. 776-787.
2. T. Basar, G. J. Olsder.*Dynamic Noncooperative Game Theory.* SIAM, 1999.
3. A. Czumaj and B. Voecking. Tight bounds for worst-case equilibria. *In Proc. SODA '02.*
4. A. Czumaj. *Selfish Routing on the Internet*, Handbook of Scheduling: Algorithms, Models, and Performance Analysis,CRC Press, 2004.

5. J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *In Proc. STOC '00.*
6. D. Fotakis, S. Kontogiannis, E. Koutsoupias, M. Mavronicolas, P. Spirakis. The structure and complexity of nash equilibria for a selfish routing game. *In Proc. ICALP '02*, pp 123–134.
7. P. Golle, K. Leyton-Brown, I. Mironov, M. Lillibridge. Incentives for Sharing in Peer-to-Peer Networks.Full version, *In Proc. WELCOM 01.*
8. S. A. Grandhi, R. D. Yates and D. J. Goodman: Resource Allocation for Cellular Radio Systems, *In Proc. IEEE Transactions on Vehicular Technology*, vol. 46, no. 3, pp. 581-587, August 1997.
9. Y.A. Korilis, A.A. Lazar, A. Orda: Achieving network optima using stackelberg routing strategies, *In Proc. IEEE/ACM Transactions of Networking*, 1997.
10. Y. A. Korilis, A. A. Lazar and A. Orda. The designer's perspective to noncooperative networks. *In Proc. IEEE INFOCOM 95.*
11. Y.A. Korilis, A.A. Lazar, A. Orda: Capacity allocation under noncooperative routing, *In Proc. IEEE/Transactions on Automatic Control*, 1997.
12. E. Koutsoupias and C. Papadimitriou. Worst-case Equilibria. *In Proc. STACS '99*, pp. 387–396.
13. H. Lin, T. Roughgarden, and E. Tardos, A Stronger Bound on Braess's Paradox, SODA 2004.
14. M. Mavronikolas and P. Spirakis: The price of Selfish Routing, *In Proc. STOC' 01.*
15. R. B. Myerson. *Game Theory: Analysis of Conflict.* Harvard University Press, 1991
16. N. Nisan and A. Ronen. Algorithmic mechanism design.*In Proc. STOC'99*, pp 129-140
17. N. Nisan. Algorithms for selfish agents: Mechanism design for distributed computation. *In Proc. STACS '99.*
18. M. J. Osborne, A. Rubinstein. *A course in Game Theory*, MIT Press
19. G. Owen. *Game Theory.* Academic Press. Orlando, FL, 3rd Edition, 1995.
20. C. Papadimitriou. Game Theory and Mathematical Economics: A Theoratical Computer Scientist's Introduction. *In Proc. FOCS '01.*
21. C. Papadimitriou. Algorithms, Games, and the Internet. *In Proc. STOC '01*, pp 749-753.
22. T. Roughgarden. The price of anarchy is independent of the network topology. *In Proc. STOC '02*, pp 428-437.
23. T. Roughgarden. Stackelberg scheduling strategies. *In Proc. STOC '01*, pp 104-113.
24. T. Roughgarden, E. Tardos. How bad is Selfish Routing?. *In Proc. FOCS '00*,pp 93-102.
25. T. Roughgarden. Designing networks for selfish users is hard. *In Proc. FOCS '01*, pp 472–481.
26. H. von Stackelberg. *Marktform aund Gleichgewicht.* Spiringer-Verlag, 1934
27. H. Yaiche, R. Mazumdar and C. Rosenberg. Distributed algorithms for fair bandwidth allocation in broadband networks, *In Proc. INFOCOM 00.*
28. H. Yaiche , R. Mazumdar and C. Rosenberg. A game-theoretic framework for bandwidth allocation and pricing in broadband networks, *In Proc. the IEEE/ACM Transactions on Networking*, Vol. 8, No. 5, Oct. 2000, pp. 667-678.
29. *Optimization Toolbox for use with MATLAB*, User's Guide, MathWorks.
30. http://students.ceid.upatras.gr/∼ politop/stackTop, Stackelberg Strategies Toolbox.

# An Improved Branch-and-Bound Algorithm for the Test Cover Problem*

Torsten Fahle and Karsten Tiemann**

Faculty of Computer Science,
Electrical Engineering and Mathematics,
University of Paderborn, Fürstenallee 11,
33102 Paderborn, Germany
{tef, tiemann}@uni-paderborn.de

**Abstract.** The test cover problem asks for the minimal number of tests needed to uniquely identify a disease, infection, etc. At ESA'02 a collection of branch-and-bound algorithms was proposed by [4]. Based on their work, we introduce several improvements that are compatible with all techniques described in [4]. We present a faster data structure, cost based variable fixing and adapt an upper bound heuristic. The resulting algorithm solves benchmark instances up to 10 times faster than the former approach and up to 100 times faster than a general MIP-solver.

## 1 Introduction

We are given a set of $m$ items and a set of tests $\mathbf{T} = \{T_1, \ldots, T_n\}$, $T_k \subseteq \{1, \ldots, m\}$, $k = 1, \ldots, n$. A test $T_k$ is positive for an item $i$ if $i \in T_k$, and negative if $i \notin T_k$. In general, we must use different tests to uniquely identify a given item because a single test can be positive for several items. We say that a test $T_k$ *separates a pair of items* $\{i, j\}$, $1 \leq i < j \leq m$, if $|T_k \cap \{i, j\}| = 1$. Finally, a collection of tests $\mathcal{T} \subseteq \mathbf{T}$ is a *valid cover* if $\forall\ 1 \leq i < j \leq m : \exists T_k \in \mathcal{T} : |T_k \cap \{i, j\}| = 1$.

The *test cover problem (TCP)* asks for a valid cover $\mathcal{T} \subseteq \mathbf{T}$ that is minimal among all valid covers. I.e. for all valid covers $\mathcal{T}'$ it holds $|\mathcal{T}| \leq |\mathcal{T}'|$. The *weighted test cover problem* is a canonical extension of the test cover problem: Given $c : \mathbf{T} \to \mathbb{N}$, where $c(T)$ represents the cost for test $T$, we look for a valid cover $\mathcal{T}$ that is cheapest among all valid covers $\mathcal{T}'$: $\sum_{T \in \mathcal{T}} c(T) \leq \sum_{T \in \mathcal{T}'} c(T)$.

Test cover problems are important in many real-life applications like medical diagnostics, biology, pharmacy, fault detection, or coding theory (see [4]).

The test cover problem has been studied by several authors in recent years. It is known to be NP-hard [3] and approximation algorithms [5] as well as exact branch-and-bound approaches [4, 7] have been proposed. The authors of [4] compared the ef-

* This work was partly supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

** International Graduate School of Dynamic Intelligent Systems.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 89–100, 2005.
© Springer-Verlag Berlin Heidelberg 2005

fectiveness of several pruning criteria, lower bounds and fixing techniques for their algorithm solving unweighted TCPs.

In this paper, we present and analyze improvements on the approach of De Bontridder et al. [4]. All techniques are compatible with the weighted case as well:

*(i)* We use a much faster data structure for the branch-and-bound algorithm. All techniques discussed in [4] benefit from this efficient data structure and numerical results show a typical speedup of 2 – 10 compared to solving the problem with the straightforward data structure.
*(ii)* We introduce cost based variable fixing techniques. That is, during tree search we try to identify tests $T_k$ that have to be (cannot be) part of any improving solution. Those tests are included (excluded) without branching. In so doing, runtime is reduced by a factor of 2 – 4 and the number of branch-and-bound nodes decreases by a factor of up to 5.
*(iii)* Finally, we improve convergence of the branch-and-bound algorithm by using an upper bound. Having a good upper bound before branch-and-bound impacts on variable fixing.

Items *(i)* and *(iii)* are based on a simple observation: Any test cover instance having $m$ items and $n$ tests can be transformed into a set covering instance having $O(m^2)$ elements and $n$ subsets. We investigate on that relation in the next section.

### 1.1 Relation Between Test Cover and Set Covering Problems

A *set covering problem (SCP)* consists of $n$ subsets $\{S_1, \ldots, S_n\} = \mathbf{S}$, $S_i \subseteq \{1, \ldots, \ell\}$. Furthermore, there is a cost function $c : \mathbf{S} \to \mathbb{N}$, where $c(S_i)$ represents the cost of the subset $S_i$. The SCP asks for a collection $\mathcal{S} \subseteq \mathbf{S}$ such that $\{1, \ldots, \ell\}$ is covered at minimal cost. To be more precise:

1. $\bigcup_{S \in \mathcal{S}} S = \{1, \ldots, \ell\}$ and
2. For all $\mathcal{S}'$ such that $\bigcup_{S \in \mathcal{S}'} S = \{1, \ldots, \ell\}$ we have $\sum_{S \in \mathcal{S}} c(S) \leq \sum_{S \in \mathcal{S}'} c(S)$.

SCPs are NP-hard. Given their importance in flight scheduling, crew rostering, etc. they have been subject of intensive research during the last decades, see the survey in [2].

A TCP instance can be interpreted as a SCP instance by considering all pairs of items and asking for a coverage of all pairs by the tests. Let $(m, \mathbf{T} = \{T_1, \ldots, T_n\}, c)$ be a TCP instance. A SCP instance $(\ell, \mathbf{S} = \{S_1, \ldots, S_n\}, c')$ is constructed by

*(i)* using all pairs of items: $\{1, \ldots, \ell\} \leftarrow \{e_{ij} \mid 1 \leq i < j \leq m\}$
*(ii)* setting $S_k$ to contain all pairs of items separated by test $T_k$: $S_k \leftarrow \{e_{ij} \mid |T_k \cap \{i, j\}| = 1,\ 1 \leq i < j \leq m\}$, $k = 1, \ldots, n$, and by
*(iii)* keeping the original costs by: $c'(S_k) := c(T_k),\ \forall\, k \in \{1, \ldots, n\}$.

It is easy to see that a SCP solution $\mathcal{S} \subseteq \mathbf{S}$ defines a solution $\mathcal{T} \subseteq \mathbf{T}$ for the TCP. By construction, the objective value of both solutions is identical. Thus, we can solve a TCP with $n$ tests and $m$ items by solving the corresponding SCP having $n$ subsets and $\Theta(m^2)$ items.

### 1.2 Organization of the Paper

We start our discussion by presenting the basic branch-and-bound approaches of [4] in Sect. 2. After that, we show how to adapt it so as to solve TCPs which are described by SCPs. In Sect. 3 we present variable fixing techniques. Further improvements are obtained by using an upper bound introduced in Sect. 4. In Sect. 5, we compare the runtimes of CPLEX, the old and the new branch-and-bound approach. A discussion of the impact of cost based variable fixing techniques closes this section. Finally, we conclude.

## 2 Basic Branch-and-Bound Approaches

In this section we describe the basic branch-and-bound approaches used in order to solve TCPs. We first briefly recall the approach used by De Bontridder et al. [4] which works on the original TCP formulation. Thereafter, we describe how to adapt that approach to handle the SCP formulation of the TCP. Interestingly, all methods used in the direct (TCP) approach can easily be adapted to work also in the indirect (SCP) one. This allows to apply all structural methods known for the TCP also in the indirect approach.

### 2.1 Solving the Test Cover Problem Directly

In [4] various lower bounds and pruning criteria, as well as variable ordering heuristics for the TCP were described. Furthermore, a branch-and-bound framework was presented that allows to use any combination of a lower bound, a pruning criterion and a variable ordering heuristic.

Each branch-and-bound node is assigned a tuple $(\mathcal{T}, \mathcal{R})$. $\mathcal{T} \subseteq \mathbf{T}$ contains those tests that are included in the solution currently under construction. $\mathcal{R} \subseteq \mathbf{T}$ contains tests that have been discarded from the current solution. Thus, $\mathcal{T} \cap \mathcal{R} = \emptyset$. Initially, it holds $\mathcal{R} = \emptyset$, $\mathcal{T} = \emptyset$. Branch-and-bound decides on the remaining tests $T \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$.

A branch-and-bound node $(\mathcal{T}, \mathcal{R})$ potentially generates $l := |\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})|$ child nodes. These nodes are ordered by some heuristic such that the most promising nodes (those with high quality) are processed first. After renumbering we obtain the child nodes: $(\mathcal{T} \cup \{T_1\}, \mathcal{R})$, $(\mathcal{T} \cup \{T_2\}, \mathcal{R} \cup \{T_1\})$, $(\mathcal{T} \cup \{T_3\}, \mathcal{R} \cup \{T_1, T_2\})$, ..., $(\mathcal{T} \cup \{T_l\}, \mathcal{R} \cup \{T_1, ..., T_{l-1}\})$.

Nodes are considered in a depth-first search, i.e. node $(\mathcal{T} \cup \{T_1\}, \mathcal{R})$ and all its successors are explored prior to node $(\mathcal{T} \cup \{T_2\}, \mathcal{R} \cup \{T_1\})$.

**Quality Criteria.** In [4] four quality criteria are described: Separation criterion $D$, information criterion $\Delta E$, power criterion $P$, and least separated pair criterion $S$. We introduce two of them in more details here and give a simple adaption to the weighted case. Before that, define the subsets $I_1^{\mathcal{T}}, I_2^{\mathcal{T}}, \ldots, I_t^{\mathcal{T}} \subset \{1, \ldots, m\}$ (introduced in [4]). Given a partial cover $\mathcal{T} \subset \mathbf{T}$, a subset $I_j^{\mathcal{T}}$ is a largest subset containing items that are pairwise not separated by the current partial solution $\mathcal{T}$.

The separation criterion $D(T|\mathcal{T})$ was introduced in [7]. It calculates the cost per additional separated pair of items that test $T$ causes when it is added to $\mathcal{T}$ as

$$D(T|\mathcal{T}) := \frac{1}{c(T)} \sum_{h=1}^{t} |T \cap I_h^{\mathcal{T}}| \cdot |I_h^{\mathcal{T}} \setminus T|$$

The least separated pair criterion $S$ was used e.g. in [7, 8, 9, 10]. First, we search for a pair of items $\{i, j\}$ that is covered by least remaining tests. Then we separate the remaining tests into those covering the item pair, and those not covering it by:

$$S(T|\mathcal{T}) := \begin{cases} 1 & \text{if } |T \cap \{i,j\}| = 1 \\ 0 & \text{else} \end{cases}$$

Criterion $S$ can be used alone and follows the idea of assigning most critical items early in the search. In our work, as well as in [4] $S$ is only used in combination with some other criterion (denoted as $S(C)$, where $C \in \{D, \Delta E, P\}$). There, all $T \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ are grouped according to $S$: $\{T \mid S(T|\mathcal{T}) = 1\}$ and $\{T \mid S(T|\mathcal{T}) = 0\}$. Within each of these groups, tests are sorted according to criterion $C$.

**Lower Bounds.** We use two different lower bounds $L_1$ and $L_2$ for pruning. The first one is based on the property that at least $\lceil \log_2 k \rceil$ tests are necessary to separate $k$ items that have not been separated so far. This results in the lower bound

$$L_1(\mathcal{T}) := \lceil \log_2(\max_{h=1,\ldots,t} |I_h^{\mathcal{T}}|) \rceil$$

Lower bound $L_2$ requires the use of the power criterion and some combinatorial values that can be found by a recursive algorithm. Because of space limitations and since this work is on algorithmic improvements we have to refer to [4] for more details on $L_2$. For weighted instances, we can transform a bound $L_q$, $q \in \{1, 2\}$ into a lower bound $L_q^w$, $q \in \{1, 2\}$ for the weighted problem by summing up the $L_q(\mathcal{T})$ smallest cost values of tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$.

**Pruning.** Pruning, that is detecting and discarding useless parts of the search tree is based on four different criteria. Let $L$ be a lower bound on the additional number of tests needed to cover all pairs of items (in addition to the selected tests in $\mathcal{T}$). We prune

**PC1** if according to the lower bound more tests are needed than are available, i.e.

$$\text{if } L > |\mathbf{T}| - |\mathcal{T}| - |\mathcal{R}| \text{ then prune}$$

**PC2** if the minimal number of tests needed to construct a solution is not smaller than the number of tests $U$ in the best solution found so far, i.e.

$$\text{if } |\mathcal{T}| + L \geq U \text{ then prune}$$

**PC2w** (PC2 for weighted case, $U$ is objective value of the best solution found so far)

$$\text{if } \sum_{T \in \mathcal{T}} c(T) + \sum_{i=1}^{L} c(T_i) \geq U \text{ then prune}$$

(PC2w requires the remaining tests $T_i \in \mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ to be sorted according to increasing costs. Such an ordering can be determined in a preprocessing step.)

**PC3** if there is an uncovered pair of items that cannot be covered by any of the remaining tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$.

### 2.2 Solving the Test Cover Problem Indirectly

We propose to solve the test cover problem indirectly. I.e. instead of working on the original formulation, we transform the TCP into a SCP (Sect. 1.1).

This transformation squares the number of "objects" in the problem, because rather than separating $m$ items in the TCP we have to cover $\Theta(m^2)$ pairs of items in the SCP. On the other hand, most techniques described before have to spend time $\Omega(m^2)$ anyway (e.g. they need to find out which pairs of items are not covered, etc.). That is, the direct formulation is more space efficient, but does not save computing time. Even worse, whereas in the SCP formulation we can delete a pair of items from consideration as soon as we have covered it, we cannot do similarly in the original formulation.

There, pairs of items are checked sequentially and an item $i$ can only be removed from the list of active items, when it is separated from all other items $j$. In the best case this happens after $m - i$ steps (after separating $\{i, i+1\}, \ldots, \{i, m\}$ we can discard item $i$). In the worst case, however, $\frac{1}{2}(m^2 - 2m) + 1$ steps are necessary before in the direct formulation a test can be discarded from consideration (see Fig. 1).

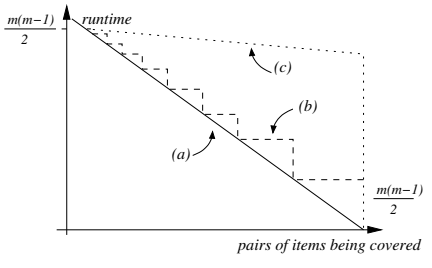



**Fig. 1.** Schematic view on the runtime ($y$-axis) when working on a branch-and-bound node with a certain number of separated pairs of items ($x$-axis). (a) is the runtime for the indirect approach – runtime is proportional to the number of pairs of items. (b) and (c) give examples for best and worst case runtimes when using the direct formulation of the problem

Interpreting a TCP as a SCP thus gives a natural way of finding a more efficient data structure for pairs of items.

## 3 Variable Fixing

Variable fixing aims in identifying tests that have to be part of a feasible or optimal solution. It is one of the building blocks in constraint programming and it is frequently used in presolving for mixed integer programming. We first explain in Sect. 3.1 a fixing technique also used in [4]. In Sect. 3.2 we introduce a new variable fixing method based on costs that can reduce runtime by a factor of 2 – 4 (see Sect. 5.3).

### 3.1 Fixings Based on Feasibility

Before starting a branch-and-bound algorithm, it is helpful to identify *essential tests*. A test $T \in \mathbf{T}$ is essential if there is a pair $\{i, j\}$ of items that is only separated by $T$. In this case $T$ must be part of any feasible solution to the TCP and it can be added to $\mathcal{T}$. As described in [4] searching for essential tests can be performed in time $O(m^2 n)$.

In our approach we check for essential tests in all branch-and-bound nodes. During branching some other test as well might become the only remaining test that covers a certain pair of items. It is thus justified to check for essential tests not only in the root node. Using an incremental algorithm we can do so in time proportional to the number of pairs not considered so far: Let $\mu_{\{i,j\}}$ denote the number of tests that cover item pair $\{i, j\}$. In an initial preprocessing step we initialize $\mu_{\{i,j\}}$ in time $O(m^2 n)$. For any branching decision or any fixing of tests, we decrement those entries of $\mu$ that correspond to pairs of items covered by the tests in question. Hence, in each branch-and-bound node we can simply check whether only one test remains that covers a certain pair of items, and we can include that test into the current solution. The number of pairs of items not considered decreases with the depth of the search tree and is always in $O(m^2)$. Thus, fixing essential tests in a branch-and-bound node requires time $O(m^2)$ per node which is the time already needed by most other techniques described in Sect. 2.

### 3.2 Fixings Based on Costs

Additionally, we can include or exclude tests that have to be or cannot be part of an improving solution. Let $L$ denote a lower bound on the number of tests required to form a valid test cover in the current branch-and-bound node, and let $U$ denote the value of the incumbent upper bound. (Both values are known in each node because we already calculate them for bounding.) We order the remaining $k := |\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})|$ tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R}) =: \{T_1, \ldots, T_k\}$ according to increasing costs.

If $k < L$ pruning criterion PC1 already prunes the current search tree. We also prune, if the cost of all tests in $\mathcal{T}$ plus the costs of the $L$ cheapest tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ exceeds the current upper bound $U$ (PC2w).

Otherwise, we can apply the following variable fixing: If replacing one of the $L$ cheapest tests $T_i$ by test $T_{L+1}$ results in a value not smaller than the incumbent solution, test $T_i$ is essential for any improving solution, and thus we can fix it:

$$\begin{array}{l} \forall\, 1 \le i \le L: \\ \quad \text{if} \quad \left(\sum_{T\in\mathcal{T}} c(T) + \sum_{j=1}^{L} c(T_j) - c(T_i) + c(T_{L+1})\right) \ge U \\ \quad \text{then} \quad \mathcal{T} := \mathcal{T} \cup \{T_i\} \end{array} \tag{1}$$

Vice versa, if already replacing test $T_L$ by some more expensive test results in a value not smaller than the incumbent solution, we can safely discard the latter test:

$$\begin{array}{l} \forall\, L+1 \le i \le k: \\ \quad \text{if} \quad \left(\sum_{T\in\mathcal{T}} c(T) + \sum_{j=1}^{L} c(T_j) - c(T_L) + c(T_i)\right) \ge U \\ \quad \text{then} \quad \mathcal{R} := \mathcal{R} \cup \{T_i\} \end{array} \tag{2}$$

Notice that these checks are only useful for the weighted case as in the unweighted case no change in the cost structure occurs when exchanging tests.

## 4 Upper Bound Computation

When interpreting a test cover problem as a set covering problem, upper bound heuristics can be adapted to the test cover problem. The two-phase approach of Beasley [1] was adapted for our work (Alg. 1). It starts with a Lagrangian lower bound $x$ and covers all missing item pairs by a cheapest subset. If after that phase some pairs of items are over-covered it tries to remove those subsets that are redundant. In some experiments we call that upper bound heuristic initially prior to any branch-and-bound.

**Algorithm 1** Constructing an upper bound $x'$ from a lower bound $x$ (see [1])

$x' \leftarrow x; \quad \mathcal{S}' \leftarrow \{S_k \mid x_k = 1,\ k = 1, \ldots, n\}$

*/* Phase 1: Cover all items by including missing sets */*
**while** ($\exists$ a pair of items $\{i, j\}$ that is not covered by $\mathcal{S}'$)
  $l \leftarrow \text{index}(\text{argmin}\{c(S_k) \mid S_k \text{ covers } \{i, j\},\ k = 1, \ldots, n\})$
  $x'_l \leftarrow 1; \quad \mathcal{S}' \leftarrow \mathcal{S}' \cup \{S_l\}$

*/* Phase 2: If pairs are over-covered: Remove redundant sets */*
**for all** ($S_k \in \mathcal{S}'$ in order of decreasing costs)
  **if** ($\mathcal{S}' \setminus \{S_k\}$ covers all pairs $\{i, j\}$)
    **then** $x'_k \leftarrow 0; \quad \mathcal{S}' \leftarrow \mathcal{S}' \setminus \{S_k\}$
**return** $x'$

## 5 Numerical Results

In [4] 140 benchmark sets were used to experimentally evaluate different branch-and-bound algorithms for the TCP. These benchmark sets were constructed randomly, and they differ with respect to the number of items $m$, the number of tests $n$, and the probability $p$ for a test to contain a certain item ($\mathrm{E}[i \in T_k] = p$). There are 10 different instances for each of the following $(m, n, p)$-combinations: (49, 25, {1/4, 1/2}), (24, 50, {1/10, 1/4, 1/2}), (49, 50, {1/10, 1/4, 1/2}), (99, 50, {1/10, 1/4, 1/2}), (49, 100, {1/10, 1/4, 1/2}). We use the same sets for the unweighted case and thus our results can be compared to those found in the earlier work on the TCP.

For the weighted case these instances were extended by assigning a cost value to each test uniformly at random from the interval $\{1, \ldots, n\}$.

All tests were performed on a Pentium 4 (1.7 GHz) running Linux 2.4.19. The algorithms were coded in C++ and compiled by gcc 2.95.3 using full optimization. In the comparison we used Ilog CPLEX 7.5 with default settings.

In their paper De Bontridder et al. note that they have not used *"clever data structures for storing and retrieving information"* but that a *"significant speedup"* could be expected from these. Therefore, in addition to the techniques used in [4] both approaches (direct and indirect) store information needed in every branch-and-bound node. We update this information incrementally rather than calculating it from scratch in each node. These data include the assignment of items to the sets $I_1^{\mathcal{T}}, I_2^{\mathcal{T}}, ..., I_t^{\mathcal{T}}$ which are needed for branching based on quality criterion $D, \Delta E, P$ as well as for lower bounds $L_1$ and $L_2$. Furthermore, for each pair of items $\{i, j\}$ not covered so far,

we store the number of tests in $\mathbf{T} \setminus (\mathcal{T} \cup \mathcal{R})$ that cover $\{i, j\}$. The latter information is needed for fixing essential tests, for the least-separated pair criterion $S$ and for PC3. Because the implementation in [4] always re-calculates the just mentioned values, it is justified to assume that our implementation of the direct approach is already faster than the original implementation used by the authors of [4].

In the following sections we compare three different approaches to solve test cover problems, namely (a) solving the TCP directly, (b) solving the TCP indirectly by transforming it to a SCP (see Sect. 1.1), and (c) solving its SCP formulation via CPLEX. In Sect. 5.3 we elaborate more on the impact of the new cost fixing and of the upper bound heuristic. All figures show average runtimes in seconds and average numbers of branch-and-bound nodes, respectively, for the 10 instances in each $(m, n, p)$-combination.

### 5.1 Unweighted Benchmark Sets

We have studied different combinations of pruning criteria, branching rules and quality orderings for both branch-and-bound approaches. For the direct approach a combination of quality criterion $S(D)$ and lower bound $L_1$ or $L_2$ (noted as $(S(D), L_1)$, $(S(D), L_2)$) was found to be most efficient (this corresponds to the findings in [4]). For the indirect approach using $L_1$ rather than $L_2$ turned out to be most efficient. Thus we use the variant $(S(D), L_1)$ in most experiments. For a detailed study on the impact of different pruning criteria, branching rules and quality criteria we refer to the work in [4] for the direct approach and to [11] for the indirect approach. We apply essential tests (Sect. 3.1) in every branch-and-bound node but do not compute an upper bound with the technique described in Sect. 4.

Figure 2 shows that both branch-and-bound approaches are between 10 – 100 times faster than CPLEX. For instances having 49 items and 100 tests only one third of all CPLEX runs terminated within 24 hours.

In most cases the indirect approach is about 4 times faster than the direct approach.

### 5.2 Weighted Benchmark Sets

On weighted benchmark sets we use pruning criteria PC1, PC2w and PC3. We do not compute an upper bound based on a Lagrangian lower bound but we apply cost fixing in every branch-and-bound node (see Sect. 5.3). For the direct approaches $(S(D), L_2)$ or $(S(P), L_2)$ are the most efficient approaches. Replacing $L_2$ by $L_1$ does only slightly reduce runtime for the smaller instances. On larger instances $(49, 100, 1/10)$, however, a factor of 2 is lost when using $L_1$ rather than $L_2$.

For the indirect approach using $(S(D), L_2)$ or $(S(P), L_2)$ is the best choice. On smaller instances these two settings are about three times faster than other combinations of branching or ordering strategies. Interestingly, Fig. 3 shows that on the 10 instances in class $(49, 100, 1/10)$ CPLEX is able to outdo both the direct and the indirect approach by a factor of 10 or 20, respectively, whereas on all other classes (130 instances), the specialized algorithms prevail by up to a factor of 10.

The indirect approach is always faster than the direct one (factor 2 – 10).

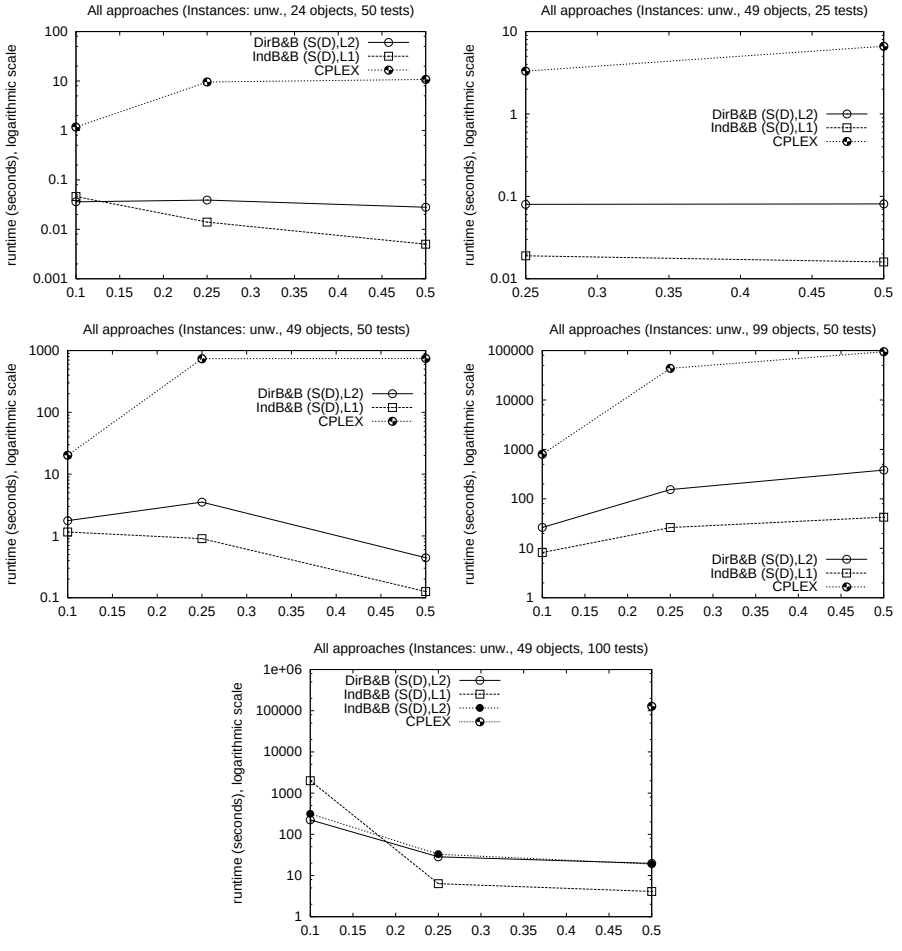



**Fig. 2.** Comparing direct approach, indirect approach and CPLEX on several instances. Notice the logarithmic scale. Lines connecting measure points are used for better readability and do not indicate intermediate results

### 5.3 Impact of Cost Fixing and Upper Bound

Cost fixing as described in Sect. 3.2 reduces runtime as well as number of branching decisions for almost all instances. As can be seen in Fig. 4 cost fixing reduces runtime of instances (49, 50, {1/10, 1/4, 1/2}) and (99, 50, {1/10, 1/4, 1/2}) by a factor of 2 – 3, whereas the number of branch-and-bound nodes is reduced by a factor of 3 – 5.

Also for the indirect approach, using cost fixing impacts positively on runtime as well as on number of branch-and-bound nodes. Only between 25% – 75% of the original runtime is needed when using cost fixing (see Fig. 5). The number of branch-and-bound nodes is only 1/6 – 1/2 of the number of nodes when not using cost fixing. Impact of cost fixing diminishes the more items are covered by a test. On the other hand, the number of

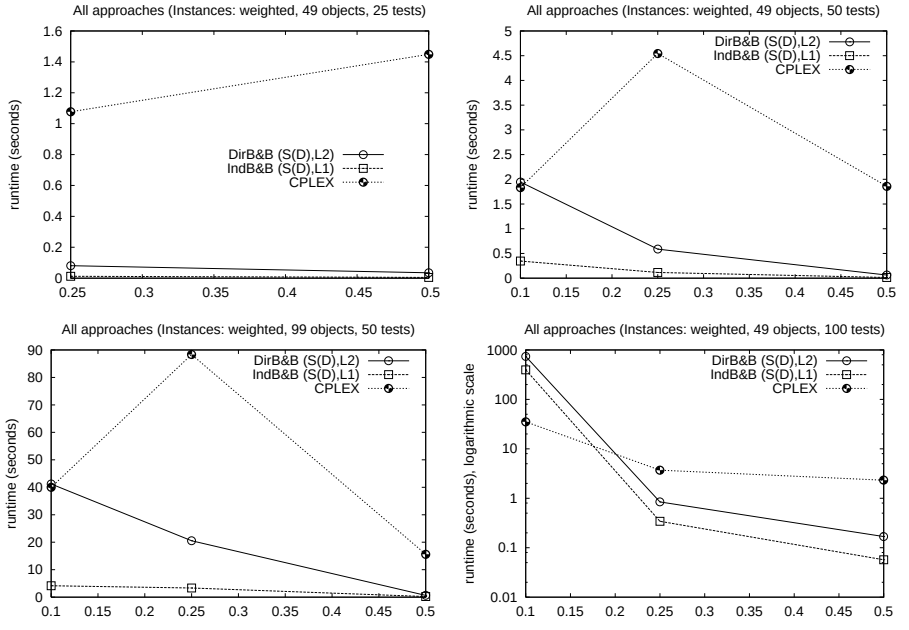



**Fig. 3.** Comparing the approaches on weighted benchmark sets

branch-and-bound nodes is very small already in that scenario. Experiments on larger instances that require more branching decisions are needed in order to conclude on the behavior of instances having many items per test.

Finding an upper bound initial to branch-and-bound as explained in Sect. 4 should be helpful for fixing variables. In our experiments it turns out that the upper bound heuristic typically reduces runtime by about 1% – 5%. Upper bounds thus are the least important ingredient for a fast solver to the test cover problem, although using them reduces runtime and number of branching nodes.

## 6 Conclusions

In this paper we presented a simple, yet fast data structure for test cover problems. Experimental results show a typical speedup of up to a factor of 10 compared to an implementation proposed earlier. Furthermore, we introduced cost based filtering for tests. These techniques turned out to be quite helpful because they reduce the number of explored branch-and-bound nodes by a factor of 5 and running time by a factor of 2 – 4.

It is known that Lagrangian techniques are quite successful for set covering problems (see [1, 2]). We performed initial experiments on applying Lagrangian bounds in all nodes of the branch-and-bound tree. This modification led to a significant decrease in the number of branch-and-bound nodes. Faster bound calculations are needed, however, to turn this decrease in tree size into a runtime improvement. Future work will thus include investigation on Lagrangian techniques as well as experiments on larger problem instances.

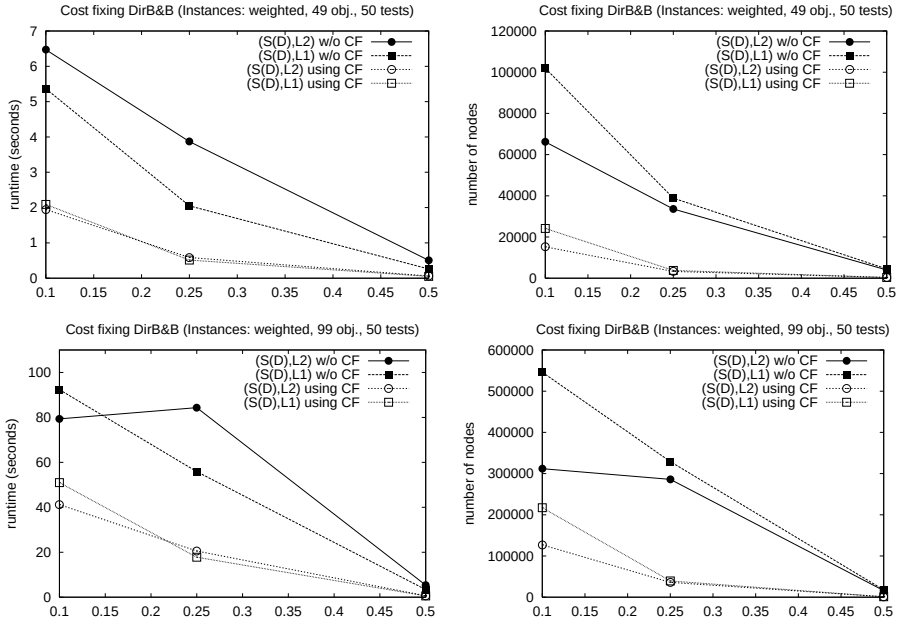



**Fig. 4.** Runtimes (left) and number of branch-and-bound nodes (right) for the direct approach DirB&B using / not using cost fixing (CF) on instances containing 49 (top) and 99 (bottom) items and 50 tests, respectively

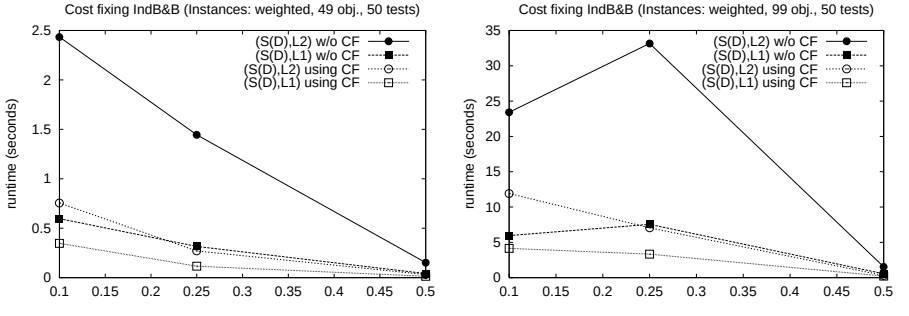



**Fig. 5.** The indirect approach IndB&B using / not using cost fixing (CF) on instances containing 50 tests and 49 (left) or 99 (right) items, respectively

## Acknowledgment

We would like to thank K.M.J. De Bontridder who provided us with the original benchmark sets used in [4].

## References


1. J.E. Beasley. A Lagrangian Heuristic for Set-Covering Problems. *Naval Research Logistics*, 37:151–164, 1990.
2. A. Caprara, M. Fischetti, P. Toth. Algorithms for the Set Covering Problem. *Annals of Operations Research*, 98:353–371, 2001.
3. M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Co., 1979.
4. K.M.J. De Bontridder, B.J. Lageweg, J.K. Lenstra, J.B. Orlin, L. Stougie. Branch-and-Bound Algorithms for the Test Cover Problem. *Proceedings of ESA 2002*, Rome/Italy. Springer LNCS 2461, pp. 223–233, 2002.
5. B.V. Halldórsson, M.M. Halldórsson, R. Ravi. On the Approximability of the Minimum Test Collection Problem. *Proceedings of ESA 2001*, Århus/Denmark. Springer LNCS 2161, pp. 158–169, 2001.
6. M. Held and R.M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees: Part II. *Mathematical Programming*, 1:6–25, 1971.
7. B. Moret and H. Shapiro. On Minimizing a Set of Tests. *SIAM Journal on Scientific and Statistical Computing*, 6:983–1003, 1985.
8. R.W. Payne. Selection Criteria for the Construction of Efficient Diagnostic Keys. *Journal of Statistical Planning and Information*, 5:27–36, 1981.
9. E.W. Rypka, W.E. Clapper, I.G. Brown, R. Babb. A Model for the Identification of Bacteria. *Journal of General Microbiology*, 46:407–424, 1967.
10. E.W. Rypka, L. Volkman, E. Kinter. Construction and Use of an Optimized Identification Scheme. *Laboratory Magazine*, 9:32–41, 1978.
11. K. Tiemann. Ein erweiterter Branch-and-Bound-Algorithmus für das Test-Cover Problem. Bachelor-Thesis (in German), University of Paderborn, 2002.

# Degree-Based Treewidth Lower Bounds*

Arie M.C.A. Koster[1], Thomas Wolle[2], and Hans L. Bodlaender[2]

[1] Zuse Institute Berlin (ZIB),
Takustraße 7, D-14194 Berlin, Germany
koster@zib.de
[2] Institute of Information and Computing Sciences,
Utrecht University P.O. Box 80.089,
3508 TB Utrecht, The Netherlands
{thomasw, hansb}@cs.uu.nl

**Abstract.** Every lower bound for treewidth can be extended by taking the maximum of the lower bound over all subgraphs or minors. This extension is shown to be a very vital idea for improving treewidth lower bounds. In this paper, we investigate a total of nine graph parameters, providing lower bounds for treewidth. The parameters have in common that they all are the vertex-degree of some vertex in a subgraph or minor of the input graph. We show relations between these graph parameters and study their computational complexity. To allow a practical comparison of the bounds, we developed heuristic algorithms for those parameters that are $NP$-hard to compute. Computational experiments show that combining the treewidth lower bounds with minors can considerably improve the lower bounds.

## 1 Introduction

Many combinatorial optimisation problems take a graph as part of the input. If this graph belongs to a specific class of graphs, typically more efficient algorithms are available to solve the problem, compared to the general case. In case of trees for example, many $NP$-hard optimisation problems can be solved in polynomial time. Over the last decades, it has been shown that many $NP$-hard combinatorial problems can be solved in polynomial time for graphs with treewidth bounded by a constant. Until recently, it was assumed that these results were of theoretical interest only. By means of the computation of so-called exact inference in probabilistic networks [17] as well as the frequency assignment problem [15] in cellular wireless networks, it has been shown that such an algorithm to compute the optimal solution can be used in practice as well.

* This work was partially supported by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/9-3, GR 883/9-4), and partially by the Netherlands Organisation for Scientific Research NWO (project *Treewidth and Combinatorial Optimisation*).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 101–112, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Polynomial time algorithms for solving combinatorial problems on a graph of bounded treewidth consist of two steps: (i) the construction of a tree decomposition of the graph with width as small as possible, and (ii) the application of dynamic programming on the tree decomposition to find the optimal solution of the combinatorial problem. Whereas the first step can be applied without knowledge of the application, the second step requires the development of an algorithm tailor-made for the specific application.

To exploit the full potential of tree decomposition approaches for as many combinatorial problems as possible, the first step is of fundamental importance. The smallest possible width of a tree decomposition is known as the treewidth of the graph. Computing the treewidth is however $NP$-hard [1]. To advance towards tree decompositions with close-to-optimal width, research in recent years has been carried out on practical algorithms for reduction and decomposition of the input graph [5, 6, 11], upper bounds [10, 9, 14], lower bounds [4, 7, 10, 18, 20], and exact algorithms (e.g. [12]).

In this paper, we research treewidth lower bounds that are based on the degree of specific vertices. Good treewidth lower bounds can be utilised to decrease the running time of branch-and-bound algorithms (see e.g. [12]). The better the lower bounds, the bigger the branches that can be pruned in a branch-and-bound method. Furthermore, treewidth lower bounds are useful to estimate the running times of dynamic programming methods that are based on tree decompositions. Such methods have running times that are typically exponential in the treewidth. Therefore, a large lower bound on the treewidth of a graph implies only little hope for an efficient dynamic programming algorithm based on a tree decomposition of that graph. In addition, lower bounds in connection with upper bounds help to assess the quality of these bounds.

Every lower bound for treewidth can be modified by taking the maximum of the lower bound over all subgraphs or minors. In [7, 8] this idea was used to obtain considerable improvements on two lower bounds: the minimum degree of a graph and the MCSLB lower bound by Lucena [18].

In this paper, we extend our research efforts to improve the quality of further known lower bounds in this way. One lower bound for treewidth is given by the second smallest degree, another one by the minimum over all non-adjacent pairs of vertices of the maximum degree of the vertices (cf. Ramachandramurthi [20]). Altogether, we examine nine parameters (defined in Section 2) and determine some relationships between them (see Section 3.1). We show that the second smallest degree over all subgraphs is computable in polynomial time, whereas the parameters for other combinations are $NP$-hard to compute (see Section 3.2). In this extended abstract, however, we omit full proofs. For the parameters that are $NP$-hard to compute, we develop several algorithms in Section 4.2 to obtain treewidth lower bounds heuristically. A computational evaluation (Section 4.3 and 4.4) of the algorithms shows that the heuristics where we combine a lower bound with edge contraction outperforms other strategies.

## 2 Preliminaries and Graph Parameters

Throughout the paper $G = (V, E)$ denotes a simple undirected graph. Unless otherwise stated, $n(G)$ (or simply $n$) denotes the number of vertices in $G$, i.e. $n := |V|$, and $m(G)$ (or simply $m$) denotes the number of edges $m := |E|$. Most of our terminology is standard graph theory/algorithm terminology. The open neighbourhood $N_G(v)$ or simply $N(v)$ of a vertex $v \in V$ is the set of vertices adjacent to $v$ in $G$. As usual, the degree in $G$ of vertex $v$ is $d_G(v)$ or simply $d(v)$, and we have $d(v) = |N(v)|$. $N(S)$ for $S \subseteq V$ denotes the open neighbourhood of $S$, i.e. $N(S) = \bigcup_{s \in S} N(s) \setminus S$.

**Subgraphs and Minors.** After deleting vertices of a graph and their incident edges, we get an *induced subgraph*. A *subgraph* is obtained, if we additionally allow deletion of edges. (We use $G' \subseteq G$ to denote that $G'$ is a subgraph of $G$.) If we furthermore allow edge-contractions, we get a *minor* (denoted as $G' \preceq G$, if $G'$ is a minor of $G$). Contracting edge $e = \{u, v\}$ in the graph $G = (V, E)$ is the operation that introduces a new vertex $a_e$ and new edges such that $a_e$ is adjacent to all the neighbours of $u$ and $v$, and deletes vertices $u$ and $v$ and all edges incident to $u$ or $v$.

**Treewidth.** The notions treewidth and tree decomposition were introduced by Robertson and Seymour in [21]. A *tree decomposition* of $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$, with $\{X_i \mid i \in I\}$ a family of subsets of $V$ and $T$ a tree, such that each of the following holds: $\bigcup_{i \in I} X_i = V$; for all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$; and for all $i_0, i_1, i_2 \in I$: if $i_1$ is on the path from $i_0$ to $i_2$ in $T$, then $X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$. The *width* of tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\max_{i \in I} |X_i| - 1$. The *treewidth* $tw(G)$ of $G$ is the minimum width among all tree decompositions of $G$. The following lemma is well known and an important fact for proving the parameters, considered in this paper, to be treewidth lower bounds.

**Lemma 1 (see e.g. [3]).** *If $G'$ is a minor of $G$, then $tw(G') \leq tw(G)$.*

**Graph Parameters.** We consider a number of graph parameters in this paper, all lower bounds on the treewidth of a graph, cf. Section 3. The *minimum degree* $\delta$ of a graph $G$ is defined as usual: $\delta(G) := \min_{v \in V} d(v)$

The *$\delta$-degeneracy* or simply the *degeneracy* $\delta D$ of a graph $G$ is defined in [2] to be the minimum number $s$ such that $G$ can be reduced to an empty graph by the successive deletion of vertices with degree at most $s$. It is easy to see that this definition of the degeneracy is equivalent (see [24]) to the following definition: $\delta D(G) := \max_{G'} \{\delta(G') \mid G' \subseteq G \wedge n(G') \geq 1\}$ The treewidth of $G$ is at least its degeneracy (see also [14]). The *$\delta$-contraction degeneracy* or simply the *contraction degeneracy* $\delta C$ of a graph $G$ was first defined in [7]. Instead of deleting a vertex $v$ of minimum degree, we contract it to a neighbour $u$, i.e. we contract the edge $\{u, v\}$. This has been proven to be a very vital idea for obtaining treewidth lower bounds [7, 8]. The contraction degeneracy

is defined as the maximum over all minors $G'$ of $G$ of the minimum degree: $\delta C(G) := \max_{G'}\{\delta(G') \mid G' \preceq G \wedge n(G') \geq 1\}$

Let be given an ordering $v_1, ..., v_n$ of the vertices of $G$ with $n \geq 2$, such that $d(v_i) \leq d(v_{i+1})$, for all $i \in \{1, ..., n-1\}$. The *second smallest degree* $\delta_2$ of a graph $G$ is defined as: $\delta_2(G) := d(v_2)$ Note that it is possible that $\delta(G) = \delta_2(G)$. Similar to the $\delta$-degeneracy and $\delta$-contraction-degeneracy we define the $\delta_2$-degeneracy and $\delta_2$-contraction-degeneracy. The *$\delta_2$-degeneracy* $\delta_2 D$ of a graph $G = (V, E)$ with $n \geq 2$ is defined as follows: $\delta_2 D(G) := \max_{G'}\{\delta_2(G') \mid G' \subseteq G \wedge n(G') \geq 2\}$ The *$\delta_2$-contraction degeneracy* $\delta_2 C$ of a graph $G = (V, E)$ with $n \geq 2$ is: $\delta_2 C(G) := \max_{G'}\{\delta_2(G') \mid G' \preceq G \wedge n(G') \geq 2\}$

In [19, 20], Ramachandramurthi introduced the parameter $\gamma_R(G)$ of a graph $G$ and proved that this is a lower bound on the treewidth of $G$. $\gamma_R(G) := \min(n-1, \min_{v,w \in V, v \neq w, \{v,w\} \notin E} \max(d(v), d(w)))$ Note that $\gamma_R(G) = n - 1$ if and only if $G$ is a complete graph on $n$ vertices. Furthermore, note that $\gamma_R(G)$ is determined by a pair $\{v, w\} \notin E$ with $\max(d(v), d(w))$ is as small as possible. We say that $\{v, w\}$ is a non-edge determining $\gamma_R(G)$, and if $d(v) \leq d(w)$ then we say that $w$ is a vertex determining $\gamma_R(G)$. Once again, we define the 'degeneracy' and 'contraction degeneracy' versions also for the parameter $\gamma_R$. The *$\gamma_R$-degeneracy* $\gamma_R D(G)$ of a graph $G = (V, E)$ with $n \geq 2$ is defined as follows: $\gamma_R D(G) := \max_{G'}\{\gamma_R(G') \mid G' \subseteq G \wedge n(G') \geq 2\}$ The *$\gamma_R$-contraction degeneracy* $\gamma_R C(G)$ of a graph $G = (V, E)$ with $n \geq 2$ is defined as: $\gamma_R C(G) := \max_{G'}\{\gamma_R(G') \mid G' \preceq G \wedge n(G') \geq 2\}$.

## 3 Theoretical Results

### 3.1 Relationships Between the Parameters

**Lemma 2.** *For a graph $G = (V, E)$ with $|V| \geq 2$, $x \in \{\delta, \delta_2, \gamma_R\}$ and $X \in \{D, C\}$, each of the following holds:*

1. $\delta(G) \leq \delta_2(G) \leq \gamma_R(G) \leq tw(G)$
2. $x(G) \leq xD(G) \leq xC(G) \leq tw(G)$
3. $\delta X(G) \leq \delta_2 X(G) \leq \gamma_R X(G) \leq tw(G)$
4. $\delta_2 X(G) \leq \delta X(G) + 1$
5. $\gamma_R X(G) \leq 2 \cdot \delta_2 X(G)$

It follows directly from Lemma 2 that all the parameters defined in Section 2 are lower bounds for treewidth. Furthermore, we see that the gap between the parameters $\delta D$ and $\delta_2 D$, and between $\delta C$ and $\delta_2 C$ can be at most one (see Lemma 2). In Section 3.2, we will see that $\delta_2 D$ can be computed in polynomial time. Therefore, Lemma 2 gives us a 2-approximation algorithm for computing the parameter $\gamma_R D$. Also in Section 3.2, we will see that $\gamma_R D$ is $NP$-hard to compute.

The next lemma shows some interesting properties of the parameter $\gamma_R$, when given a vertex sequence sorted according to non-decreasing degree.

**Lemma 3.** *Let be given a graph $G$ on $n$ vertices with $G \neq K_n$. Furthermore, let be given an ordering $v_1, ..., v_n$ of $V(G)$, such that $d(v_i) \leq d(v_{i+1})$, for all $i \in \{1, ..., n-1\}$. We define $j := \min\{i \in \{1, ..., n\} \mid \exists l \in \{1, ..., i-1\} : \{v_i, v_l\} \notin E(G)\}$. Then we have:*

1. *$d(v_j) = \gamma_R(G)$*
2. *$v_1, ..., v_{j-1}$ form a clique in $G$*

### 3.2 Computational Complexity of the Parameters

#### A Bucket Data Structure

In this section, we briefly describe a data structure that can be used in many of our algorithms. A more detailed description can be found in [24]. We extend the standard adjacency-list data structure of a graph $G = (V, E)$ in the following way. We store in doubly linked lists the adjacent vertices for every vertex of the graph, and we also use cross pointers for each edge $\{v_i, v_j\}$ (i.e. a pointer between vertex $v_i$ in the adjacency-list for vertex $v_j$ and vertex $v_j$ in the adjacency-list for vertex $v_i$). In addition to this *advanced-adjacency-list*, we create $n = |V|$ buckets that can be implemented by doubly-linked lists $B_0, ..., B_{n-1}$. List $B_d$ contains exactly those vertices with degree $d$. We maintain a pointer $p(v)$ for every vertex $v$ that points to the exact position in the list $B_d$ that contains $v$ for the appropriate $d$.

**Lemma 4 (see [24]).** *Let be given a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$. An algorithm performing a sequence of $O(n)$ vertex deletions and searches for a vertex with smallest or second smallest degree can be implemented to use $O(n + m)$ time.*

#### Known Results

It is easy to see that $\delta(G)$ and $\delta_2(G)$ can be computed in $O(n + m)$ time. Also the parameter $\gamma_R(G)$ can be computed in $O(n+m)$ time, see [19] or Section 4.1. Interestingly enough, the definition of the degeneracy as in [2] (see also Section 2) reflects an algorithm to compute this parameter: Successively delete a vertex of minimum degree and return the maximum of the encountered minimum degrees. Using the data structure described in this section, $\delta D(G)$ can be computed in time $O(n + m)$. Computing $\delta C$ is $NP$-hard as is shown in [7].

#### $\delta_2 D$ Is Computable in Polynomial Time

A strategy to compute $\delta_2 D$ is as follows. We can fix a vertex $v$ of which we suppose it will be the vertex of minimum degree in a subgraph $G'$ of $G$ with $\delta_2(G') = \delta_2 D(G)$. Starting with the original graph, we successively delete a vertex in $V(H) \setminus \{v\}$ of smallest degree, where $H$ is the current considered subgraph of $G$ (initially: $H = G$). Since we do not know whether our choice of $v$ was optimal, doing this for all vertices $v \in V$ leads to a correct algorithm to compute $\delta_2 D(G)$. Using the bucket data structure, described above, this method

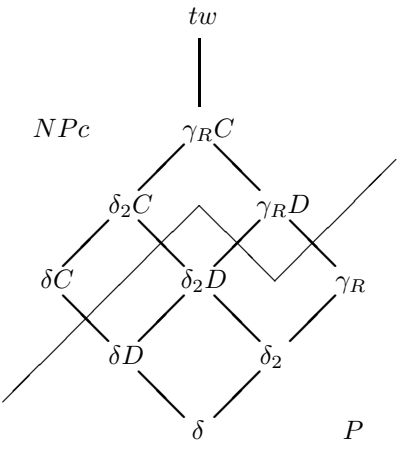



**Fig. 1.** An overview of some theoretical results

can be implemented to take $O(n \cdot m)$ time. We call this algorithm `Delta2D`. The following pseudo-code makes this algorithm more precise.

```
Algorithm Delta2D

 1   delta2D := 0
 2   for each v ∈ V do
 3       H := G
 4       repeat
 5           if δ2(H) > delta2D then delta2D := δ2(H) endif
 6           V* := V(H) \ {v}
 7           let u ∈ {w ∈ V* | ∄w' ∈ V* : dH(w') < dH(w)}
 8           H := H[V(H) \ {u}]
 9       until |V(H)| = 1
10   endfor
11   return delta2D
```

**Lemma 5.** *Algorithm* `Delta2D` *computes* $\delta_2 D(G)$ *and can be implemented to run in* $O(n \cdot m)$ *time, for a given connected graph* $G = (V, E)$ *with* $|V| \geq 2$.

### *NP*-completeness Results

Here, we will state the computational hardness of the decision problems corresponding to the parameters $\gamma_R D$, $\gamma_R C$ and $\delta_2 C$.

**Theorem 1.** *Let* $G$ *be a graph,* $G'$ *be a bipartite graph and* $k$ *be an integer. Each of the following is NP-complete to decide:* $\gamma_R D(G) \geq k$, $\gamma_R C(G') \geq k$ *and* $\delta_2 C(G) \geq k$.

Figure 1 represents some of the theoretical results. A thick line between two parameters indicates that the parameter below is smaller or equal to the

parameter above, as stated by Lemma 2. The thin line marks the border between polynomial computability and $NP$-hardness of the corresponding parameters (see Theorem 1 and other results in Section 3.2).

## 4 Experimental Results

In this section, we describe exact and heuristic algorithms, which we used in our experiments to compute the corresponding parameters.

### 4.1 Exact Algorithms

An implementation of algorithms to compute $\delta$ and $\delta_2$ is straightforward. It is obvious that, in linear time, both parameters can be computed exactly. The parameters $\delta D$ and $\delta_2 D$ were computed as described in Section 3.2. Ramachandramurthi shows in [19] that $\gamma_R$ can be computed in $O(n + m)$ time. In our experiments, we use a different algorithm that does not use an adjacency matrix. See the full version of this article ([16]) for more details.

### 4.2 Heuristics

For the parameters that are $NP$-hard to compute, we have developed heuristics some of which are based on the polynomial counterparts.

*$\gamma_R$-degeneracy:* For the $\gamma_R D$, we developed three heuristics based on the following observation: Let $v_1, ..., v_n$ be a sorted sequence of the vertices according to non-decreasing degree in $G$, and let $\gamma_R(G)$ be determined by $v_j$ for some $j > 1$ (see Lemma 3). Thus, $v_j$ is not adjacent to some vertex $v_k$ with $k < j$, whereas $v_1, ..., v_{j-1}$ induce a clique in $G$. Let $V'$ be the set of all vertices $v_i$ with $i < j$ and $\{v_i, v_j\} \notin E$. Now, for any subgraph $G' \subset G$ with $(\{v_j\} \cup V') \subseteq V(G')$, we have that $\gamma_R(G') \leq \gamma_R(G)$. Hence, only subgraphs without either $v_j$ or $V'$ are of further interest. Based on this observation, we implemented two heuristics. In the heuristic $\gamma_R D$-left, we remove the vertices in $V'$ (the vertices that are more to the left in the sequence) from the graph and continue. Whereas in the heuristic $\gamma_R D$-right, we delete the vertex $v_j$ (the vertex that is more to the right in the sequence) and go to the next iteration.

*$\delta$-contraction degeneracy:* For computing lower bounds for $\delta C$, we have examined various strategies for contraction in [7]. The most promising one has been to recursively contract a vertex of minimum degree with a neighbour that has the least number of common neighbours (denoted as the least-c strategy).

*$\delta_2$-contraction degeneracy:* For $\delta_2 C$ we implemented three heuristic algorithms. The first one, *all-v* is based on the polynomial time implementation for $\delta_2 D$. We fix all vertices once at a time and perform the $\delta C$ heuristic (with least-c strategy) on the rest of the graph. The best second smallest degree recorded provides a lower bound on $\delta_2 C$. The other two $\delta_2 C$-heuristics are based on

the algorithms for $\delta C$. Instead of recording the minimum degree we also can record the second smallest degree (*M*aximum *S*econd *D*egree with contraction, abbreviated as MSD+). If we contract a vertex of minimum degree with one of its neighbours (according to the least-c strategy), we obtain the algorithm MSD+1. If the vertex of second smallest degree is contracted with one of its neighbours (also according to the least-c strategy), we obtain the algorithm MSD+2.

*$\gamma_R$-contraction degeneracy:* For $\gamma_R C$ the same strategies as for $\gamma_R D$ have been implemented. The only difference is that instead of removing all vertices in $V'$ or $v_j$, we contract each of the vertices with a neighbour that is selected according to the least-c strategy. Inspired by the good results of the '$\delta_2 C$ all-v' heuristic, we furthermore implemented the all-v strategy as described above also for the $\gamma_R$-contraction degeneracy. The difference is that instead of computing $\delta_2$ of each obtained minor, we now compute $\gamma_R$.

### 4.3 Experiments

The algorithms and heuristics described above have been tested on a large number of graphs from various application areas such as probabilistic networks, frequency assignment, travelling salesman problem and vertex colouring (see e.g. [7, 8] for details). All algorithms have been written in C++, and the computations have been carried out on a Linux operated PC with a 3.0 GHz Intel Pentium 4 processor. Many of the tested graphs as well as most of the experimental results on their treewidth (from, among others, [7, 8] and this article) can be obtained from [23].

In the tables below, we present the results for some selected instances only. The result of these representative instances reflect typical behaviour for the whole set of instances. The best known upper bound for treewidth (see [14]) is reported in the column with title UB. Columns headed by LB give treewidth lower bounds in the terms of the according parameter or a lower bound for the parameter. The best lower bounds in the tables are highlited in bold font. Values in columns headed by CPU are running times in seconds.

Table 1 shows the sizes of the graphs, and the results obtained for the treewidth lower bounds without contraction. These bounds are the exact parameters apart from the values for the two $\gamma_R D$-heuristics. As the computation times for $\delta$, $\delta_2$ and $\gamma_R$ are negligible, we omit them in the table. Also the $\delta D$ can be computed within a fraction of a second. The computational complexity of $\delta_2 D$ is $O(n)$ larger than the one of $\delta D$ which is reflected in the CPU times for this parameter.

Table 2 shows the results for the same graphs as in Table 1. Furthermore, in Table 2, we give the treewidth lower bounds according to the parameters that involve contraction. For $\delta C$, we only give the results of the least-c strategy, as this seems to be the most promising one (see [7]). For $\delta_2 C$ and $\gamma_R C$, the results of the heuristics as described in Section 4.2 are shown.

**Table 1.** Graph sizes, upper bounds and lower bounds without contractions

| instance | size | | | $\delta$ | $\delta_2$ | $\gamma_R$ | $\delta D$ | | $\delta_2 D$ | | $\gamma_R D$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | left | | right | |
| | $\|V\|$ | $\|E\|$ | UB | LB | LB | LB | LB | CPU | LB | CPU | LB | CPU | LB | CPU |
| link | 724 | 1738 | 13 | 0 | 0 | 0 | 4 | 0.01 | 4 | 3.67 | 4 | 0.01 | 4 | 0.01 |
| munin1 | 189 | 366 | 11 | 1 | 1 | 1 | 4 | 0.00 | 4 | 0.23 | 4 | 0.00 | 4 | 0.00 |
| munin3 | 1044 | 1745 | 7 | 1 | 1 | 1 | 3 | 0.01 | 3 | 6.70 | 3 | 0.02 | 3 | 0.01 |
| pignet2 | 3032 | 7264 | 135 | 2 | 2 | 2 | 4 | 0.04 | 4 | 69.87 | 4 | 0.04 | 4 | 0.05 |
| celar06 | 100 | 350 | 11 | 1 | 1 | 1 | 10 | 0.01 | **11** | 0.08 | **11** | 0.00 | 10 | 0.00 |
| celar07pp | 162 | 764 | 18 | 3 | 3 | 3 | 11 | 0.01 | 12 | 0.27 | 12 | 0.00 | 11 | 0.01 |
| graph04 | 200 | 734 | 55 | 3 | 3 | 3 | 6 | 0.01 | 6 | 0.36 | 6 | 0.00 | 6 | 0.00 |
| rl5934-pp | 904 | 1800 | 23 | 3 | 3 | 3 | 3 | 0.01 | 3 | 5.33 | 3 | 0.01 | 3 | 0.01 |
| school1 | 385 | 19095 | 188 | 1 | 1 | 1 | 73 | 0.04 | 74 | 7.89 | 75 | 0.03 | 73 | 0.03 |
| school1-nsh | 352 | 14612 | 162 | 1 | 1 | 1 | 61 | 0.02 | 62 | 5.69 | 62 | 0.03 | 61 | 0.02 |
| zeroin.i.1 | 126 | 4100 | 50 | 28 | 29 | 32 | 48 | 0.00 | 48 | 0.58 | **50** | 0.01 | **50** | 0.01 |

**Table 2.** Treewidth lower bounds with contraction

| instance | $\delta C$ | | $\delta_2 C$ | | | | | | $\gamma_R C$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | least-c | | all-v | | MSD+1 | | MSD+2 | | left | | right | | all-v | |
| | LB | CPU | LB | CPU | LB | CPU | LB | CPU | LB | CPU | LB | CPU | LB | CPU |
| link | 11 | 0.02 | **12** | 17.27 | 11 | 0.02 | 11 | 0.03 | 11 | 0.02 | **12** | 0.02 | **12** | 150.13 |
| munin1 | **10** | 0.01 | **10** | 0.58 | **10** | 0.00 | **10** | 0.00 | 9 | 0.01 | **10** | 0.00 | **10** | 3.07 |
| munin3 | **7** | 0.01 | **7** | 13.20 | **7** | 0.01 | **7** | 0.02 | **7** | 0.01 | **7** | 0.02 | **7** | 312.92 |
| pignet2 | 38 | 0.11 | **40** | 369.00 | 39 | 0.12 | 39 | 0.14 | 38 | 0.12 | 39 | 0.12 | **40** | 11525.1 |
| celar06 | **11** | 0.00 | **11** | 0.16 | **11** | 0.01 | **11** | 0.00 | **11** | 0.00 | **11** | 0.00 | **11** | 0.30 |
| celar07pp | **15** | 0.00 | **15** | 0.77 | **15** | 0.01 | **15** | 0.01 | **15** | 0.00 | **15** | 0.01 | **15** | 2.08 |
| graph04 | 20 | 0.01 | 20 | 2.72 | 20 | 0.01 | 19 | 0.01 | 20 | 0.02 | 19 | 0.01 | **21** | 4.78 |
| rl5934-pp | 5 | 0.02 | **6** | 36.12 | 5 | 0.02 | 5 | 0.03 | 5 | 0.03 | **6** | 0.02 | **6** | 221.72 |
| school1 | 122 | 0.48 | 124 | 180.30 | 123 | 0.48 | 122 | 0.51 | 122 | 0.45 | 122 | 0.49 | **125** | 215.35 |
| school1-nsh | 106 | 0.37 | **108** | 173.51 | 106 | 0.35 | 107 | 0.38 | 104 | 0.34 | 106 | 0.36 | **108** | 146.19 |
| zeroin.i.1 | **50** | 0.03 | **50** | 6.25 | **50** | 0.03 | **50** | 0.03 | **50** | 0.03 | **50** | 0.03 | **50** | 5.43 |

### 4.4 Discussion

The results of algorithms and heuristics that do not involve edge-contractions (Table 1) show that the degeneracy lower bounds (i.e. the lower bounds involving subgraphs) are significantly better than the simple lower bounds, as could be expected. Comparing the results for $\delta D$ and $\delta_2 D$, we see that in four cases we have that $\delta_2 D = \delta D + 1$. In the other seven cases $\delta_2 D = \delta D$. Bigger gaps than one between $\delta D$ and $\delta_2 D$ are not possible (confirm Lemma 2). In some cases other small improvements (compared to $\delta D$ and $\delta_2 D$) could be obtained by the heuristics for $\gamma_R D$. The two $\gamma_R D$-heuristics are all comparable in value and running times. Apart from the running times for computing $\delta_2 D$, the computation times are in all cases marginal, which is desirable for methods involving computing lower bounds many times (e.g. branch & bound). Even though the $\delta_2 D$

algorithm has much higher running times than the other algorithms in Table 1, it is still much faster than some heuristics with contraction. Furthermore, we expect that its running time could be improved by a more efficient implementation. No further investigations about parameters without contraction have been carried out as the parameters with contraction are of considerably more interest.

We can see that when using edge-contractions, the treewidth lower bounds can be significantly improved (compare Table 2 with Table 1). The results show that values for $\delta_2 C$ are typically equal or only marginal better than the value for $\delta C$. The same is true for $\gamma_R C$ with respect to $\delta_2 C$. The best results are obtained by the most time consuming algorithms: $\delta_2 C$ and $\gamma_R C$ with all-v strategy. By construction of the heuristic for $\gamma_R C$ with all-v strategy, it is clear that it is at least as good as the heuristic for $\delta_2 C$ with all-v strategy. Sometimes, it is even a little bit better. As in the case of the $\delta_2 D$ algorithm, the computation times of the $\delta_2 C$ and $\gamma_R C$ heuristics with all-v strategy could probably be improved by more efficient implementations. The other strategies for $\delta_2 C$ and $\gamma_R C$ are comparable in value and running times. No clear trend between them could be identified. In a few cases, we can observe that the gap between $\delta C$ and $\delta_2 C$ is more than one. This does not contradict Lemma 2, because the considered values are not the exact values. Different strategies for heuristics can result in different values with larger gaps between them. With the same argument, we can explain that in a few cases lower bounds of one parameter that in theory is at least as good as another parameter can be smaller than lower bounds of the other parameter.

As said above, using $\gamma_R$ instead of $\delta_2$ in the degeneracy and contraction degeneracy heuristics, gives only small improvements in some cases. Therefore, the ratio of two between those parameters as stated in Lemma 2 is far from the ratios observed in our experiments. Proving a smaller ratio and/or finding a graph with ratio as large as possible, remains a topic for further research.

It was already remarked in [7] that the $\delta$-contraction degeneracy of a planar graph can never be larger than 5. In fact, we have that $\delta C(G) \leq \delta_2 C(G) \leq \gamma(G) + 5$, where $\gamma$ denotes the genus of a graph (see [24]). This behaviour can be observed in our experiments, e.g. for the graph rl5934-pp, which is expected to be nearly planar. However, the $\gamma_R$-contraction degeneracy might be larger than $\gamma(G) + 5$.

## 5 Conclusions

In this article, we continued our research in [7] on degree-based treewidth lower bounds, where we combined the minimum degree lower bound with subgraphs and edge-contraction/minors. Here, we applied this combination to two other treewidth lower bounds, namely the second smallest degree lower bound and the Ramachandramurthi lower bound [19].

We obtained theoretical results showing how the parameters are related to each other. We also examined the computational complexity of the parameters. Here, it is interesting to note that all contraction degeneracy problems are

$NP$-hard, while the degeneracy problems are polynomial, except for the $\gamma_R$-degeneracy, which has been shown to be $NP$-hard.

In our experiments, we could observe potent improvements when comparing the simple parameters with their degeneracy counterparts. An even bigger improvement was achieved when edge-contractions were involved. Therefore, we can conclude that the incorporation of contraction improves the lower bounds for treewidth considerably. However, the added value of $\delta_2 C$ and $\gamma_R C$ in comparison to $\delta C$ is from a practical perspective marginal. The best lower bounds for $\delta_2 C$ and $\gamma_R C$ were obtained by heuristics with considerably long running times. Hence, if the lower bound has to be computed frequently, e.g. within a branch-and-bound algorithm, it is advisable to first compute a lower bound for $\delta C$, and only in tight cases using a slower but hopefully better lower bound.

It remains an interesting topic to research other treewidth lower bounds that can be combined with minors, in the hope to obtain large improvements. Furthermore, good lower bounds for graphs with bounded genus are desirable, because lower bounds based on $\delta$, $\delta_2$ or $\gamma_R$ do not perform very well on such graphs (see [24]). However, treewidth lower bounds for planar graphs (i.e. graphs with genus zero) can be obtained e.g. by computing the branchwidth of the graph (see [13, 22]).

## References


1. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a $k$-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.
2. M. Behzad, G. Chartrand, and L. Lesniak-Foster. *Graphs and Digraphs.* Pindle, Weber & Schmidt, Boston, 1979.
3. H. L. Bodlaender. A partial $k$-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
4. H. L. Bodlaender and A. M. C. A. Koster. On the Maximum Cardinality Search lower bound for treewidth. In J. Hromkovič, M. Nagl, and B. Westfechtel, editors, *Proc. 30th International Workshop on Graph-Theoretic Concepts in Computer Science WG 2004*, pages 81–92. Springer-Verlag, Lecture Notes in Computer Science 3353, 2004.
5. H. L. Bodlaender and A. M. C. A. Koster. Safe separators for treewidth. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments ALENEX04*, pages 70–78, 2004.
6. H. L. Bodlaender, A. M. C. A. Koster, F. v. d. Eijkhof, and L. C. van der Gaag. Preprocessing for triangulation of probabilistic networks. In J. Breese and D. Koller, editors, *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 32–39, San Francisco, 2001. Morgan Kaufmann.
7. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. In S. Albers and T. Radzik, editors, *Proceedings 12th Annual European Symposium on Algorithms, ESA2004*, pages 628–639. Springer, Lecture Notes in Computer Science, vol. 3221, 2004.
8. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. Technical Report UU-CS-2004-34, Dept. of Computer Science, Utrecht University, Utrecht, The Netherlands, 2004.

9. F. Clautiaux, S. N. A. Moukrim, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Oper. Res.*, 38:13–26, 2004.
10. F. Clautiaux, J. Carlier, A. Moukrim, and S. Négre. New lower and upper bounds for graph treewidth. In J. D. P. Rolim, editor, *Proceedings International Workshop on Experimental and Efficient Algorithms, WEA 2003*, pages 70–80. Springer Verlag, Lecture Notes in Computer Science, vol. 2647, 2003.
11. F. v. d. Eijkhof and H. L. Bodlaender. Safe reduction rules for weighted treewidth. In L. Kučera, editor, *Proceedings 28th Int. Workshop on Graph Theoretic Concepts in Computer Science, WG'02*, pages 176–185. Springer Verlag, Lecture Notes in Computer Science, vol. 2573, 2002.
12. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In proceedings UAI'04, Uncertainty in Artificial Intelligence, 2004.
13. I. V. Hicks. Planar branch decompositions I: The ratcatcher. INFORMS Journal on Computing (to appear, 2005).
14. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. In H. Broersma, U. Faigle, J. Hurink, and S. Pickl, editors, *Electronic Notes in Discrete Mathematics*, volume 8. Elsevier Science Publishers, 2001.
15. A. M. C. A. Koster, S. P. M. van Hoesel, and A. W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40:170–180, 2002.
16. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. Technical Report UU-CS-2004-050, Institute for Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.
17. S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.
18. B. Lucena. A new lower bound for tree-width using maximum cardinality search. *SIAM J. Disc. Math.*, 16:345–353, 2003.
19. S. Ramachandramurthi. *Algorithms for VLSI Layout Based on Graph Width Metrics*. PhD thesis, Computer Science Department, University of Tennessee, Knoxville, Tennessee, USA, 1994.
20. S. Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Disc. Math.*, 10:146–157, 1997.
21. N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of treewidth. *J. Algorithms*, 7:309–322, 1986.
22. P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.
23. Treewidthlib. http://www.cs.uu.nl/people/hansb/treewidthlib, 2004-03-31.
24. T. Wolle, A. M. C. A. Koster, and H. L. Bodlaender. A note on contraction degeneracy. Technical Report UU-CS-2004-042, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.

# Inferring AS Relationships: Dead End or Lively Beginning?

Xenofontas Dimitropoulos[1,2], Dmitri Krioukov[2], Bradley Huffaker[2], kc claffy[2], and George Riley[1]

[1] School of Electrical and Computer Engineering,
Georgia Institute of Technology,
Atlanta, Georgia 30332–0250
{fontas, riley}@ece.gatech.edu
[2] Cooperative Association for Internet Data Analysis (CAIDA),
La Jolla, California 92093–0505
{dima, brad, kc}@caida.org

**Abstract.** Recent techniques for inferring business relationships between ASs [1, 2] have yielded maps that have extremely few *invalid* BGP paths in the terminology of Gao [3]. However, some relationships inferred by these newer algorithms are incorrect, leading to the deduction of unrealistic AS hierarchies. We investigate this problem and discover what causes it. Having obtained such insight, we generalize the problem of AS relationship inference as a multiobjective optimization problem with node-degree-based corrections to the original objective function of minimizing the number of invalid paths. We solve the generalized version of the problem using the semidefinite programming relaxation of the MAX2SAT problem. Keeping the number of invalid paths small, we obtain a more veracious solution than that yielded by recent heuristics.

## 1 Introduction

As packets flow in the Internet, money also flows, not necessarily in the same direction. Business relationships between ASs reflect both flows, indicating a direction of money transfer as well as a set of constraints to the flow of traffic. Knowing AS business relationships is therefore of critical importance to providers, vendors, researchers, and policy makers, since such knowledge sheds more light on the relative "importance" of ASs.

The problem is also of multidimensional interest to the research community. Indeed, the Internet AS-level topology and its evolutionary dynamics result from business decisions among Internet players. Knowledge of AS relationships in the Internet provides a valuable validation framework for economy-based Internet topology evolution modeling, which in turn promotes deeper understanding of the fundamental laws driving the evolution of the Internet topology and its hierarchy.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 113–125, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Unfortunately, the work on inferring AS relationships from BGP data has recently encountered difficulties. We briefly describe this situation in its historical context.

Gao introduces the AS relationship inference problem in her pioneering paper [3]. This work approximates reality by assuming that any AS-link is of one of the following three types: customer-provider, peering, or sibling. If all ASs strictly adhere to import and export policies described in [3], then every BGP path must comply with the following hierarchical pattern: an uphill segment of zero or more customer-to-provider or sibling-to-sibling links, followed by zero or one peer-to-peer links, followed by a downhill segment of zero or more provider-to-customer or sibling-to-sibling links. Paths with the described hierarchical structure are deemed *valid*. After introducing insight about valid paths, Gao proposes an inference heuristic that identifies top providers and peering links based on AS degrees and valid paths.

In [4], Subramanian *et al.* (SARK) slightly relax the problem by not inferring sibling links, and introduce a more consistent and elegant mathematical formulation. The authors render the problem into a combinatorial optimization problem: given an undirected graph $G$ derived from a set of BGP paths $P$, assign the edge type (customer-provider or peering) to every edge in $G$ such that the total number of valid paths in $P$ is maximized. The authors call the problem the type-of-relationship (ToR) problem, conjecture that it is NP-complete, and provide a simple heuristic approximation.

Di Battista *et al.* (DPP) in [1] and independently Erlebach *et al.* (EHS) in [2] prove that the ToR problem is indeed NP-complete. EHS prove also that it is even harder, specifically APX-complete.[1] More importantly for practical purposes, both DPP and EHS make the straightforward observation that peering edges *cannot* be inferred in the ToR problem formulation. Indeed, as the validation data presented by Xia *et al.* in [5] indicates, only 24.63% of the validated SARK peering links are correct.

Even more problematic is the following dilemma. DPP (and EHS) come up with heuristics that outperform the SARK algorithm in terms of producing smaller numbers of invalid paths [1, 2]. Although these results seem to be a positive sign, closer examination of the AS relationships produced by the DPP algorithm [6] reveals that the DPP inferences are further from reality than the SARK inferences. In the next section we show that improved solutions to the ToR problem do not yield practically correct answers and contain obviously misidentified edges, e.g. well-known global providers appear as customers of small ASs. As a consequence, we claim that improved solutions to the unmodified ToR problem do not produce realistic results.

An alternative approach to AS relationship inference is to disregard BGP paths and switch attention to other data sources (e.g. WHOIS) [7, 8], but noth-

[1] There exists no polynomial-time algorithm approximating an APX-complete problem above a certain *inapproximability* limit (ratio) dependent on the particular problem.

ing suggests that we have exhausted all possibilities of extracting relevant information from BGP data. Indeed, in this study we seek to answer the following question: can we adjust the original (ToR) problem formulation, so that an algorithmic solution to the modified problem would yield a better answer from the practical perspective?

The main contribution of this paper is that we positively answer this question. We describe our approach and preliminary results in the subsequent two sections, and conclude by describing future directions of this work.

## 2 Methodology

### 2.1 Inspiration Behind Our Approach

The main idea behind our approach is to formalize our knowledge regarding why improved solutions to the ToR problem fail to yield practically right answers. To this end we reformulate the ToR problem as a multiobjective optimization problem introducing certain corrections to the original objective function. We seek a modification of the original objective function, such that the minimum of the new objective function reflects an AS relationship mapping that is closer to reality.

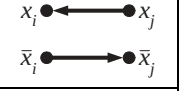
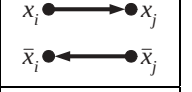
### 2.2 Mapping to 2SAT

To achieve this purpose, we start with the DPP and EHS results [1, 2] that deliver the fewest invalid paths. Suppose we have a set of BGP paths $P$ from which we can extract the undirected AS-level graph $G(V, E)$. We introduce *direction* to every edge in $E$ *from* the customer AS *to* the provider AS. Directing edges in $E$ induces direction of edges in $P$. A path in $P$ is valid if it does not contain the following invalid pattern: a provider-to-customer edge followed by a customer-to-provider edge. The ToR problem is to assign direction to edges in $E$ minimizing the number of paths in $P$ containing the invalid pattern.

The problem of identifying the directions of all edges in $E$ making *all* paths in $P$ valid—assuming such edge orientation exists—can be reduced to the 2SAT problem.[2] Initially, we arbitrarily direct all edges in $E$ and introduce a boolean variable $x_i$ for every edge $i$, $i = 1 \ldots \left|E\right|$. If the algorithms described below assign the value *true* to $x_i$, then edge $i$ keeps its original direction, while assignment of *false* to $x_i$ reverses the direction of $i$. We then split each path in $P$ into pairs of adjacent edges involving triplets of ASs (all 1-link paths are always valid) and perform mapping between the obtained pairs and 2-variable clauses as shown in Table 1. The mapping is such that only clauses corresponding to the invalid path pattern yield the *false* value when both variables are *true*. If there exists

[2] 2SAT is a variation of the satisfiability problem: given a set of clauses with two boolean variables per clause $l_i \vee l_j$, find an assignment of values to variables satisfying all the clauses. MAX2SAT is a related problem: find the assignment maximizing the number of simultaneously satisfied clauses.

**Table 1.** Mapping between pairs of adjacent edges in $P$, 2SAT clauses, and edges in $G_{2SAT}$. The invalid path pattern is in the last row

| Edges in $P$ | 2SAT clause | Edges in $G_{2SAT}$ |
|---|---|---|
|   | $x_i \vee x_j$ |   |
|   | $x_i \vee \bar{x}_j$ |   |
|   | $\bar{x}_i \vee x_j$ |   |
|   | $\bar{x}_i \vee \bar{x}_j$ |   |

an assignment of values to all the variables such that all clauses are satisfied, then this assignment makes all paths valid.

To solve the 2SAT problem, we construct a dual graph, the 2SAT graph $G_{2SAT}(V_{2SAT}, E_{2SAT})$, according to the rules shown in Table 1: every edge $i \in E$ in the original graph $G$ gives birth to two vertices $x_i$ and $\bar{x}_i$ in $V_{2SAT}$, and every pair of adjacent links $l_i \vee l_j$ in $P$, where literal $l_i$ ($l_j$) is either $x_i$ ($x_j$) or $\bar{x}_i$ ($\bar{x}_j$), gives birth to two directed edges in $E_{2SAT}$: from vertex $\bar{l}_i$ to vertex $l_j$ and from vertex $\bar{l}_j$ to vertex $l_i$. As shown in [9], there exists an assignment satisfying all the clauses if there is no edge $i$ such that both of its corresponding vertices in the 2SAT graph, $x_i, \bar{x}_i \in V_{2SAT}$, belong to the same strongly connected component[3] (SCC) in $G_{2SAT}$.

If an assignment satisfying all the clauses exists we can easily find it. We perform topological sorting[4] $t$ on nodes in $V_{2SAT}$ and assign *true* or *false* to a variable $x_i$ depending on if $t(\bar{x}_i) < t(x_i)$ or $t(x_i) < t(\bar{x}_i)$ respectively. All operations described so far can be done in linear time.

### 2.3 MAX2SAT: DPP Versus EHS

As soon as a set of BGP paths $P$ is "rich enough," there is no assignment satisfying all clauses and making all paths valid. Furthermore, the ToR problem of maximizing the number of valid paths can be reduced to the MAX2SAT [1, 2]

[3] An SCC is a set of nodes in a directed graph s. t. there exists a directed path between every ordered pair of nodes.

[4] Given a directed graph $G(V, E)$, function $t : V \mapsto \mathbb{R}$ is topological sorting if $t(i) \leq t(j)$ for every ordered pair of nodes $i, j \in V$ s. t. there exists a directed path from $i$ to $j$.

problem of maximizing the number of satisfied clauses. Making this observation, DPP propose a heuristic to find the maximal subset of paths $P_S \subset P$ such that all paths in $P_S$ are valid.

EHS use a different approach. They first direct the edges $i \in E$ that can be directed without causing conflicts. Such edges correspond to vertices $x_i, \bar{x}_i \in V_{2SAT}$ that have indegree or outdegree zero. Then EHS iteratively remove edges directed as described above and strip $P$, $G$, and $G_{2SAT}$ accordingly. This procedure significantly shortens the average path length in $P$, which improves the approximation of ToR by MAX2SAT. Finally, they approximate MAX2SAT to find a solution to the ToR problem.

### 2.4 Solving MAX2SAT with SDP

The MAX2SAT problem is NP- and APX-complete [10], but Goemans and Williamson (GW) [11] construct a famous approximation algorithm that uses semidefinite programming (SDP) and delivers an approximation ratio of 0.878. The best approximation ratio currently known is 0.940, due to improvements to GW by Lewin, Livnat, and Zwick (LLZ) in [12]. Note that this approximation ratio is pretty close to the MAX2SAT inapproximability limit of $\frac{21}{22} \sim 0.954$ [13].

To cast a MAX2SAT problem with $m_2$ clauses involving $m_1$ literals (variables $x_i$ and their negations $\bar{x}_i$, $i = 1 \dots m_1$) to a semidefinite program, we first get rid of negated variables by introducing $m_1$ variables $x_{m_1+i} = \bar{x}_i$. Then we establish mapping between boolean variables $x_k$, $k = 1 \dots 2m_1$, and $2m_1 + 1$ auxiliary variables $y_0, y_k \in \{-1, 1\}$, $y_{m_1+i} = -y_i$, using formula $x_k = (1 + y_0 y_k)/2$. This mapping guarantees that $x_k = true \Leftrightarrow y_k = y_0$ and $x_k = false \Leftrightarrow y_k = -y_0$. Given the described construction, we call $y_0$ the *truth* variable. After trivial algebra, the MAX2SAT problem becomes the maximization problem for the sum $1/4 \sum_{k,l=1}^{2m_1} w_{kl}(3 + y_0 y_k + y_0 y_l - y_k y_l)$, where weights $w_{kl}$ are either 1 if clause $x_k \vee x_l$ is present in the original MAX2SAT instance or 0 otherwise. Hereafter we fix the notations for indices $i, j = 1 \dots m_1$ and $k, l = 1 \dots 2m_1$.

The final transformation to make the problem solvable by SDP is relaxation. Relaxation involves mapping variables $y_0, y_k$ to $2m_1+1$ unit vectors $v_0, v_k \in \mathbb{R}^{m_1+1}$ fixed at the same origin—all vector ends lie on the unit sphere $S_{m_1}$. The problem is to maximize the sum composed of vector scalar products:

$$\begin{aligned} \max \ & \frac{1}{4} \sum_{k,l=1}^{2m_1} w_{kl}(3 + v_0 \cdot v_k + v_0 \cdot v_l - v_k \cdot v_l) \\ \text{s.t.} \ & v_0 \cdot v_0 = v_k \cdot v_k = 1, \ v_i \cdot v_{m_1+i} = -1, \\ & k = 1 \dots 2m_1, \ i = 1 \dots m_1. \end{aligned} \tag{1}$$

Interestingly, this problem, solvable by SDP, is equivalent to the following minimum energy problem in physics. Vectors $v_0, v_k$ point to the locations of particles $p_0, p_k$ freely moving on the sphere $S_{m_1}$ except that particles $p_i$ and $p_{m_1+i}$ are constrained to lie opposite on the sphere. For every MAX2SAT clause $x_k \vee x_l$, we introduce three constant forces of equal strength (see Fig. 1): one repulsive force between particles $p_k$ and $p_l$, and two attractive forces: between $p_k$ and $p_0$,

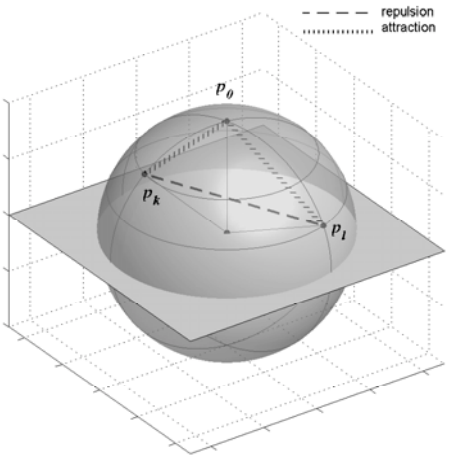



**Fig. 1.** The semidefinite programming relaxation to the MAX2SAT problem. Point $p_0$ (corresponding to vector $v_0$ from the text) is the *truth* point. It attracts both points $p_k$ and $p_l$ representing the boolean variables from the clause $x_k \vee x_l$. Points $p_k$ and $p_l$ repel each other. The problem is to identify the locations of all points on the sphere that minimize the potential energy of the system. Given an orientation by SDP, we cut the system by a random hyperplane and assign value *true* to the variables corresponding to points lying on the same side of the hyperplane as the truth $p_0$

and between $p_l$ and $p_0$—the *truth* particle $p_0$ attracts all other particles $p_k$ with the forces proportional to the number of clauses containing $x_k$. The goal is to find the location of particles on the sphere minimizing the potential energy of the system. If we built such a mission-specific computer in the lab, it would solve this problem in constant time. SDP solves it in polynomial time.

To extract the solution for the MAX2SAT problem from the solution obtained by SDP for the relaxed problem, we perform rounding. Rounding involves cutting the sphere by a randomly oriented hyperplane containing the sphere center. We assign value *true* (*false*) to variables $x_k$ corresponding to vectors $v_k$ lying on the same (opposite) side of the hyperplane as the *truth* vector $v_0$. GW prove that the solution to the MAX2SAT problem obtained this way delivers the approximation ratio of 0.878 [11]. We can also rotate the vector output obtained by SDP before rounding and skew the distribution of the hyperplane orientation to slightly prefer the orientation perpendicular to $v_0$. These two techniques explored to their greatest depths by LLZ improve the approximation ratio up to 0.9401 [12].

### 2.5 Analysis of the Unperturbed Solution

We now have the solution to the original ToR problem and are ready to analyze it. While the number of invalid paths is small [2], the solution is not

perfect—some inferred AS relationships are not in fact accurate. What causes these misclassifications?

First, some edges may be directed either way resulting in exactly the same number of invalid paths—such edges are directed randomly. To exemplify, consider path $p \in P$, $p = \{i_1 i_2 \dots i_{|p|-1} j\}$, $i_1, i_2, \dots, i_{|p|-1}, j \in E$, and suppose that the last edge $j$ appears only in one path (that is, $p$) and that it is from some large provider (like UUNET) to a small customer. Suppose that other edges $i_1, i_2, \dots, i_{|p|-1}$ appear in several other paths and that they are correctly inferred as customer-to-provider. In this scenario both orientations of edge $j$ (i.e. correct and incorrect: provider-to-customer and customer-to-provider) render path $p$ valid. Thus, edge $j$ is directed randomly, increasing the likelihood of an incorrect inference. We can find many incorrect inferences of this type in our experiments in the next section and in [6], e.g. well-known large providers like UUNET, AT&T, Sprintlink, Level3, are inferred as customers of smaller ASs like AS1 (AS degree 67), AS2685 (2), AS8043 (1), AS13649 (7), respectively.

Second, not all edges are customer-to-provider or provider-to-customer. In particular, trying to direct sibling edges leads to proliferation of error. Indeed, when the only objective is to maximize the number of valid paths, directing a sibling edge brings the risk of misdirecting the dependent edges sharing a clause with the sibling edge. To clarify, consider path $p \in P$, $p = \{ij\}$, $i, j \in E$, and suppose that in reality $i$ is a sibling edge that appears in multiple paths and that $j$ is a customer-to-provider edge that appears only in one path $p$. The algorithm can classify edge $i$ either as customer-to-provider or provider-to-customer depending on the structure of the paths in which it appears. If this structure results in directing $i$ as provider-to-customer, then the algorithm erroneously directs edge $j$ also as provider-to-customer to make path $p$ valid. In other words, the outcome is that we maximize the number of valid paths at the cost of inferring edge $j$ incorrectly.

We can conclude that the maximum number of valid paths does not correspond to a correct answer because, as illustrated in the above two examples, it can result in miss-inferred links. Specifically, in the presence of multiple solutions there is nothing in the objective function to require the algorithm to prefer the proper orientation for edge $j$. Our next key question is: Can we adjust the objective function to infer the edge direction correctly?

### 2.6 Our New Generalized Objective Function

A rigorous way to pursue the above question is to add to the objective function some small modifier selecting the correct edge direction for links unresolved by the unperturbed objective function. Ideally this modifier should be a function of "AS importance," such as the relative size of the customer tree of an AS. Unfortunately, defined this way the modifier is a function of the end result, edge orientation, which makes the problem intractable (i.e. we cannot solve it until we solve it).

The simplest correcting function that does not depend on the edge direction and is still related to perceived "AS importance," is the AS degree "gradient" in

the original undirected graph $G$—the difference between node degrees of adjacent ASs. In the examples from the previous subsection, the algorithm that is trying not only to minimize the number of invalid paths but also to direct edges from adjacent nodes of lower degrees to nodes of higher degrees will effectively have an incentive to correctly infer the last edge $j \in p$.

More formally, we modify the objective function as follows. In the original problem formulation, weights $w_{kl}$ for 2-link clauses $x_k \vee x_l$ (pairs of adjacent links in $P$) are either 0 or 1. We first alter them to be either 0, if pair $\{kl\} \notin P$, or $w_{kl}(\alpha) = c_2\alpha$ otherwise. The normalization coefficient $c_2$ is determined from the condition $\sum_{k \neq l} w_{kl}(\alpha) = \alpha \Rightarrow c_2 = 1/m_2$ (recall that $m_2$ is the number of 2-link clauses), and $\alpha$ is an external parameter, $0 \leq \alpha \leq 1$, whose meaning we explain below.

In addition, for every edge $i \in E$, we introduce a 1-link clause weighted by a function of the node degree gradient. More specifically, we initially orient every edge $i \in E$ along the node degree gradient: if $d_i^-$ and $d_i^+$, $d_i^- \leq d_i^+$, are degrees of nodes adjacent to edge $i$, we direct $i$ from the $d_i^-$-degree node to the $d_i^+$-degree node, for use as input to our algorithm.[5]

Then, we add 1-link clauses $x_i \vee x_i$, $\forall i \in E$, to our MAX2SAT instance, and we weight them by $w_{ii}(\alpha) = c_1(1-\alpha)f(d_i^-, d_i^+)$. The normalization coefficient $c_1$ is determined from the condition $\sum_i w_{ii}(\alpha) = 1 - \alpha$, and the function $f$ should satisfy the following two conditions: 1) it should "roughly depend" on the *relative* node degree gradient $(d_i^+ - d_i^-)/d_i^+$; and 2) it should provide higher values for node pairs with the same relative degree gradient but higher absolute degree values. The first condition is transparent: we expect that an AS with node degree 5, for example, is more likely a customer of an AS with node degree 10 than a 995-degree AS is a customer of a 1000-degree AS. The second condition is due to the fact that we do not know the true AS degrees: we approximate them by degrees of nodes in our BGP-derived graph $G$. The graphs derived from BGP data have a tendency to underestimate the node degree of small ASs, while they yield more accurate degrees for larger ASs [14]. Because of the larger error associated with small ASs, an AS with node degree 5, for example, is less likely a customer of an AS with node degree 10 than a 500-degree AS is a customer of a 1000-degree AS.

We select the following function satisfying the two criteria described above:

$$f(d_i^-, d_i^+) = \frac{d_i^+ - d_i^-}{d_i^+ + d_i^-} \log(d_i^+ + d_i^-). \tag{2}$$

In summary, our new objective function looks exactly as the one in (1), but with different weights on clauses:

[5] An initial direction along the node degree gradient does not affect the solution since any initial direction is possible. We select the node degree gradient direction to simplify stripping of non-conflict edges in the next section.

$$w_{kl}(\alpha) = \begin{cases} c_2\alpha & \text{if} \quad \{kl\} \in P, \\ c_1(1-\alpha)f(d_k^-, d_k^+) & \text{if} \quad k = l \le m_1, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Now we can explain the role of the parameter $\alpha$. Since $\sum_{k\neq l} w_{kl}(\alpha) = \alpha$ and $\sum_{k=l} w_{kl}(\alpha) = 1 - \alpha$, parameter $\alpha$ measures the relative importance of sums of all 2- and 1-link clauses. If $\alpha = 1$, then the problem is equivalent to the original unperturbed ToR problem—only the number of invalid paths matters. If $\alpha = 0$, then, similar to Gao, only node degrees matter. Note that in the terminology of multiobjective optimization, we consider the simplest scalar method of weighted sums.

In our analogy with physics in Fig. 1, we have weakened the repulsive forces among particles other than the *truth* particle $p_0$, and we have strengthened the forces between $p_0$ and other particles. When $\alpha = 0$, there are no repulsive forces, the *truth* particle $p_0$ attracts all other particles to itself, and all the vectors become collinear with $v_0$. Cut by any hyperplane, they all lie on the same side as $v_0$, which means that all variables $x_i$ are assigned value *true* and all links $i$ remain directed along the node degree gradient in the output of our algorithm.

## 3 Results

In our experiments, the BGP path set $P$ is a union of BGP tables from RouteViews [15] and 18 BGP route servers from [16] collected on May 13, 2004. Paths of length 1 are removed since they are always valid. The total number of paths is 1,025,775 containing 17,557 ASs, 37,021 links, and 382,917 unique pairs of adjacent links.

We first pre-process the data by discovering sibling links. For this purpose, we use a union of WHOIS databases from ARIN, RIPE, APNIC, and LACNIC collected on June 10, 2004. We say that two ASs belong to the same organization if, in the WHOIS database, they have exactly the same organization names, or names different only in the last digits, e.g. "ATT-37" and "ATT-38,"or very similar names, e.g. "UUNET South Africa" and "UUNET Germany." We infer links in $P$ between adjacent ASs belonging to the same organization as sibling. We find 211 sibling links in our dataset, which we ignore in subsequent steps. More precisely, we do not assign boolean variables to them.

We then direct the remaining links in the original graph $G$ along the node degree gradient, assign boolean variables to them, and construct the dual $G_{2SAT}$ graph. After directing edge $i$ along the node degree gradient, we check whether this direction satisfies all clauses containing $l_i$ ($x_i$ or $\bar{x}_i$). If so, we then remove the edge and strip $P$, $G$, and $G_{2SAT}$ accordingly. In this case we say that edge $i$ causes no conflicts because the value of the corresponding literal $l_i$ satisfies all the clauses in which $l_i$ appears, independent of the values of all other literals sharing the clauses with $l_i$. A non-conflict edge has two corresponding vertices in the $G_{2SAT}$ graph, $x_i$ and $\bar{x}_i$. It follows from the construction of the $G_{2SAT}$ graph that $x_i$ has an outdegree of zero and $\bar{x}_i$ has an indegree of zero. We repeat

the described procedure until we cannot remove any more edges. The stripped graph $G$ has 1,590 vertices (9% of the original $|V|$) and 4,249 edges (11% of the original $|E|$). The stripped $G_{2SAT}$ graph has 8,498 vertices and 46,920 edges. In summary, we have 4,249 ($m_1$) 1-link clauses and 23,460 ($m_2$) 2-link clauses. We feed this data into a publicly available SDP solver DSDP v4.7 [17], reusing parts of the code from [2] and utilizing the LEDA v4.5 software library [18]. We incorporate the pre-rounding rotation and skewed distribution of hyperplane orientation by LLZ [12].

Fig. 2 shows results of edge orientations we derive for different values of $\alpha$ in (3). Specifically, the figure shows the percentage of valid paths, edges directed as in the $\alpha = 0$ case, and edges directed as in the $\alpha = 1$ case. In the particular extreme case of $\alpha = 1$, the problem reduces to the original ToR problem considered by DPP and EHS, and its solution yields the highest percentage of valid paths, 99.67%. By decreasing $\alpha$, we increase preference to directing edges along the node degree gradient, and at the other extreme of $\alpha = 0$, all edges become directed along the node gradient, but the number of valid paths is 92.95%.

Note that changing $\alpha$ from 0 to 0.1 redirects 1.64% of edges, which leads to a significant 6.53% increase in the number of valid paths. We also observe that the tweak of $\alpha$ from 1 to 0.9 redirects 2.56% of edges without causing any significant decrease (only 0.008%) in the number of valid paths. We find that most of these edges are directed randomly in the $\alpha = 1$ case because oriented either way they yield the same number of valid paths. In other words, the AS relationships represented by these edges cannot be inferred by minimizing the number of invalid paths.

We also rank ASs by means of our inference results with different $\alpha$ values. To this end we split all ASs into hierarchical levels as follows. We first order all ASs by their *reachability*—that is, the number of ASs that a given AS can reach "for free" traversing only provider-to-customer edges. We then group ASs with the same reachability into levels. ASs at the highest level can reach all other ASs "for free." ASs at the lowest level have the smallest reachability (fewest "free" destinations). Then we define the position *depth* of AS X as the number of ASs

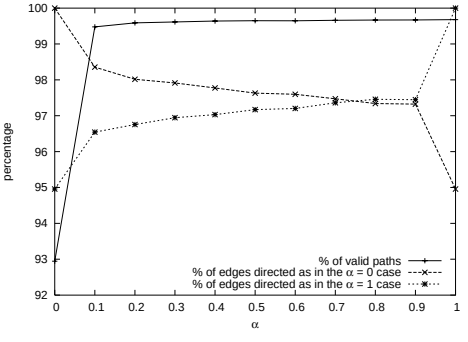



**Fig. 2.** Percentage of valid paths, of edges directed as in the $\alpha = 0$ case and of edges directed as in the $\alpha = 1$ case for different values of $\alpha$

**Table 2.** Hierarchical ranking of ASs. The position *depth* (the number of AS at the levels above) and *width* (the number of ASs at the same level) of the top five ASs in the $\alpha = 0$ and $\alpha = 1$ cases are shown for different values of $\alpha$. The customer leaf ASs are marked with asterisks

| | | | $\alpha = 0.0$ | | $\alpha = 0.2$ | | $\alpha = 0.5$ | | $\alpha = 0.8$ | | $\alpha = 1.0$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AS # | name | degree | dep. | wid. | dep. | wid. | dep. | wid. | dep. | wid. | dep. | wid. |
| 701 | UUNET | 2373 | 0 | 1 | 0 | 173 | 1 | 232 | 1 | 252 | 17 | 476 |
| 1239 | Sprint | 1787 | 1 | 1 | 0 | 173 | 1 | 232 | 1 | 252 | 17 | 476 |
| 7018 | AT&T | 1723 | 2 | 1 | 0 | 173 | 1 | 232 | 1 | 252 | 17 | 476 |
| 3356 | Level 3 | 1085 | 3 | 1 | 0 | 173 | 1 | 232 | 1 | 252 | 17 | 476 |
| 209 | Qwest | 1072 | 4 | 1 | 0 | 173 | 1 | 232 | 1 | 252 | 17 | 476 |
| 3643 | Sprint Austr. | 17 | 194 | 1 | 222 | 1 | 250 | 1 | 268 | 1 | 0 | 4 |
| 6721 | Nextra Czech Net | 3 | 1742 | 941 | 833 | 88 | 868 | 90 | 884 | 89 | 0 | 4 |
| 11551 | Pressroom Ser. | 2 | 1742 | 941 | 1419 | 398 | 1445 | 390 | 1457 | 386 | 0 | 4 |
| 1243 | Army Systems | 2 | 2683 | 14725* | 2753 | 14655* | 1445 | 390 | 1457 | 386 | 0 | 4 |
| 6712 | France Transpac | 2 | 2683 | 14725* | 2753 | 14655* | 292 | 3 | 1 | 252 | 4 | 13 |

at the levels above the level of AS X. The position *width* of AS X is the number of ASs at the same level as AS X.

Table 2 shows the results of our AS ranking. For different values of $\alpha$, we track the positions of the top five ASs in the $\alpha = 0$ and $\alpha = 1$ cases. In the former case, well-known large ISPs are at the top, but the number of invalid paths is relatively large, cf. Fig. 2. In the latter case delivering the solution to the unperturbed ToR problem, ASs with small degrees occupy the top positions in the hierarchy. These ASs appear in much lower positions when $\alpha \neq 1$. Counter to reality, the large ISPs are not even near the top of the hierarchy. We observe that the depth[6] of these large ASs increases as $\alpha$ approaches 1, indicating an increasingly stronger deviation from reality. The deviation is maximized when $\alpha = 1$. This observation pronounces the limitation of the ToR problem formulation based solely on maximization of the number of valid paths.

## 4 Conclusion and Future Work

Using a standard multiobjective optimization method, we have constructed a natural generalization of the known AS relationship inference heuristics. We have extended the combinatorial optimization approach based on minimization of invalid paths, by incorporating AS-degree-based information into the problem formulation. Utilizing this technique, we have obtained first results that are more realistic than the inferences produced by the recent state-of-the-art

[6] Note that the large ISPs are at the same depth as soon as $\alpha \neq 0$, which is expected since they form "almost a clique" [4] and are likely to belong to the same SCC. All nodes in the same SCC have the same reachability. The converse is not necessarily true.

heuristics [1, 2]. We conclude that our approach opens a promising path toward increasingly veracious inferences of business relationships between ASs.

The list of open issues that we plan to address in our future work includes: 1) modifications to the algorithm to infer *peering*; 2) careful analysis of the *trade-off surface* [19] of the problem, required for selecting the value of the external parameters (e.g. $\alpha$) corresponding to the right answer; 3) detailed examination of the *structure* of the AS graph directed by inferred AS relationships; 4) *validation* considered as a set of constraints narrowing the range of feasible values of external parameters; and 5) investigation of other *AS-ranking mechanisms* responsible for the structure of the inferred AS hierarchy.

## Acknowledgements

We thank Thomas Erlebach, Alexander Hall and Thomas Schank for sharing their code with us.

Support for this work was provided by the Cisco University Research Program, by DARPA N66002-00-1-893, and by NSF ANI-0221172, ANI-9977544, ANI-0136969 and CNS-0434996, with support from DHS/NCS.

## References

1. Battista, G.D., Patrignani, M., Pizzonia, M.: Computing the types of the relationships between Autonomous Systems. In: IEEE INFOCOM. (2003)
2. Erlebach, T., Hall, A., Schank, T.: Classifying customer-provider relationships in the Internet. In: Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN). (2002)
3. Gao, L.: On inferring Autonomous System relationships in the Internet. In: IEEE/ACM Transactions on Networking. (2001)
4. Subramanian, L., Agarwal, S., Rexford, J., Katz, R.H.: Characterizing the Internet hierarchy from multiple vantage points. In: IEEE INFOCOM. (2002)
5. Xia, J., Gao, L.: On the evaluation of AS relationship inferences. In: IEEE GLOBECOM. (2004)
6. Rimondini, M.: Statistics and comparisons about two solutions for computing the types of relationships between Autonomous Systems (2002) `http://www.dia.uniroma3.it/~compunet/files/ToR-solutions-comparison.pdf`.
7. Siganos, G., Faloutsos, M.: Analyzing BGP policies: Methodology and tool. In: IEEE INFOCOM. (2004)
8. Huber, B., Leinen, S., O'Dell, R., Wattenhofer, R.: Inferring AS relationships beyond counting edges. Technical Report TR 446, ETH Zürich (2004)
9. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear time algorithm for testing the truth of certain quantified boolean formulae. Information Processing Letters **8** (1979) 121–123
10. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties. Springer Verlag, Berlin (1999)
11. Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM **42** (1995) 1115–1145

12. Lewin, M., Livnat, D., Zwick, U.: Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems. In: Proceedings of the $9^{th}$ International IPCO Conference on Integer Programming and Combinatorial Optimization. (2002)
13. Håstad, J.: Some optimal inapproximability results. In: Proceedings of the $29^{th}$ Annual ACM Symposium on Theory of Computing. (1997)
14. Chang, H., Govindan, R., Jamin, S., Shenker, S.J., Willinger, W.: Towards capturing representative AS-level Internet topologies. Computer Networks Journal **44** (2004) 737–755
15. Meyer, D.: University of Oregon Route Views Project (2004)
16. : A traceroute server list. http://www.traceroute.org (2004)
17. Benson, S., Ye, Y., Zhang, X.: A dual-scaling algorithm for semidefinite programming (2004) `http://www-unix.mcs.anl.gov/~benson/dsdp/`.
18. GmbH, A.S.S.: L E D A library (2004) `http://www.algorithmic-solutions.com/enleda.htm`.
19. Collette, Y., Siarry, P.: Multiobjective Optimization: Principles and Case Studies. Springer-Verlag, Berlin (2003)

# Acceleration of Shortest Path and Constrained Shortest Path Computation

Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling*

TU Berlin, Institut für Mathematik,
Str. des 17. Juni 136, 10623 Berlin, Germany.
{Ekkehard.Koehler, Rolf.Moehring, Heiko.Schilling}@TU-Berlin.DE

**Abstract.** We study acceleration methods for point-to-point shortest path and constrained shortest path computations in directed graphs, in particular in road and railroad networks. Our acceleration methods are allowed to use a preprocessing of the network data to create auxiliary information which is then used to speed-up shortest path queries. We focus on two methods based on Dijkstra's algorithm for shortest path computations and two methods based on a generalized version of Dijkstra for constrained shortest paths. The methods are compared with other acceleration techniques, most of them published only recently. We also look at appropriate combinations of different methods to find further improvements. For shortest path computations we investigate hierarchical multiway-separator and arc-flag approaches. The hierarchical multiway-separator approach divides the graph into regions along a multiway-separator and gathers information to improve the search for shortest paths that stretch over several regions. A new multiway-separator heuristic is presented which improves the hierarchical separator approach. The arc-flag approach divides the graph into regions and gathers information on whether an arc is on a shortest path into a given region. Both methods yield significant speed-ups of the plain Dijkstra's algorithm. The arc flag method combined with an appropriate partition and a bi-directed search achieves an average speed-up of up to 1,400 on large networks. This combination narrows down the search space of Dijkstra's algorithm to almost the size of the corresponding shortest path for long distance shortest path queries. For the constrained shortest path problem we show that goal-directed and bi-directed acceleration methods can be used both individually and in combination. The goal-directed search achieves the best speed-up factor of 110 for the constrained problem.

## 1 Introduction

In combinatorial optimization computing shortest paths is regarded as one of the most fundamental problems. What makes the shortest paths problem so interesting is the important role it plays in numerous real world problems: combinatorial models of real world scenarios often contain or reduce to shortest path computations. Much research has been done on shortest path problems and there is a large variety of different algorithms for computing shortest paths efficiently in a given network. In the present paper

* Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research Cluster "Algorithms on Large and Complex Networks" (1126).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 126–138, 2005.
© Springer-Verlag Berlin Heidelberg 2005

we look at one of the most common variants of the problem where one has to find a shortest path between two nodes in a directed graph, the *point-to-point shortest path problem* (*P2P*). Motivated by real-world applications, we assume that the shortest path problem has to be solved repeatedly for the same network . Thus, preprocessing of the network data is possible and can support the computations that follow. The purpose of this paper is to study implementations of different acceleration methods for shortest path and constrained shortest path computations in traffic networks. In contrast to general graphs, road networks are very sparse graphs. They usually have an embedding in the plane and the considered arc lengths often resemble Euclidean distances.

For shortest path computations we compare several approaches, published only recently, and focus on four approaches which we found appropriate for our purpose. For shortest path computations we first investigate a hierarchical multiway-separator method similar to Frederikson's [5]. In a preprocessing phase we determine a small node multiway-separator that divides the graph into regions of almost balanced size, then information is gathered on the distances between the separator nodes of that particular region and is used in subsequent shortest path computations. Second, we consider a generalization of a region-based arc labeling approach that we refer to as the *arc flag approach*. The basic idea of the arc flag method using a simple rectangular partition has been suggested by Lauther [12]. The arc-flag approach divides the graph into regions and gathers information for each arc on whether this arc is on a shortest path into a given region. For each arc this information is stored in a vector. More precisely, the vector contains a flag for each region of the graph indicating whether this arc is on a shortest path into that particular region. Thus, the size of each vector is determined by the number of regions and the number of vectors is determined by the number of arcs. Arc flags are used in the Dijkstra computation to avoid exploring unnecessary paths.

In addition to ordinary shortest path computations, we also study accelerating constrained shortest path computations. We look at networks where each arc is assigned two values: a length and a cost. The aim is to compute a shortest path with respect to the length such that the sum of the cost values of the corresponding arcs does not exceed a given cost bound. This is a well-known weakly NP-hard problem. A standard algorithm for constrained shortest paths is a generalized Dijkstra algorithm [1]. In the present paper we analyze the behavior of several acceleration methods for shortest paths when applied to the generalized Dijkstra algorithm. In particular we investigate a goal-directed search, a bi-directed search, and a combination of the two. The study is motivated by a routing project in cooperation with DaimlerChrysler AG. In this project we have to compute routes which guarantee a given fairness condition. This is where the constrained shortest path problem comes in.

To compare the different approaches we look at computational results for a given set of road and railroad networks. We first present the results for the main methods in this paper. We also compare them with results for other methods and finally we take into account combinations of our main methods with the other methods we discussed. In our tests we only present combinations which seem to be appropriate and leave out non-appropriate methods as, for example, pure goal-directed search in combination with the arc-flag method, since the latter is already highly goal-directed by construction.

*Related work.* There are various studies of acceleration methods for shortest path computations. A recent overview is given in [6]. For hierarchical methods we refer to [5, 18]. Gutman [8] introduces a method based on the concept of *reach*: for each node a single reach value together with Euclidean coordinates is stored in order to enable a specific kind of goal-directed search. Gutman reports that he computes shortest paths 10 times faster than the plain Dijkstra algorithm. Goldberg describes an approach which uses the $A^*$ search in combination with a lower-bounding technique based on so-called *landmarks* and the triangle inequality. Using just one landmark the results that Goldberg's algorithm produces are not as good as Gutman's, but with 16 landmarks he reports on a speed-up of up to 17. The basic arc-flag approach using a rectangular geographic partition of the underlying graph is described by Lauther [13], who observed a speed-up of up to 64. Experimental studies for other geometric speed-up techniques can be found in Holzer, Schulz, and Willhalm [9]. For a recent overview on the many techniques for the constrained shortest path problem we refer to [4]. Recent experimental results on it can be found in [14, 17].

*Our contributions.* For the hierarchical multiway-separator approach we extend Goodrich's algorithm to non-planar graphs. We introduce a new heuristic which for our data computes a smaller multiway-separator than METIS [15]. The sizes of the multiway-separator range between 67 % and 85 % of the sizes of the separators computed by METIS. Because of the smaller multiway-separator size our heuristic improves the hierarchical multiway-separator approach up to a speed-up factor of 14.

With the arc-flag method we investigate a new type of shortest path acceleration. It uses a partition of the node set of the graph into regions and precomputes one bit (flag) of information per arc and region. It consistently yields the best speed-up results on our road networks. When combined with an arc separator partition we obtain a speed-up factor of 220. A combination with a bi-directed search yields a speed-up factor of up to 1,400. It may seem promising to combine the arc flag method also with Gutman's acceleration method [8]. However, our experiments have shown that although this method reduces the search space, it does not reduce the running time any further.

For the constrained shortest path problem we show that the goal-directed and the bi-directed approach can be used both individually and in combination. Here, the simple goal-directed search yields the greatest speed-up factors (110) and the bi-directed search which does not need preprocessing still provides reasonable results (factor of 5). To our knowledge this is the first time that these standard techniques have been applied to constrained shortest path problems on road networks.

## 2 Preliminaries

The input to the P2P problem is a directed graph $G = (V,A)$ with $n := |V|$ nodes, $m := |A|$ arcs, a source node $s$, a target node $t$ and a nonnegative arc length $\ell(a)$, for each arc $a \in A$. Additionally, in the constrained shortest path case, there are nonnegative arc costs $c(a)$ for each arc $a \in A$. The P2P problem is to find a length minimal path in a graph $G$ from $s$ to $t$, i.e., the sum of the arc lengths of all arcs in the path should be minimal. We will refer to the path as a *shortest $s$,$t$–path* in $G$ and the sum of its arc length is denoted by $dist_s(t)$, the *shortest path distance* from $s$ to $t$. In the *constrained*

*P2P problem* the aim is to find a length minimal $s,t$–path for which the sum of the arc costs of all arcs in the path does not exceed a given *cost bound* $C_{s,t}$.

Our acceleration methods are based on Dijkstra's algorithm [3] which computes distance labels $d_s(u)$ from $s$ to all reachable nodes $u \in V$ until $dist_s(t)$ is determined. The algorithm maintains a preliminary distance $d_s(v)$ for all nodes and a set $S$ of nodes whose final shortest path distance from $s$ has already been determined, i.e., $d_s(v) = dist_s(v)$. The algorithm starts with setting $d_s(s) = 0$ and inserts $s$ into $S$. Then it repeatedly scans nodes $u \notin S$ in nondecreasing order of their distance label $d_s(u)$. It inserts $u$ into $S$ and updates labels of all adjacent nodes $w$ with $(u,w) \in A$. Each node $u$ is scanned and inserted into $S$ at most once. On insertion arc $(u,v)$ is considered and then $d_s(w)$ is updated by the sum $d_s(u) + \ell(u,w)$ if it is *dominated* by the sum, i.e., $d_s(u) + \ell(u,w) < d_s(w)$. Note, that it is not necessary for the algorithm to traverse the whole graph. The set of arcs which are traversed during the run of the algorithm is the *search space*. With our acceleration methods we restrict Dijkstra's algorithm to a smaller search space that still leads to the shortest path and thus results in a faster running time. Using auxiliary precomputed information from our acceleration methods the algorithm is able to reject arcs before the update test which cannot be on a shortest path.

In the bi-directed search a second Dijkstra run is started simultaneously from $t$ and computes a distance $dist_t(u)$ from $t$ in the *reverse graph*, the graph with every arc reversed. The bi-directed search algorithm alternates between running the forward (common) and reverse search version of Dijkstra's algorithm and stops with an appropriate stopping criterion when the two searches meet. Note that any alternation strategy will correctly determine a shortest path.

In the constrained case we use a generalized version of Dijkstra's algorithm by Aneja, Aggarwal, and Nair [1]. Here, instead of one distance label per node a whole set of label pairs $(d_s(u), c_s(u))$ are used for each node $u$, each of them representing distance $d_s(u)$ and cost $c_s(u)$ of a path from $s$ to $u$. If a node $w$ is adjacent to $u$, all of its label pairs $(d_s(w), c_s(w))$ are removed if they are dominated by $(d_s(u) + \ell(u,w), c_s(u) + c(u,w))$, i.e., $d_s(u) + \ell(u,w) < d_s(w)$ and $c_s(u) + c(u,w) < c_s(w)$. Thus, the generalized version of Dijkstra's algorithm maintains a list of non-dominating label pairs at each node and stops once the target is reached.

## 3 Shortest Path Acceleration Methods

In this section we consider two acceleration methods for shortest path computation. Both methods have been used before for the case of planar embedded graphs. Here we extend them to work on almost planar graphs such as road or railroad networks. Note that in theory our extensions also work on arbitrary graphs.

*The Multiway-Separator Approach.* A significant acceleration can already be achieved by a divide and conquer method in combination with an appropriate preprocessing. In the *multiway-separator approach*, due to Frederickson [5], one computes a small node set whose removal partitions the graph into regions of roughly equal size such that there is no path between different regions.

Our heuristic to determine a balanced multiway-separator in road networks is based on an approach by Goodrich [7]. Goodrich uses a multiway-separator that divides the

graph in up to $O(n^\varepsilon)$, $0 < \varepsilon < \frac{1}{2}$, regions. The construction of the multiway-separator by Goodrich involves two steps. In the first step a breadth first search (BFS) tree from some root node $s$ is computed and $O(n^\varepsilon)$ so-called *starter-levels* ($0 < \varepsilon < \frac{1}{2}$) in the tree are determined. For each of these starter-levels in the tree a *cut-level* above and below is determined, such that each cut-level contains maximally $2\lceil\sqrt{n}\rceil$ nodes and its distance is at most $\sqrt{n}/2$ levels away from the associated starter level. The nodes in the cut-level are marked as separator nodes. The size of the regions is bounded by $O(n^{1-\varepsilon})$. In the second step Goodrich uses fundamental cycles to balance their size.

In our heuristic we make use of the first step of Goodrich's algorithm since the second step is not applicable to non-planar graphs. Instead, we apply again Goodrich's step 1 with a modified BFS-tree computation, together with a final cleaning step for merging small connected components. In that way we reduce the number of separator nodes and obtain regions of roughly equal size. Altogether, our multiway-separator heuristic consists of three steps: a BFS-tree computation for a coarse separation of the graph, a second BFS-tree computation for a finer separation, and a cleanup step. The size of the resulting multiway-separator and the regions depend essentially on the choice of the different parameters in our multiway-separator heuristic.

After the multiway-separator has been constructed, every node which is not in the multiway-separator is assigned to exactly one region. The separator nodes belong to all regions separated by them and are defined as border nodes for that region. Then, all shortest paths between separator nodes of the same region are precomputed and the paths and their lengths are made available via lookup tables. This provides efficient access during the subsequent path searches. For each determined shortest path between border nodes of a particular region an additional arc is introduced with the shortest path distance assigned to it as arc length. Border nodes together with the additionally inserted arcs form a *hierarchy layer* on top of the original graph. If a shortest path search starts at some node $s$ lying in some region $R_s$ the Dijkstra algorithm begins with scanning nodes in $R_s$. However, when leaving region $R_s$, the search algorithm walks only along arcs of the hierarchy layer, until it reaches the target region. Then, for the rest of the search it again walks along original arcs in the graph. If the determined shortest path stretches over several regions it consists of original and additionally inserted arcs. But it can be reconstructed with the path information stored in the lookup table. Our multiway-separator heuristic together with the hierarchical acceleration method delivers a speed-up factor of up to 14 compared to the plain Dijkstra's algorithm (see Section 5 for further results).

*The Arc-Flag Approach.* A significantly stronger speed-up can be achieved with the *arc-flag approach*. This approach is based on a partition of the graph into node sets $R_1, \ldots, R_k$, which we call *regions*. Each node is assigned to exactly one region. At each arc $a$ we store a flag for each region $R_i$ ($0 < i \leq k$). This flag is set to TRUE if $a$ is on a shortest path to at least one node in $R_i$ or if $a$ lies in $R_i$, otherwise it is set to FALSE. For each arc $a$ this information is stored in a vector of flags $f_a$. Thus the size of $f_a$ is $k$, the number of regions, whereas the number of vectors is the number of arcs (see Figure 1). A shortest path search from a node $s$ to a node $t$ in region $R_j$ can now be conducted using a Dijkstra algorithm that only traverses arcs $a$ where $f_a(j)$ is TRUE.

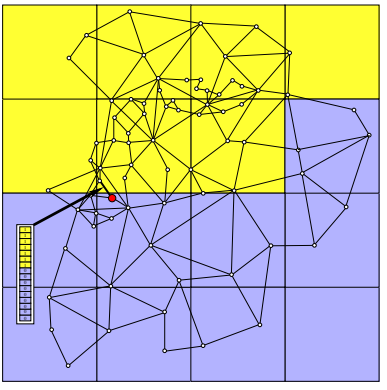


**Fig. 1.** In the arc-flag method at each arc $a$ a vector $f_a$ of arc-flags is stored such that $f_a[i]$ indicates if $a$ is on a shortest path into region $i$

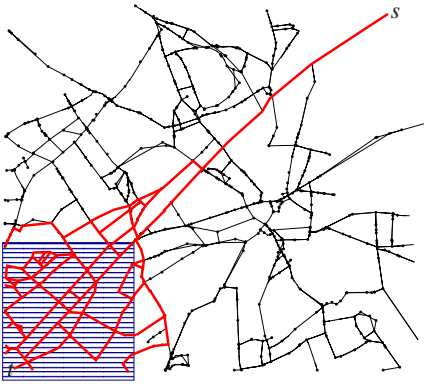



**Fig. 2.** The search space of a Dijkstra computation with arc-flag acceleration. The search started in $s$ and the region containing the target node $t$ is highlighted

The basic idea of this approach using a simple rectangular partition has been reported by Lauther [12]. His partition requires an embedding of the graph in the plane. When combined with a bi-directed search Lauther [13] obtains a speed-up factor of 64 on the European truck driver's road map (326,159 nodes, 513,567 arcs, 300 requests, 139 regions). With an improved partition of the graph we obtain a speed-up factor of 677 on an instance of roughly the same size (362,554 nodes, 920,464 arcs, 2,500 requests, 100 regions). Here we use fewer regions than Lauther and also apply bi-directed search. The speed-up of this method increases with larger instances up to a factor of 1,400.

The preprocessing for this approach can be done as follows. Note that all shortest paths entering a region $R_i$ have to use some arc $a$ that crosses the border of $R_i$. Now for each of those crossing arcs a shortest path tree in the reverse graph is computed starting at arc $a$. All arcs in this $a$–rooted reverse shortest path tree obtain the value TRUE in their flag-vector at position $i$. Doing this for all arcs entering $R_i$ one can fill up all entries at the $i$-th component of the flag-vector of all arcs in $G$. Note, that it is not possible to reduce the problem to one shortest path tree computation per region, since then it may be possible that we miss necessary flags. The set of arcs crossing the border of some region $R$ form an arc cut $\mathscr{C}_R$. The total preprocessing time for that region then amounts to $O(|\mathscr{C}_R| n \log n)$. This can be reduced further since information on a computed shortest path tree of a border crossing arc $a$ can be used for subsequent shortest path tree computations of that region. An additional reduction of the preprocessing time can be achieved by improving the partition, e.g., by computing small multi-way arc separator. Using METIS [15] for this task we can reduce the preprocessing time by a factor of 2 compared to the rectangular partitioning, while using the same number of regions. Here the shortest path query time decreases by a factor of up to 4. The reason for this additional speedup is the fact that the arc separator determined by METIS much better represents the specific structure of the graph.

## 4 Constrained Shortest Path Acceleration Methods

In addition to accelerating shortest path queries we have also investigated the effect of standard acceleration methods for the resource constrained shortest path problem.

*The Goal-Directed Approach.* Our *goal-directed approach* attempts to accelerate the path search by employing a lower bound on the (remaining) path lengths and costs to the target node. This is achieved by modifying the arc lengths and costs and thereby forcing Dijkstra's algorithm to prefer nodes closer to the target node over those further away. In road and railroad networks usually Euclidean distances are used as lower bounds. However, in the case of constrained shortest paths one can exploit the fact that computing the shortest path from the target node to all other nodes is cheap compared to the overall cost of the extended Dijkstra. Hence, by simply computing both a reverse shortest path tree from the target node with respect to length and a reverse shortest path tree with respect to cost, it is possible to determine very good lower bounds on the remaining path lengths. One can use these trees for directing the constrained shortest path search. Note that here it is not necessary to restrict the method to Euclidean distances. Thus our goal-directed technique for constrained shortest paths is not limited to graphs that are embedded in the plane. Compared to other acceleration methods, this procedure consistently has delivered the best results in our tests (up to speed-up factors of 110). More details on this acceleration method can be found in [11].

*The Bi-Directed Approach.* In the *bi-directed approach* the constrained shortest paths are computed simultaneously from the start and the target node. In a traditional bi-directed search there is a simple stopping criterion to stop the search when the two frontiers meet. However, in the resource constrained case the stopping criterion is more complex. We use the usual stopping criterion for the ordinary shortest path problem as a starting point for a generalization to constrained shortest paths. Our generalized stopping criterion requires the labeling Dijkstra to explore labels in a lexicographic order (length before cost). For more details, please refer to the extended version of this paper [11].

## 5 Implementation and Experiments

*Implementation.* The methods presented in this paper were implemented in C++ using the GNU g++ compiler version 3.4.2 with the optimizing option "-O3" on Linux 2.4/2.6 systems (SuSE 9.1). All computations were done on 64 bit machines: Intel Itanium II machines 1.2 GHz with 64 GB shared memory and 500 KB cache memory and AMD Opteron machines 2.2 GHz with 8 GB memory and 1 MB cache memory. For this work we had to efficiently plug together and test several different versions of (constrained) shortest path algorithms. For developing such a framework we used generic programming techniques via template meta-programming in C++ as described, e.g., in [2].

*Instances.* All computations were done on real world networks described in Table 1. Each arc in these networks has a nonnegative integer geographic length. The arc costs are nonnegative rational numbers and arise from a routing project; see [10] for more details. For each network instance we randomly generated up to 2,500 route requests. The measured speed-up factors and running times are averaged over all computed requests.

**Table 1.** Input networks used in the paper. Dijkstra running time is the shortest path query time computed by a plain Dijkstra's algorithm for a single request. The values are averaged over 2,500 requests which were computed on an Opteron processor (2.2 GHz)

| name | description | # nodes | # arcs | Dijkstra running time [sec] |
|---|---|---|---|---|
| B | Berlin | 12,100 | 19,570 | 0.1 |
| GR | German Railway | 14,938 | 32,520 | 0.2 |
| GH | German Highway | 53,315 | 109,540 | 1.0 |
| AA | North Rhine-Westphalia south | 362,554 | 920,464 | 5.3 |
| TH | Thuringia | 422,917 | 1,030,148 | 6.1 |
| OS | Berlin, Brandenburg, Saxony, Saxony-Anhalt, Mecklenburg | 474,431 | 1,169,224 | 7.0 |
| NW | North Rhine-Westphalia north | 560,865 | 1,410,076 | 9.9 |
| NO | Lower Saxony, Schleswig-Holstein, Hamburg, Bremen | 655,192 | 1,611,148 | 11.6 |
| HS | Hesse, Saarland, Rhineland-Palatinate | 675,465 | 1,696,054 | 11.7 |
| BY | Bavaria | 1,045,567 | 2,533,612 | 17.2 |

*Shortest Path Computations.* Dijkstra's standard algorithm was compared with the following acceleration methods: node multiway-separator heuristic (sep-heu), arc-flag approach together with a rectangular partition (af-rect) and an arc multiway-separator partition (af-sep), which were computed with METIS [15]. Combining the bi-directed search and the arc-flag method with an arc multiway-separator partition by METIS was the most successful method in our tests with a speed-up of up to 1,400 (af-sep-bi).

In all computations we measured the preprocessing time, the average over all shortest path requests of the shortest path query time, of the length (number of arcs) of the computed shortest path, and the size of the shortest path search space (number of arcs). For the arc-flag method we also measured these parameters separately for the 10% shortest and the 10% longest requests with respect to their shortest path distance.

*Constrained Shortest Path Computations.* In our networks the path length corresponds to travel times and the cost bound to a geographic length bound on the paths. This is motivated by route guidance systems, where the cost corresponds to a given fairness condition. More precisely, the cost bound is determined by a factor times the geographic path length of an $s,t$–path with minimum length. In our experiments we tested factors of 1.05, 1.1 and 1.2. Computational results for constrained shortest path acceleration methods are displayed in Table 4.

## 6 Discussion and Conclusion

*Shortest Path Acceleration Methods.* The Multiway-Separator Approach achieved a speed-up factor of up to 14, see Figure 3. Our multiway-separator heuristic was able to find a smaller multiway-separator on our road network instances than METIS (between 67 % and 85 % of the multiway-separator sizes computed by METIS). The heuristic was also able to improve upon the hierarchical separator methods by Frederickson [5]. But the hierarchical separator method together with our separator heuristic is not among the best acceleration techniques studied here. For our multiway-separator

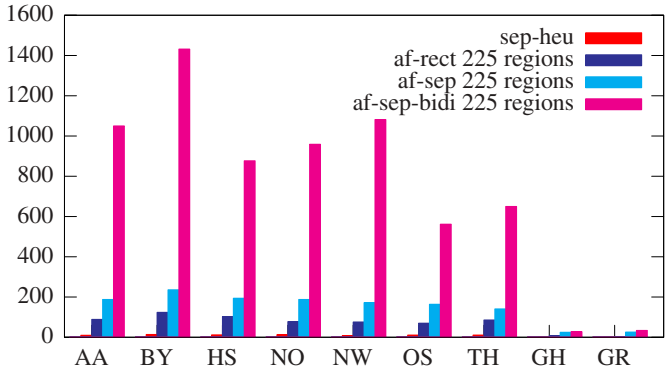



**Fig. 3.** Speed-up factors on all networks compared to the plain Dijkstra algorithm (factor of 1). Results are shown for the node multiway-separator heuristic (sep-heu), the arc-flag with a rectangular partition (af-rect, 225 regions), the arc-flag with an arc multiway-separator partition (af-sep, 225 regions), and the arc-flag with an arc multiway-separator partition combined with bi-directed search (af-sep-bidi, 225 regions)

heuristic typical separator sizes are 1.67% (AA) and 1.34% (OS) of the graph nodes and the resulting regions are of balanced size. The preprocessing time of the hierarchical multiway-separator technique is comparably small, 32 min (AA) and 28 min (OS). The reason for the performance of the hierarchical approach is the number of artificially inserted arcs, which is 18.6% (AA) and 12.6% (OS) of the number of original graph arcs. This is not in line with the aim to reduce the search space in the shortest path computation. Another problem is the huge memory consumption of this method, which is higher than for all other methods we discussed: e.g., 4.5 GB on our largest instance BY. We also studied combinations of the hierarchical multiway-separator approach with both the goal-directed and the bi-directed search, but they only led to small improvements.

**Table 2.** Number of arcs of computed shortest path vs. search space on network OS (474,431 nodes, 1,169,224 arcs). The reduction of the search space is the fraction of the size of the search space vs. the number of arcs in the corresponding shortest path, averaged over all requests. |requests| is the number of computed requests, req.length is the relative length of the request, av. |s.path| is the average number of arcs of the determined shortest paths. Results are shown for the arc-flag method with a rectangular partition (af-rect) and a multiway-separator partition (af-sep) as well as a combination of these two with bi-directed search (af-rect+bidi, af-sep+bidi). All partitions consist of 225 regions. It is remarkable that the arc-flag method combined with a bi-directed search narrows the search space down to almost the size of the shortest path

| network | \|requests\| | req.length | av. \|s.path\| | plain Dijkstra | af-rect | af-sep | af-rect+bidi | af-sep+bidi |
|---|---|---|---|---|---|---|---|---|
| OS | 250 | long | 791 | × 1,313.3 | × 11.0 | × 6.9 | × 1.3 | × 1.2 |
| | 250 | short | 136 | × 571.7 | × 74.0 | × 26.0 | × 19.7 | × 5.4 |
| | 2,500 | all | 437 | × 1,342.8 | × 23.0 | × 11.0 | × 2.4 | × 1.8 |

**Table 3.** Speed-up factors of the arc-flag method with an arc multiway-separator partition with different numbers of regions and combined with a bi-directed search. Speed-up factors are compared to the plain Dijkstra algorithm (factor of 1). Averaged values for 2,500 requests

| network | # nodes | # arcs | plain Dij. | 25 regions | 100 regions | 225 regions | 400 regions | 625 regions |
|---|---|---|---|---|---|---|---|---|
| GR | 14,938 | 32,520 | × 1 | × 8.56 | × 18.4 | × 32.6 | × 38.39 | × 43.7 |
| GH | 53,315 | 109,540 | × 1 | × 11.7 | × 23.2 | × 28.7 | × 31.7 | × 33.4 |
| AA | 362,554 | 920,464 | × 1 | × 260.6 | × 677.6 | × 1,020.5 | × 1,136.9 | × 148.9 |
| OS | 474,431 | 1,169,224 | × 1 | × 190.2 | × 489.6 | × 598.3 | × 722.7 | × 671.7 |

For the arc-flag method there is a clear trade-off between speed-up factor and memory usage. Depending on the chosen partition, one can regard the arc-flag acceleration of shortest path computation as an interpolation between no precomputed information at all (plain Dijkstra) and complete precomputation by determining all possible shortest paths of the graph. Whereas the former is achieved by choosing a partition of the graph into just one region, the latter means partitioning the graph in such a way that a region is given for every single node of the graph. Thus in theory we can get as close as possible to the ideal shortest path search by increasing the number of regions in the partition ('ideal' means that the shortest path algorithm visits only arcs which actually belong to the shortest path itself). Obviously, an increase in the number of regions also entails an increase in preprocessing time and memory consumption (e.g., 625 regions for AA: 5.3 h preprocessing time; 1.6 GB memory).

Using a combination of this method together with other techniques, the best result that we achieved was on the largest instance (BY). The arc flag together with an arc separator partition and combined with a bi-directed search delivers a speed-up factor of 1,400 compared to the plain Dijkstra. In this case we used a partition into 225 regions. Thus we need an extra space of 450 bits ($\approx$ 56 byte) per arc. With just 6 bytes of information per arc (25 regions) the arc-flag method together with a bi-directed search consistently delivered speed-up factors of up to 260 (instance AA, Table 3). This was also the best result for memory consumption vs. speed-up factor. Moreover, Table 3 shows that this method is suitable in particular for larger instances (OS, AA) and long distance requests. For long requests on OS we narrowed the search space down to a factor of 1.2 times the number of shortest path arcs (corresponding to a speed-up factor of 844), while we consistently achieved a factor of around 1.7 for a partition with 225 regions (corresponding to a speed-up factor of 598).

Another important point is the relatively small preprocessing time for our big instances when compared to other preprocessing methods. For example, using the separator partition with 100 regions took us about 2.5 hours (OS) or 2.9 hours (AA). This is about half of what was spent on the rectangular partition. Still, the overall preprocessing time can be reduced even further by using information on shortest path trees which have been computed already for a region during the preprocessing phase.

Although we used this method on road networks with a given embedding in the plane, the question of whether one really needs such an embedding of the graph depends on the partition method that is used. Obviously, for the rectangular partition it is needed, but for an arc multiway-separator partition it is not. The choice of the underlying partition is crucial for the speed-up of this method. Using an arc multiway-separator

partition instead of the rectangular partition results in an additional speed-up factor of 4. The reason for this is the fact that the arc multiway-separator partition determined by the help of METIS adapts much better to the specific structure of the network.

As for combinations of acceleration methods, the bi-directed search seems to be a perfect match for the arc-flags (additional speed-up factors of up to 7). The goal-directed search is less useful for the arc-flag method, since the method is already goal-directed by construction. Typically, the arc-flag method creates a cone-like spreading of the search space as it approaches the target region. In fact, at the beginning of a shortest path search the algorithm is forced by the arc-flags to walk along shortest path arcs only. Just before the target region is reached we can observe the spreading that was described above (see Figure 2). To cope with this behavior of the arc-flag method and in order to improve the algorithms even further, we suggest to study 2-level partitions; a coarse partition for far away target nodes and a finer one for nearby nodes. An extensive investigation of the speedup that can be obtained along these lines is presented in [16].

*Constrained Shortest Path Acceleration Methods.* In Section 4 we explained how to adapt well-known acceleration techniques for shortest path computations to the constrained shortest path search. Here the goal-directed search yields the best results, but the bi-directed search still delivers good accelerations. The advantage of the bi-directed search is that it does not require any additional preprocessing. The combined version (goal- and bi-directed) suffers from the lack of a good stopping criterion. See Table 4 for computational results of these methods.

The preprocessing phase of our methods is comparably short: on the Berlin road network for the goal-directed search the preprocessing takes up to 161 seconds and for the combined version up to 339 seconds. In the combined case the preprocessing requires two shortest path tree computations to compute the lower bounds from the target node to all other nodes in the graph. On hard instances where a large number

**Table 4.** Constrained shortest path acceleration methods on the Berlin road network (12,100 nodes, 19,570 arcs). Results are shown for the plain generalized Dijkstra (plain), the goal-directed search (go), the bi-directed search (bi), and the combination (go-bi). c. fact. is the constrained factor, av. |s.path| is the average length of the determined shortest paths, |search space| is the size of the search space (number of arcs), max. |label list| is the maximum size of a node label list during a computation, prepro. time is the preprocessing time in seconds, comp. time is the computation time in seconds, and speed-up fact. is the speed-up factor compared to the plain generalized Dijkstra algorithm (factor of 1). Computed values are on average for 1,000 requests, on a Itanium II processor (1.2 GHz)

| method | c. fact. | av. \|s.path\| | \|search space\| | max. \|label list\| | prepro. time [s] | comp. time [s] | speed-up fact. |
|---|---|---|---|---|---|---|---|
| plain | 1.2 | 68 | 80,803 | 23 | 0 | 1,212 | × 1 |
| go | 1.2 | 68 | 1,484 | 24 | 161 | 13 | × 93 |
| bi | 1.2 | 68 | 16,045 | 20 | 0 | 231 | × 5 |
| go-bi | 1.2 | 68 | 18,401 | 39 | 339 | 197 | × 6 |
| plain | 1.05 | 76 | 95,241 | 23 | 0 | 1,534 | × 1 |
| go | 1.05 | 76 | 1,847 | 26 | 154 | 14 | × 110 |
| bi | 1.05 | 76 | 29,447 | 25 | 0 | 496 | × 3 |
| go-bi | 1.05 | 76 | 4,793 | 26 | 323 | 37 | × 42 |

of labels is created during the search process, the preprocessing time can be neglected compared to the overall processing time of a non-accelerated generalized Dijkstra run. A further advantage of the goal-directed search is the possibility to have more than one Pareto-optimal path computed within one run. This is of particular importance for applications such as the routing project mentioned before.

## Acknowledgment

We would like to thank Benjamin Wardemann and Ronald Wotzlaw for their help with the implementation and the group of Dorothea Wagner at the University of Karlsruhe for providing access to Opteron machines.

## References

1. Aneja, Y.P., Aggarwal, V., Nair, K.P.K.: Shortest chain subject to side constraints. Networks **13** (1983) 295–302
2. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co. (2000)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Mathematik (1955) 269–271
4. Dumitrescu, I.: Constrained path and cycle problems. PhD thesis, The University of Melbourne (2002)
5. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. **16** (1987) 1004–1022
6. Goldberg, A.V., Harrelson, C.: Computing the shortest path: $A^*$ search meets graph theory. In: Proc. of the 16th Annual ACM-SIAM Symp. on Discrete Algorithms. (2005) 156–165
7. Goodrich, M.T.: Planar Separators and Parallel Polygon Triangulation. Journal of Computer and System Sciences **51** (1995) 374–389
8. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: Proc. of the 6th ALENEX. (2004) 100–111
9. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In Ribeiro, C.C., Martins, S.L., eds.: Experimental and Efficient Algorithms: 3rd International Workshop, (WEA 2004). Volume 3059 of LNCS., Springer (2004) 269–284
10. Jahn, O., Möhring, R.H., Schulz, A.S., Moses, N.E.S.: System optimal routing of traffic flows with user constraints in networks with congestion. Oper. Res. (2005) to appear.
11. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. Technical Report Report-042-2004, TU Berlin (2004)
12. Lauther, U.: Slow preprocessing of graphs for extremely fast shortest path calculations (1997) Lecture at the Workshop on Computational Integer Programming at ZIB.
13. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In Raubal, M., Sliwinski, A., Kuhn, W., eds.: Geoinformation und Mobilität. Volume 22 of IfGI prints., Institut für Geoinformatik, Münster (2004) 22–32
14. Mehlhorn, K., Ziegelmann, M.: Resource constrained shortest paths. In: Proc. 8th European Symposium on Algorithms. LNCS 1879, Springer (2000) 326–337
15. METIS: A family of multilevel partitioning algorithms (2003) http://www-users.cs.umn.edu/∼karypis/metis.

16. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. LNCS, Springer (2005) this volume.
17. Müller-Hannemann, M., Schnee, M.: Finding all attractive train connections by multi-criteria pareto search. In: 4th Workshop on Algorithmic Methods and Models for Optimization of Railways. (2004) to appear.
18. Schulz, F., Wagner, D., Zaroliagis, C.: Using multi-level graphs for timetable information in railway systems. In: Proc. of the 4th ALENEX 2002. LNCS 2409, Springer (2002) 43–59

# A General Buffer Scheme for the Windows Scheduling Problem

Amotz Bar-Noy[1], Jacob Christensen[2], Richard E. Ladner[2], and Tami Tamir[3]

[1] Computer and Information Science Department, Brooklyn College, 2900 Bedford Avenue Brooklyn, NY 11210
amotz@sci.brooklyn.cuny.edu
[2] Department of Computer Science and Engineering, Box 352350,University of Washington,Seattle, WA 98195
{jacoblc, ladner}@cs.washington.edu
[3] School of Computer Science, The Interdisciplinary Center, Herzliya, Israel
tami@idc.ac.il

**Abstract.** Broadcasting is an efficient alternative to unicast for delivering popular on-demand media requests. Windows scheduling algorithms provide a way to satisfy all requests with both low bandwidth and low latency. Consider a system of $n$ pages that need to be scheduled (transmitted) on identical channels an infinite number of times. Time is slotted, and it takes one time slot to transmit each page. In the *windows scheduling problem* (WS) each page $i$, $1 \leq i \leq n$, is associated with a *request window* $w_i$. In a feasible schedule for WS, page $i$ must be scheduled at least once in any window of $w_i$ time slots. The objective function is to minimize the number of channels required to schedule all the pages. The main contribution of this paper is the design of a general *buffer scheme* for the windows scheduling problem such that *any* algorithm for WS follows this scheme. As a result, this scheme can serve as a tool to analyze and/or exhaust all possible WS-algorithms. The buffer scheme is based on modelling the system as a nondeterministic finite state channel in which any directed cycle corresponds to a legal schedule and vice-versa. Since WS is NP-hard, we present some heuristics and pruning-rules for cycle detection that ensure reasonable cycle-search time.

By introducing various rules, the buffer scheme can be transformed into deterministic scheduling algorithms. We show that a simple page-selection rule for the buffer scheme provides an optimal schedule to WS for the case where all the $w_i$'s have divisible sizes, and other good schedules for some other general cases. By using an exhaustive-search, we prove impossibility results for other important instances.

We also show how to extend the buffer scheme to more generalized environments in which (*i*) pages are arriving and departing on-line, (*ii*) the window constraint has some *jitter*, and (*iii*) different pages might have different lengths.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 139–151, 2005.
© Springer-Verlag Berlin Heidelberg 2005

## 1 Introduction

Currently, popular on-demand data on the Internet is provided in a unicast way, by requesting it from a server. Such systems are called *pull* systems. A very high demand over a short period of time may put stress on both server and network bandwidth. This stress can be alleviated by replicating data in mirrors or caches. An alternative approach to on-demand for popular data is a *push* system where the data is provided by periodic broadcast or multicast. Those desiring and authorized to receive the data simply wait, hopefully a short period of time, for the broadcast. Pushing has the advantage over pulling in that it requires less server and network bandwidth, as long as the demand is high. This approach to providing popular data has led to a very interesting problem. What are the best ways to partition the channel in a time multiplexed way to provide the service in a push system? This general question can be modelled mathematically in a number of ways. We choose a specific approach called *windows scheduling (WS)* [5, 6]. In this paper, we propose a new algorithmic technique called the *buffer scheme* that can be used to design algorithms to solve WS problems and several extensions of WS that cannot be solved using known algorithms. In addition, the buffer scheme can be used to prove new impossibility results.

An instance to WS is a sequence $\mathcal{W} = \langle w_1, \ldots, w_n \rangle$ of $n$ request windows, and a set of $h$ identical channels. The window request $w_i$ is associated with a page $i$. Time is slotted, and it takes one time slot to transmit any page on any channel. The output is a feasible schedule (a schedule in short) in which for all $i$, the page $i$ must be scheduled (transmitted) on one of the $h$ channels at least once in any window of $w_i$ consecutive time slots. Equivalently, the requirement is that the *gap* between any two consecutive appearances of $i$ in the schedule is at most $w_i$. We say that a schedule is *perfect* if the gap between any two consecutive appearances of $i$ in the schedule is a constant $w'_i$ for some $w'_i \leq w_i$.

The optimization problem associated with WS is to minimize the number of channels required to schedule all $n$ pages. Define $1/w_i$ as the *width* of page $i$ and let $h_0(\mathcal{W}) = \lceil \sum_i 1/w_i \rceil$. Then $h_0(\mathcal{W})$ is an obvious lower bound on the minimum number of channels required for $\mathcal{W}$.

*Example I:* An interesting example is that of *harmonic scheduling*, that is, scheduling sequences $\mathcal{H}_n = \langle 1, 2, \ldots n \rangle$ in a minimum number of channels. Harmonic windows scheduling is the basis of many popular media delivery schemes (e.g., [21, 15, 16, 18]). The following is a non-perfect schedule of 9 pages on 3 channels for the window sequence $\mathcal{H}_9 = \langle 1, 2, \ldots, 9 \rangle$.

$$\begin{bmatrix} 1\,4\,1\,1\,1\,1\,1\,6\,1\,1\,1\,1\cdots \\ 2\,1\,2\,5\,2\,4\,2\,5\,2\,4\,2\,5\cdots \\ 3\,6\,7\,3\,8\,9\,3\,1\,7\,3\,9\,8\cdots \end{bmatrix}$$

Note that a page may be scheduled on different channels (e.g., 1 is scheduled on all three channels). Also, the gaps between any two consecutive appearances of $i$ need not be exactly $w_i$ or another *fixed* number (e.g., the actual window granted

to 5 is 4 and the actual windows of 8 and 9 are sometimes 5 and sometimes 7). Even though this schedule is not "nicely" structured, it is feasible since it obeys the requirement that the maximal gap between any two appearances of $i$ is at most $w_i$ for any $i$. Using our buffer scheme in exhaustive search mode, we show that there is no schedule for $\mathcal{H}_{10} = \langle 1, 2, \ldots, 10 \rangle$ on three channels even though $\sum_{i=1}^{10} 1/i < 3$.

*Example II:* In this paper we demonstrate that for some instances such "flexible" schedules achieve better performance. Indeed, for the above example, there exists a perfect feasible schedule on three channels. However for the following instance this is not the case. Let $n = 5$ and $\mathcal{W} = \langle 3, 5, 8, 8, 8 \rangle$. We show in this paper that there is no feasible perfect schedule of these 5 pages on a single channel. However,

$$[3, 5, 8_a, 3, 8_b, 5, 3, 8_c, 8_a, 3, 5, 8_b, 3, 8_c, 5, 3, 8_a, 8_b, 3, 5, 8_c, \ldots]$$

is a feasible non-perfect schedule on a single channel. This schedule was found by efficiently implementing the buffer scheme. Most previous techniques only produce perfect schedules.

### 1.1 Contributions

The main contribution of this paper is the design of a general *buffer scheme* for the windows scheduling problem. We show that *any* algorithm for WS follows this scheme. Thus, this scheme can serve as a tool to analyze all WS-algorithms. The buffer scheme is based on presenting the system as a nondeterministic finite state machine in which any directed cycle corresponds to a legal schedule and vice-versa. The state space is very large, therefore we present some heuristics and pruning-rules to ensure reasonable cycle-search time.

By introducing various rules for the buffer scheme, it can be transformed into deterministic scheduling algorithms. We show that a simple greedy rule for the buffer scheme provides an optimal schedule to WS for the case where all the $w_i$'s have divisible sizes. Our theoretical results are accompanied by experiments. We implemented the deterministic buffer scheme with various page selection rules. The experiments show that for many instances the deterministic schemes perform better than the known greedy WS algorithm presented in [5].

By using an exhaustive-search, we prove impossibility results and find the best possible schedules. As mentioned earlier, we prove that there is no schedule of $\mathcal{H}_{10} = \langle 1, 2, \ldots, 10 \rangle$ on three channels. In addition, we find the best possible schedules for other important instances. Similar to branch and bound, the search is done efficiently thanks to heavy pruning of early detected dead-ends. The results achieved in the exhaustive-search experiments appear not to be achievable in any other way.

The main advantage of the buffer scheme is its ability to produce non-perfect schedules. Most of the known algorithms (with or without guaranteed performance) produce perfect schedules. However, in the WS problem and its applications such a restriction is not required. We demonstrate that the Earliest

Deadline First (EDF) strategy is not the best for WS even though it optimal for similar problems. We develop some understanding that leads us to the design of the Largest Backward Move (LBM) strategy that performs well in our simulations.

The basic windows scheduling problem can be generalized in several ways that cannot be handled by previous techniques that only produce perfect schedules. (i) *Dynamic (on-line) environment:* pages are arriving and departing on-line and the set of windows is not known in advance. Here the scheme is extended naturally emphasizing its advantage as a framework to algorithms as opposed to other greedy heuristics for the off-line setting that cannot be generalized with such an ease. (ii) *Jitter windows:* each page is given by a pair of windows $(w'_i, w_i)$ meaning that page $i$ needs to be scheduled *at least* once in any window of $w_i$ time slots and *at most* once in any window of $w'_i$ time slots. In the original definition, $w'_i = 1$. Here again the generalization is natural. (iii) *Different lengths:* pages might have different lengths. The buffer scheme can be generalized to produce high quality schedules in these generalizations.

### 1.2 Prior Results and Related Work

The windows scheduling problem belongs to the class of *periodic scheduling* problems in which each page needs to be scheduled an infinite number of times. However, the optimization goal in of the windows scheduling problem is of the "max" type whereas traditional optimization goals belong to the "average" type. That is, traditional objectives insist that each page $i$ would receive its required share $(1/w_i)$ even if some of the gaps could be larger than $w_i$. The issue is usually to optimize some fairness requirements that do not allow the gaps to be too different than $w_i$. Two examples are *periodic scheduling* [17] and *the chairman assignment problem* [20]. For both problems the Earliest Deadline First strategy was proven to be optimal. Our paper demonstrates that this is not the case for the windows scheduling problem.

The pinwheel problem is the windows scheduling problem with one channel. The problem was defined in [13, 14] for unit-length pages and was generalized to arbitrary length pages in [8, 12]. In these papers and other papers about the pinwheel problem the focus was to understand which inputs can be scheduled on one channel. In particular, the papers [10, 11] optimized the bound on the value of $\sum_{i=1}^{n}(1/w_i)$ that guarantees a feasible schedule.

The windows scheduling problem was defined in [5], where it is shown how to construct perfect schedules that use $h_0(\mathcal{W}) + O(\log h_0(\mathcal{W}))$ channels. This asymptotic result is complemented with a practical greedy algorithm, but no approximation bound has been proved for it yet. Both the asymptotic and greedy algorithms produce only perfect schedules.

The general WS problem can be thought of as a scheduling problem for push broadcast systems (e.g, Broadcast Disks ([1]) or TeleText services ([2])) In such a system there are clients and servers. The server chooses what information to push in order to optimize the quality of service for the clients (mainly the response time). In a more generalized model the servers are not the information

providers. They sell their service to various providers who supply content and request that the content be broadcast regularly. The regularity can be defined by a window size. Finally, various maintenance problems where considered with similar environments and optimization goals (e.g., [22, 3]).

WS is known to be NP-hard. In a way, this justifies the efforts of this paper. A proof for the case where $i$ must be granted an exact $w_i$ window is given in [4]. Another proof which is suitable also for the flexible case in which the schedule of $i$ need not be perfect is given in [7].

## 2 The General Buffer Scheme

In this section, we describe the buffer scheme and prove that for any instance of windows scheduling, any schedule can be generated by the buffer scheme. We then discuss how the buffer scheme can be simulated efficiently by early detection and pruning of dead-end states. Using these pruning rules, we establish an efficient implementation of the scheme that can exhaust all possible solutions. For big instances, for which exhaustive search is not feasible, we suggest a greedy rule that produces a single execution of the scheme that "hopefully" generates a correct infinite schedule.

### 2.1 Overview of the Scheme

Let $\mathcal{W} = \langle w_1, \ldots, w_n \rangle$ and number of channels $h$ be an instance of the windows scheduling problem. Let $w^* = \max_i \{w_i\}$. We represent the pages state using a set of buffers, $B_1, B_2, \ldots, B_{w^*}$. Each page is located in some buffer. A page located in $B_j$ must be transmitted during the next $j$ slots. Initially, buffer $B_j$ includes all the pages with $w_i = j$. We denote by $b_j$ the number of pages in $B_j$ and by $\ell_i$ the *location* of $i$ (i.e., $i \in B_{\ell_i}$).

In each iteration, the scheme schedules at most $h$ pages on the $h$ channels. By definition, the pages of $B_1$ must be scheduled. In addition, the scheme selects at most $h - b_1$ additional pages from other buffers to be scheduled in this iteration. The way these pages are selected is the crucial part of the scheme and is discussed later. After selecting the pages to be scheduled, the scheme updates the content of the buffers.

- For all $j > 1$, all the *non-scheduled* pages located in $B_j$ are moved to $B_{j-1}$.
- Each scheduled page, $i$, is placed in $B_{w_i}$ - to ensure that the next schedule of $i$ will be during the next $w_i$ slots.

This description implies that the space complexity of the buffer scheme depends on $w^*$. However, by using a data structure that is 'page-oriented', the buffer scheme can be implemented in space $O(n)$.

From the pages' point of view, a page is first located as far as possible ($w_i$ slots) from a deadline (represented by $B_1$), it then gets closer and closer to the deadline and can be selected to be transmitted in any time during this advancement toward the deadline. With no specific rule for selecting which of the $h - b_1$

pages should be scheduled, the buffer scheme behaves like a nondeterministic finite state machine with a very large state space, where a state is simply an assignment of pages to buffers.

In running the buffer scheme nondeterministically, it fails if in some time point $b_1 > h$. The scheme is successful if it produces an infinite schedule. This is equivalent to having two time slots $t_1, t_2$ such that the states at $t_1$ and $t_2$ are identical. Given these two time slots, the page-selection sequence between $t_1$ and $t_2$ can be repeated forever to obtain an infinite schedule.

**Theorem 1.** *When it does not fail, the buffers scheme produces a feasible schedule, and any feasible schedule for WS can be produced by an execution of the buffer scheme.*

**Remark:** In our simulations and in the page-selection rules we suggest, no channel is 'idle' in the execution; that is, exactly $h$ pages are scheduled in each time slot. It is important to observe that this no-idle policy is superior over scheduling policies that allow idles.

### 2.2 Page Selection Criteria and Dead-Ends Detection

As mentioned above, the buffer scheme fails if at some time point $b_1 > h$, that is, more than $h$ pages must be scheduled in the next time slot. However, we can establish other, more tight, dead-end conditions. Then, by trying to avoid these dead-ends, we can establish "good" page selection criteria. In this section, we present a tight dead-end criteria, and describe how to greedily select pages in each time slot in a way that delays (and hopefully avoids) a dead-end state. Given a state of the buffers, let $c(i, j)$ denote the number of times $i$ must be scheduled during the next $j$ slots in any feasible schedule.

*Claim.* For any $i, j$,

$$c(i,j) \geq \begin{cases} 0 & \text{if } j < \ell_i \\ 1 + \left\lfloor \frac{j-\ell_i}{w_i} \right\rfloor & \text{if } j \geq \ell_i \end{cases}$$

*Proof.* If $j < \ell_i$, that is, if $i$ is located beyond the first $j$ buffers, we do not need to schedule $i$ at all during the next $j$ slots. If $j \geq \ell_i$, then we must schedule $i$ once during the next $\ell_j$ slots. After this schedule, $i$ will be located in $B_{w_i}$. Note that for any $t$, given that $i \in B_{w_i}$ we must schedule $i$ at least $\lfloor t/w_i \rfloor$ times during the next $t$ slots. In our case, we have $t = j - \ell_i$, since this is the minimal number of slots that remains after the first schedule of $i$.

For example, if $\ell_i = 1$, $w_i = 3$ and $j = 11$, then $c(i, j) = 4$. This implies that $i$ must be scheduled at least 4 times during the next 11 slots: once in the next slot, and three more times in the remaining 10 slots. Let $c(j)$ denote the total number of page schedules the system must provide in the next $j$ slots. By definition, $c(j) = \sum_{i=1}^{n} c(i, j)$. By definition, $jh$ is the number of available page schedules in the next $j$ slots. Let $f(j) = jh - c(j)$ denote the *freedom level* existing in the next $j$ slots.

If for some $j$, $f(j) < 0$ then a dead-end state is reached. If $f(j) = 0$, then only pages from the first $j$ buffers must be scheduled in the next $j$ slots. If $f(j) > 0$, then some freedom exists in the way the pages are selected. That is, $c(j)$ pages must be selected from the first $j$ buffers, and the remaining $f(j)$ pages can come from any buffer. In particular, for $j = 1$, only the pages in $B_1$ are considered, thus, this rule generalizes the obvious condition for $B_1$.

Importantly, it is possible to know how many pages must be selected from the first $j$ buffers *in the next slot*. For any $j$, the system can provide at most $(j-1)h$ page-schedules during any $j-1$ slots. Thus, at least $n(j) = c(j)-(j-1)h$ pages from the first $j$ buffers must be selected in the next slot in order to avoid a dead-end. Again, this condition generalizes the condition for $B_1$.

### 2.3 Delaying Dead-Ends and Deterministic Rules

We present a greedy way to select the pages to be scheduled based on the parameters $c(j)$ and $n(j)$ that are calculated during the selection process. Let $s$ denote the number of pages selected so far in the current iteration. Initially, $j = 1$ and $s = 0$. As long as $s < h$, continue selecting pages as follows. For each $j$, if $n(j) > h$ the selection process fails. If $n(j) = s$, there are no constraints due to $B_j$ (since $s$ pages have already been selected from the first $j$ buffers) and the selection proceeds to $j + 1$. Otherwise ($s < n(j) \leq h$), select from the first $j$ buffers $n(j) - s$ pages that were not selected yet, and proceed to $j+1$. Note that this scheme is still nondeterministic because we have not yet specified exactly which pages are scheduled. We call this scheme the *restricted buffer scheme*.

**Theorem 2.** *Any legal schedule for WS can be generated by the restricted buffer scheme.*

We now give some deterministic rules for deciding exactly which pages to schedule in a restricted buffer scheme. In applying the restricted buffer scheme, it must determine, given a specific $k$ and $j$, which $k$ pages from the first $j$ buffers are to be scheduled in the next time slot. Naturally, high priority is given to pages whose transmission will reduce the most the *load* on the channels.

This load can be measured by a potential function based on the locations of the pages. We suggest two greedy selection rules, each of them maximizes a different potential function. Our first greedy rule is suitable for the potential function $\phi_1 = \sum_i \ell_i$. Our second greedy rule is suitable for the potential function $\phi_2 = \sum_i \ell_i/w_i$. These two approaches are realized by the following rules:

1. Select pages for which $w_i - \ell_i$ is maximal.
2. Select pages for which $(w_i - \ell_i)/w_i$ is maximal.

In the first rule, denoted LBM (*Largest Backward Move*), pages that can increase $\phi_1$ the most are selected. In LBM, pages that will move the most are scheduled first. In the second rule, denoted WLBM (*weighted LBM*), the pages that increase $\phi_2$ the most are selected. Each of these rules can be applied when ties are broken in favor of pages associated with smaller windows or larger windows. Our simulations reveal that breaking ties in favor of pages with small

windows performs better for almost all inputs. On the other hand, we cannot crown any of these two rules as the ultimate winner. For the first rule we show that it is optimal for a large set of instances, even without the dead-end detection of the restricted buffer scheme. The second rule performs better on large harmonic instances. For both rules, the simulations give good results (see Section 3).

In our simulations, a third natural greedy rule is considered, *Earliest Deadline First*, in which the pages with minimal $\ell_i$ are selected. This rule is optimal for other periodic scheduling problems that care about average gaps (e.g., periodic scheduling [17] and the chairman assignment problem [20]). However, in our problem this rule performs poorly. This can be explained by the fact that deadlines are well considered by the dead-end detection mechanism of the restricted buffer scheme. The role of the additional page selection is to reduce future load on the channels.

### 2.4 The LBM Selection Rule

Let LBM be the buffer scheme with the greedy rule that prefers pages with large $(w_i - \ell_i)$ and breaks ties in favor of pages with smaller windows. We show that LBM is optimal for a large set of instances even without the dead-end detection mechanism of the restricted buffer scheme. Without dead-end detection, LBM runs as follows:

1. Initialization: Put $i$ in buffer $B_{w_i}$ for all $1 \leq i \leq n$.
2. In each time slot:
   (a) If $b_1 > h$ then terminate with a failure.
   (b) Otherwise, schedule all the pages from $B_1$.
   (c) If $h > b_1$, select $h - b_1$ additional pages with the largest $(w_i - \ell_i)$, break ties in favor of pages with smaller windows.

*Optimality for Divisible-size Instances:*

**Definition 1.** *An instance $\mathcal{W}$ of WS is a divisible-size instance, if $w_{i+1}$ divides $w_i$ in the sorted sequence of windows $w_1 \geq \cdots \geq w_i \geq w_{i+1} \geq \cdots \geq w_n$ for all $1 \leq i < n$.*

For example, an instance in which all the windows are powers of 2 is a divisible-size instance. The divisible-size constraint is not unreasonable. For example, pages could be advertising slots which are only offered in windows that are powers of 2, in a way that magazines sell space only in certain fractions, 1/2 page, 1/4 page, and so on. The following Theorem proves that LBM is optimal for divisible-size instances.

**Theorem 3.** *If an instance, $\mathcal{W}$, of WS is a divisible-size instance and $h \geq h_0(\mathcal{W})$, then LBM never fails.*

## 3 Deterministic Rules Experiments

We simulated the buffer scheme with the deterministic page-selection rules given in Section 2.3. The performance of the buffer scheme, measured by the number of channels required to schedule the pages, was compared for each instance, $\mathcal{W}$, with the lower bound $h_0(\mathcal{W})$ and with the number of channels required by the greedy algorithm, Best-Fit Increasing (*BFI*), given in [5]. The algorithm BFI schedules the pages in non-decreasing order of their window request. Page $i$ with window request $w_i$ is assigned to a channel that can allocate to it a window $w'_i$ such that $w_i - w'_i$ is non-negative and minimal. In other words, when scheduling the next page, BFI tries to minimize the lost width $(1/w'_i - 1/w_i)$. Note that BFI produces only perfect schedules.

In our simulations we considered several classes of instances. In this extended abstract we report about two of them:
$(i)$ Random - Sequences generated randomly, $w_i$ is chosen randomly in $2, \ldots, 500$ according to the following distribution. Let $S = \sum_{i=2}^{500} i$ then the probability of choosing $w_i = i$ is $i/S$. The simulation results for random instances are shown in Figure 1. The same set of randomly chosen pages was scheduled by the greedy BFI algorithm, by the buffer scheme using the LBM rule and by the buffer scheme using the weighted LBM rule. It can be seen that the buffer scheme always performs better, or not worse, than the greedy algorithm. Also, the buffer scheme is always within one channel from the lower bound (given by $h_0(\mathcal{W})$).
$(ii)$ Harmonic - $\mathcal{H}_n = \langle 1, 2, \ldots, n \rangle$. The simulation results for Harmonic instances is shown in Figure 2. For each number of channels $h = 2, \ldots, 8$ and for each rule, the maximal $n$ such that $\mathcal{H}_n$ is scheduled successfully is presented. For these instances, the algorithm BFI performs better than any of the deterministic rules of the buffer scheme. The differences though are not significant. In particular, for any harmonic sequence, none of the rules failed on $h_0(\mathcal{W}) + 1$ channels.

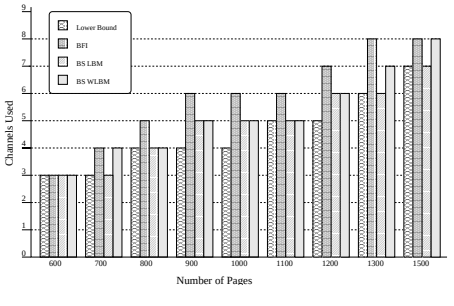



**Fig. 1.** Simulation results for random instances

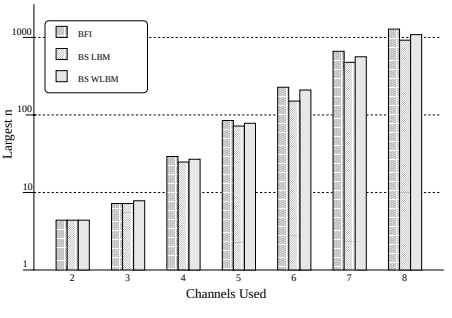



**Fig. 2.** Simulation results for harmonic instances

## 4 The Exhaustive Buffer Scheme

In this section, we demonstrate the usefulness of the buffer scheme for practical cases for which it is possible to run an efficient implementation of the scheme that exhausts all possible solutions. Dead-end detection is integrated in the search. It enables early pruning of dead-end states and ensures reasonable cycle-search time. We use the scheme to find the best schedules for some instances and to prove non-trivial impossibility results for other instances.

To obtain our results, we reduce the problem of finding a schedule based on the buffer scheme to the problem of detecting a directed cycle in a finite directed graph. This problem can be solved using standard Depth First Search (DFS). Consider the directed graph $G$ in which each vertex represents a possible state of the buffers, and there is an edge from $v_1$ to $v_2$ if and only if it is possible to move from the state represented by $v_1$ to the state represented by $v_2$ in one time slot - that is, by scheduling $h$ pages (including all the pages of $B_1$) and updating the resulting page locations as needed. Note that $G$ is finite since the number of pages is finite and each page has a finite number of potential locations. Now use a standard DFS to detect if there is a directed cycle. If a cycle is detected, then this cycle induces an infinite schedule. If no directed cycle exists, by Theorem 1, there is no schedule.

*Windows Scheduling for Broadcasting Schemes:* The buffer scheme can find for small values of $n$ the minimal $d$ such that there exists a schedule of the instance $\mathcal{W} = \langle d, \ldots, d+n-1 \rangle$ on $h$ channels. These instances are of special interest for the media-on-demand application since a schedule of $\mathcal{W}$ would imply a broadcasting scheme for $h$ channels with delay guaranteed at most $d/n$ of the media length (using the shifting technique presented in [6]). In this scheme, the transmission is partitioned into $n$ segments. The trade-off is between the number of segments and the delay. Table 1 summarizes our simulation results for $n = 5, 6, 7, 8$ segments and a single channel. For each $5 \leq n \leq 8$, we performed an efficient exhaustive search over all possible executions of the buffer scheme. While for some values of $n$ the optimal schedules are perfect and can be generated by simple greedy heuristics, for other values of $n$, the non-perfect schedules produced by the buffer scheme are the only known schedules. This indicates that for some values of $n$ and $d$ the best schedule is not perfect. No existing technique can produce such schedules.

To illustrate that optimal schedules might be non-structured, we present the optimal one-channel schedule for $\langle 5, \ldots, 11 \rangle$. No specific selection rule was ap-

**Table 1.** Some best possible schedules for small number of segments

| # of segments | best range | delay |
|---|---|---|
| 5 | 4..8 | $4/5 = 0.8$ |
| 6 | 5..10 | $5/6 = 0.833$ |
| 7 | 5..11 | $5/7 = 0.714$ |
| 8 | 6..13 | $6/8 = 0.75$ |

plied to produce this schedule, it was generated by exhaustive search over the non-deterministic execution of the buffer scheme. $[10, 9, 7, 5, 8, 6, 9, 11, 5, 7, 10, 6, 8, 5, 11, 9, 7, 6, 5, 8, 10, 6, 7, 5, 9, 11, 6, 8, 5, 7, 10, 9, 6, 5, 7, 8, 11, 5, 6]$.

*Impossibility Results:* Using the buffer scheme, we were able to solve an open problem from [5] by proving that no schedule exists on three channels for the instance $\mathcal{H}_{10} = \langle 1, \ldots, 10 \rangle$ even though $\sum_{i=1}^{10} 1/i < 3$. Using the early detection of dead-ends we able to reduce the search proving impossibility from 3,628,800 states to only 60,000 states. Using similar techniques we determined that there are no one channel schedules for any of the sequences $\langle 3..7 \rangle$, $\langle 4..9 \rangle$, $\langle 4..10 \rangle$, and $\langle 5..12 \rangle$. This means that the ranges given in the Table 1 are optimal.

*Arbitrary Instances:* Most of the previous algorithms suggested for WS produce perfect schedules. The buffer scheme removes this constraint. We demonstrate this by the following, one out of many, example. Consider the instance $\mathcal{W} = \langle 3, 5, 8, 8, 8 \rangle$. Using the fact that $gcd(3, 8) = gcd(3, 5) = 1$, it can be shown that there is no perfect schedule for $\mathcal{W}$ on a single channel. The exhaustive search and the deterministic buffer scheme with LBM produce the following non-perfect schedule for $\mathcal{W}$:

$$[3, 5, 8_a, 3, 8_b, 5, 3, 8_c, 8_a, 3, 5, 8_b, 3, 8_c, 5, 3, 8_a, 8_b, 3, 5, 8_c, \ldots].$$

We could not find any special pattern or structure in this schedule, suggesting that the only non-manual way to produce it is by using the buffer scheme.

## 5 Extensions to Other Models

We show how the buffer scheme paradigm can be extended to more general environments. As opposed to other known heuristics for WS, the first two extensions are simple and natural.

*Dynamic Window Scheduling:* In the dynamic (on-line) version of WS, pages arrive and depart over time [9]. This can be supported by the buffer scheme as follows: (*i*) Any arriving page with window $w_i$ is placed upon arrival in $B_{w_i}$. (*ii*) Any departing page is removed from its current location. The number $h$ of active channels can be adjusted according to the current load. That is, add a new channel whenever the current total width is larger than some threshold (to be determined by the scheme), and release some active channels whenever the current total load is smaller than some threshold.

*Window Scheduling with Jitter:* In this model, each page is associated with a pair of window sizes $(w'_i, w_i)$ meaning that $i$ needs to be scheduled *at least* once in any window of $w_i$ time slots, and *at most* once in any window of $w'_i$ time slots. That is, the gap between consecutive appearances of $i$ in the schedule must be between $w'_i$ and $w_i$. In the original WS, $w'_i = 1$ for all $1 \leq i \leq n$. In the other extreme, in which $w'_i = w_i$, only perfect schedules are feasible and the gap

between any two appearances of $i$ in the schedule is exactly $w_i$. To support such instances with a buffer scheme, we modify the page-selection rules as follows: $(i)$ After scheduling $i$, put it in buffer $B_{w_i}$. $(ii)$ Page $i$ can be selected for scheduling only if it is currently located in one of the buffers $B_1, B_2, \ldots, B_{w_i - w'_i + 1}$. This ensures that at least $w'_i$ slots have passed since the last time $i$ was scheduled. The first selection of $i$ can be from any buffer.

*Pages with Different Lengths:* In this model, each page is associated with a window $w_i$ and with a length $p_i$. Page $i$ needs to be allocated at least $p_i$ transmission slots in any window of $w_i$ slots. Clearly, $p_i \leq w_i$ for all $1 \leq i \leq n$, otherwise it is impossible to schedule this page. We consider *non-preemptive* windows scheduling in which for any $i$, the $p_i$ slots allocated to $i$ must be successive. In other words, $i$ must be scheduled non-preemptively on the channels and the gap between any two beginnings of schedules is at most $w_i$.[1] To support pages with different lengths, each $i$ is represented as a chain of $p_i$ page-segments of length 1. Due to lack of space we do not give here the full details of how these page segments are selected one after the other.

## References


1. S. Acharya, M. J. Franklin, and S. Zdonik. Dissemination-based data delivery using broadcast disks. *IEEE Personal Communications*, Vol. 2, No. 6, 50-60, 1995.
2. M. H. Ammar and J. W.Wong. The design of teletext broadcast cycles. *Performance Evaluation*, Vol. 5, No. 4, 235-242, 1985.
3. S. Anily, C. A. Glass, and R. Hassin. The scheduling of maintenance service. *Discrete Applied Mathematics,* Vol. 82, 27-42, 1998.
4. A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber. Minimizing service and operation costs of periodic scheduling. *Mathematics of Operations Research,* Vol. 27, No. 3, 518-544, 2002.
5. A. Bar-Noy and R. E. Ladner. Windows scheduling problems for broadcast systems. *SIAM Journal on Computing (SICOMP),* Vol. 32, No. 4, 1091-1113, 2003.
6. A. Bar-Noy, R. E. Ladner, and T. Tamir. Scheduling techniques for media-on-demand. *Proc. of the 14-th SODA,* 791-800, 2003.
7. A. Bar-Noy, R. E. Ladner, and T. Tamir. Windows scheduling as a restricted bin-packing problem. *Proc. of the 15-th SODA,* 217-226, 2004.
8. S. K. Baruah S-S. Lin. Pfair Scheduling of Generalized Pinwheel Task Systems *IEEE Trans. on Comp.,* Vol. 47, 812–816, 1998.
9. W. T. Chan and P. W. H. Wong, On-line Windows Scheduling of Temporary Items, *Proc. of the 15th ISAAC*, 259-270, 2004.
10. M. Y. Chan and F. Chin. General schedulers for the pinwheel problem based on double-integer reduction. *IEEE Trans. on Computers,* Vol. 41, 755–768, 1992.
11. M. Y. Chan and F. Chin. Schedulers for larger classes of pinwheel instances. *Algorithmica,* Vol. 9, 425–462, 1993.


[1] In a work in progress about WS with arbitrary length pages, we show that *preemptive* WS is equivalent to WS of unit-length pages. Thus, we consider here only the more restricted problem of non-preemptive scheduling.

12. E. A. Feinberg, M. Bender, M. Curry, D. Huang, T. Koutsoudis, and J. Bernstein. Sensor resource management for an airborne early warning radar. In *Proceedings of SPIE The International Society of Optical Engineering,* 145–156, 2002.
13. R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *Proc. of the 22-nd Hawaii International Conf. on System Sciences,* 693–702, 1989.
14. R. Holte, L. Rosier, I. Tulchinsky, and D. Varvel. Pinwheel scheduling with two distinct numbers. *Theoretical Computer Science,* Vol. 100, 105–135, 1992.
15. K. A. Hua and S. Sheu. An efficient periodic broadcast technique for digital video libraries. *Multimedia Tools and Applications*. Vol. 10, 157-177, 2000.
16. L. Juhn and L. Tseng. Harmonic broadcasting for video-on-demand service. *IEEE Transactions on Broadcasting*, Vol. 43, No. 3, 268-271, 1997.
17. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM,* Vol. 20, No. 1, 46-61, 1973.
18. J.F. Pâris, S. W. Carter, and D. D. E. Long. A hybrid broadcasting protocol for video on demand. *Proc. of the IS&T/SPIE Conference on Multimedia Computing and Networking*, 317-326, 1999.
19. J.F. Pâris. A broadcasting protocol for video-on-demand using optional partial preloading. *XIth International Conference on Computing*, Vol. I, 319-329, 2002.
20. R. Tijdeman. The chairman assignment problem. *Discrete Mathematics,* Vol. 32, 323-330, 1980.
21. S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *ACM Multimedia Systems Journal*, Vol. 4, No. 3, 197-208, 1996.
22. W. Wei and C. Liu. On a periodic maintenance problem. *Operations Res. Letters,* Vol. 2, 90-93, 1983.

# Implementation of Approximation Algorithms for the Multicast Congestion Problem

Qiang Lu[1,⋆] and Hu Zhang[2,⋆⋆]

[1] College of Civil Engineering and Architecture,
Zhejiang University, Hangzhou 310027, China
qlu66@zju.edu.cn
[2] Department of Computing and Software, McMaster University,
1280 Main Street West, Hamilton,
Ontario L8S 4K1, Canada
zhanghu@mcmaster.ca

**Abstract.** We implement the approximation algorithm for the multicast congestion problem in communication networks in [14] based on the fast approximation algorithm for packing problems in [13]. We use an approximate minimum Steiner tree solver as an oracle in our implementation. Furthermore, we design some heuristics for our implementation such that both the quality of solution and the running time are improved significantly, while the correctness of the solution is preserved. We also present brief analysis of these heuristics. Numerical results are reported for large scale instances. We show that our implementation results are much better than the results of a theoretically good algorithm in [10].

## 1 Introduction

We study the *multicast congestion problem* in communication networks. In a given communication network represented by an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, each vertex $v$ represents a processor, which is able to receive, duplicate and send data packets. A *multicast request* is a set $S \subseteq V$ of vertices (called *terminals*) which are to be connected such that they can receive copies of the same data packet from the source simultaneously. To fulfil a request

⋆ This work was done in part when this author was visiting the University of Kiel. Research supported in part by a scholar visiting program between the University of Kiel and Zhejiang University, by DGET, Zh0446, WAARD, G10444, grant of Zhejiang University.

⋆⋆ This work was done in part when this author was studying at the University of Kiel, Germany. Research supported in part by the DFG Graduiertenkolleg 357, Effiziente Algorithmen und Mehrskalenmethoden, by EU Thematic Network APPOL II, Approximation and Online Algorithms for Optimization Problems, IST-2001-32007, by EU Project CRESCCO, Critical Resource Sharing for Cooperation in Complex Systems, IST-2001-33135, by an MITACS grant of Canada, by the NSERC Discovery Grant DG 5-48923, and by the Canada Research Chair program.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 152–164, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$S$, one subtree $T$ in $G$ is to be generated for spanning $S$, called an $S$-tree. In the multicast congestion problem in communication networks we are given a graph $G$ and a set of multicast requests $S_1, \ldots, S_k \subseteq V$. A feasible solution is a set of $k$ trees $T_1, \ldots, T_k$, where $T_q$ connects the terminals in $S_q$, called an $S_q$-tree. The *congestion* of an edge in a solution is the number of $S_q$-trees which use the edge. The goal of the problem is to find a solution of $S_q$-trees for all $q = 1, \ldots, k$ that minimizes the maximum edge congestion.

If each request consists of only two terminals, the multicast congestion problem is reduced to the standard routing problem of finding integral paths with minimum congestion. In fact it is a generalization of the problem of finding edge disjoint shortest paths for source and destination pairs. This problem is $\mathcal{NP}$-hard [15] and hence the multicast congestion problem is also $\mathcal{NP}$-hard.

Another related problem is the *Steiner tree problem* in graphs. Given a graph $G = (V, E)$, a set $S \subseteq V$ of terminals and a non-negative length function (cost or weight) on the edges, a *Steiner tree* $T$ is a subtree spanning all vertices in $S$. The vertices of $T$ may be in $V \setminus S$. The goal of the Steiner tree problem in graphs is to find a minimum Steiner tree, i.e., a Steiner tree with minimum total edge length. Compared with the multicast congestion problem, in the Steiner tree problem there is only a single multicast and the objective function is different. However, the Steiner tree problem is proved $\mathcal{APX}$-hard [15, 1, 5]:

**Proposition 1.** *The Steiner tree problem in graphs is $\mathcal{NP}$-hard, even for unweighted graphs. Furthermore, there exists a constant $\bar{c} > 1$ such that there is no polynomial-time approximation algorithm for the Steiner tree problem in graphs with an approximation ratio less than $\bar{c}$, unless $\mathcal{P} = \mathcal{NP}$.*

The best known lower bound is $\bar{c} = 96/95 \approx 1.0105$ [8].

Since the multicast congestion problem is $\mathcal{NP}$-hard, interests turn to approximation algorithms. In [20] a routing problem in the design of a certain class of VLSI circuits was studied as a special case of the multicast congestion problem. The goal is to reduce the maximum edge congestion of a two-dimensional rectilinear lattice with a specific set of a polynomial number of trees. By solving the relaxation of the integer linear program and applying randomized rounding, a randomized algorithm was proposed such that the congestion is bounded by $\mathcal{OPT} + O(\sqrt{\mathcal{OPT} \ln(n^2/\varepsilon)})$ with probability $1 - \varepsilon$ when $\mathcal{OPT}$ is sufficiently large, where $\mathcal{OPT}$ is the optimal value. Vempala and Vöcking [22] proposed an approximation algorithm for the multicast congestion problem. They applied a separation oracle and decomposed the fractional solution for each multicast into a set of paths. An $O(\ln n)$-approximate solution can be delivered in time $O(n^6\alpha^2 + n^7\alpha)$ by their algorithm, where $\alpha$ involves the number $k$ and some other logarithmic factors. Carr and Vempala [6] proposed a randomized asymptotic algorithm for the multicast congestion problem with a constant approximation ratio. They analyzed the solution to the linear programming (LP) relaxation by the ellipsoid method, and showed that it is a convex combination of $S_i$-trees. By picking a tree with probability equal to its convex multiplier, they obtained a solution with congestions bounded by $2\exp(1)c \cdot \mathcal{OPT} + O(\ln n)$ with probability at least $1 - 1/n$, where $c > 1$ is the approximation ratio of the approximate

minimum Steiner tree solver. The algorithm needs $\tilde{O}(n^7)$ time including $k$ as a multiplication factor. Without awareness of above theoretical results, Chen et al. [7] studied this problem from practical point of view, which was called *multicast packing problem* in their paper. They showed some lower bounds for the problem and implemented some instances with small sizes by the branch-and-cut algorithm. More works on the multicast packing problem can be found in [18].

Baltz and Srivastav [3] studied the multicast congestion problem and proposed a formulation based on the ideas of Klein et al. [16] for the concurrent multicommodity flow problem with uniform capacities. The integer linear program has an exponential number of variables and they constructed a combinatorial LP-algorithm to obtain a polynomial number of $S_q$-trees for each multicast request $S_q$. Finally a randomized rounding technique in [19] was applied. The solution of their algorithm is bounded by

$$\begin{cases} (1+\varepsilon)c\cdot\mathcal{OPT}+(1+\varepsilon)(\exp(1)-1)\sqrt{c\cdot\mathcal{OPT}\ln m}, \text{ if } c\cdot\mathcal{OPT}\geq\ln m, \\ (1+\varepsilon)c\cdot\mathcal{OPT}+\dfrac{(1+\varepsilon)\exp(1)\ln m}{1+\ln(\ln m/(c\cdot\mathcal{OPT}))}, \qquad \text{otherwise.} \end{cases} \tag{1}$$

In the case $c\cdot\mathcal{OPT}\geq\ln m$ the bound is in fact $(1+\varepsilon)\exp(1)c\cdot\mathcal{OPT}$ and otherwise it is $(1+\varepsilon)c\cdot\mathcal{OPT}+O(\ln m)$. The running time is $O(\beta nk^3\varepsilon^{-9}\ln^3(m/\varepsilon)\cdot\min\{\ln m,\ln k\})$, where $\beta$ is the running time of the approximate minimum Steiner tree solver. A randomized asymptotic approximation algorithm for the multicast congestion problem was presented in [14]. They applied the fast approximation algorithm for packing problems in [13] to solve the LP relaxation of the integer linear program in [3]. They showed that the block problem is the Steiner tree problem. The solution hence is bounded by (1) and the running time is improved to $O(m(\ln m+\varepsilon^{-2}\ln\varepsilon^{-1})(k\beta+m\ln\ln(m\varepsilon^{-1})))$. Baltz and Srivastav [4] further proposed an approximation algorithm for the multicast congestion problem based on the algorithm for packing problems in [10], which has the best known complexity $O(k(m+\beta)\varepsilon^{-2}\ln k\ln m)$. They also conducted some implementation with typical instances to explore the behaviour of the algorithms. It was reported that the algorithm in [10] is very impractical. In addition, they presented a heuristic based on an online algorithm in [2], which can find good solutions for their test instances within a few iterations.

In this paper we implement the algorithm in [14] with large scale instances. We design some heuristics to speed up the computation and to improve the quality of solution delivered in our implementation. We also present brief analysis of the heuristics. The numerical results show that the algorithm for packing problems [13] is reliable and practical. We also compare our results with those by the algorithm in [10] and the heuristic in [4]. Because other algorithms mentioned above are very impractical, we do not consider them for implementation.

The paper is organized as follows: In Section 2 the approximation algorithm for the multicast congestion problem in [14] is briefly reviewed. We analyze the technique to overcome the hardness of exponential number of variables in Section 3. Our heuristics are presented in Section 4. Finally, numerical results are reported in Section 5 with comparison with other approaches.

## 2 Approximation Algorithm

Let $\mathcal{T}_q$ be the set of all $S_q$-trees for any $q \in \{1, \ldots, k\}$. Here the cardinality of $\mathcal{T}_q$ may be exponentially large. Define by $x_q(T)$ a variable indicating whether the tree $T \in \mathcal{T}_q$ is chosen in a solution for the multicast request $S_q$. Based on the idea in [3, 4], the following integer linear program can be formulated:

$$\begin{array}{ll} \min \lambda & \\ \text{s.t.} \sum_{q=1}^{k} \sum_{T \in \mathcal{T}_q \ \& \ e_i \in T} x_q(T) \leq \lambda, & \text{for all } i \in \{1, \ldots, m\}; \\ \sum_{T \in \mathcal{T}_q} x_q(T) = 1, & \text{for all } q \in \{1, \ldots, k\}; \\ x_q(T) \in \{0, 1\}, & \text{for all } q \text{ and all } T \in \mathcal{T}_q, \end{array} \tag{2}$$

where $\lambda$ is the maximum congestion. The first set of constraints show that the congestion on any edge is bounded by $\lambda$, and the second set of constraints indicate that exact one Steiner tree is chosen for one request. As usual, the strategy is to first solve the LP relaxation of (2) and then round the fractional solution to a feasible solution.

We define a vector $x_q = (x_q(T_1), x_q(T_2), \ldots)^T$ for all $T_1, T_2, \ldots \in \mathcal{T}_q$ representing the vector of indicator variables corresponding to all Steiner trees for the $q$-th request. Denote by a vector $x = (x_1^T, \ldots, x_k^T)^T$ the vector of all indicator variables. Furthermore, a vector function $f(x) = (f_1(x), \ldots, f_m(x))^T$ is used, where $f_i(x) = \sum_{q=1}^{k} \sum_{T \in \mathcal{T}_q \ \& \ e_i \in T} x_q(T)$ represents the congestion on edge $e_i$, for $i \in \{1, \ldots, m\}$. In addition, we define by $B = B_1 \times \ldots \times B_k$ where $B_q = \{(x_q(T))^T | T \in \mathcal{T}_q, \sum_{T \in \mathcal{T}_q} x_q(T) = 1, x_q(T) \geq 0\}$, for $q \in \{1, \ldots, k\}$. It is obvious that $x_q \in B_q$ and $x \in B$. In this way the LP relaxation of (2) is formulated as the following *packing problem* (the linear case of the *min-max resource sharing problems* [12, 24, 13]): $\min\{\lambda | f(x) \leq \lambda, x \in B\}$. Thus we are able to use the approximation algorithm for packing problems [13] to solve the LP relaxation of (2).

The computational bottleneck lies on the exponential number of variables $x_q(T)$ in (2). The algorithm for packing problems in [13] is employed in [14] with a column generation technique implicitly applied. We briefly describe the algorithm as follows. The algorithm is an iterative method. In each iteration (coordination step) there are three steps. In the first step a *price vector* $w$ is calculated according to current iterate $x$. Then an approximate block solver is called as an oracle to generate an approximate solution $\hat{x}$ corresponding to the price vector $w$ in the second step. In the third step the iterate is moved to $(1 - \tau)x + \tau\hat{x}$ with an appropriate step length $\tau \in (0, 1)$. The coordination step stops when any one of two stopping rules holds with respect to an relative error tolerance $\sigma$, which indicates that the resulting iterate is a $c(1 + \sigma)$-approximate solution. Scaling phase strategy is applied to reduce the coordination complexity. In the first phase $\sigma = 1$ is set. When a coordination step stops, current phase finishes and $\sigma$ is halved to start a new phase, until $\sigma \leq \varepsilon$. Finally the delivered solution fulfils $\lambda(x) \leq c(1 + \varepsilon)\lambda^*$, where $\lambda^*$ is the optimum value of the LP relaxation of (2) (See [13, 14]).

The block problem is exactly the Steiner tree problem in graphs and the edge length function is the price vector $w$ [14]. So $k$ minimum Steiner trees are computed corresponding to the $k$ requests $S_1, \ldots, S_k$ with respect to the length function in current iteration. In the iterative procedure lengths on the edges with large congestions increase while edges with small congestions have decreasing lengths. In this way the edges with large congestions are punished and have low probability to be selected in the generated Steiner trees. The best known algorithm for the Steiner tree problem has an approximation ratio $c = 1 + (\ln 3)/2 \approx 1.550$ [21] but the complexity is large. So in our implementation, we use a 2-approximate minimum Steiner tree solver ($\mathcal{MSTS}$) as the block solver, and its time complexity is $O(m + n \ln n)$ [17, 9]. We call this algorithm $\mathcal{MC}$ and its details can be found in [13, 14]. Then the following result holds [13, 14]:

**Theorem 1.** *For a given relative accuracy $\varepsilon \in (0, 1)$, Algorithm $\mathcal{MC}$ delivers a solution $x$ such that $\lambda(x) \leq c(1 + \varepsilon)\lambda^*$ in $N = O(m(\ln m + \varepsilon^{-2} \ln \varepsilon^{-1}))$ iterations. The overall complexity of Algorithm $\mathcal{MC}$ is $O(m(\ln m + \varepsilon^{-2} \ln \varepsilon^{-1})(k\beta + m \ln \ln(m\varepsilon^{-1})))$, where $\beta$ is the complexity of the approximate minimum Steiner tree solver.*

## 3 The Number of Variables

In the LP relaxation of (2), there can be an exponential number of variables. However, with the algorithm in [13, 14], a column generation technique is automatically applied and totally the trees generated by the algorithm is a polynomial size subset of $\mathcal{T} = \cup_{q=1}^{k} \mathcal{T}_q$.

If a Steiner tree $T_{q_j} \in \mathcal{T}_q$ is chosen for a request $S_q$, the corresponding indicator variable is set to $x_{q_j} = 1$. In the fractional sense, it represents the probability to choose the corresponding Steiner tree $T_{q_j}$. For any tree $T_{q_j} \in \mathcal{T}_q$ for a request $S_q$, if it is not generated by $\mathcal{MSTS}$ in any iteration of Algorithm $\mathcal{MC}$, then the corresponding indicator variable $x_{q_j} = 0$, which shows that it will never be chosen. Because in each iteration, there are $k$ Steiner trees generated for the $k$ requests, respectively, we conclude that there are only polynomially many trees generated in Algorithm $\mathcal{MC}$ according to Theorem 1:

**Theorem 2.** *When Algorithm $\mathcal{MC}$ halts, there are only $O(km(\ln m + \varepsilon^{-2} \ln \varepsilon^{-1}))$ non-zero indicator variables of the vector $x$ and only the same number of Steiner trees generated.*

In our implementation, we maintain a vector $x$ with a size $k(N + 1)$, where $N$ is the actual number of iterations. We also maintain a set $\mathcal{T}$ of Steiner trees generated in the algorithm. Notice that here $\mathcal{T}$ is not the set of all feasible Steiner trees. At the beginning the set $\mathcal{T}$ is empty and all components of $x$ are zeros. In the initialization step, $k$ Steiner trees are generated. Then the first $k$ components of $x$ are all ones and the corresponding generated $k$ Steiner trees $T_1, \ldots, T_k$ are included in $\mathcal{T}$. In the $j$-th iteration, for the $q$-th request a Steiner tree $T_{jk+q}$ is

generated. No matter whether it is identical to any previously generated tree, we just consider it as a new one and include it in the tree set $\mathcal{T}$. Meanwhile, we set the corresponding components $\hat{x}_{jk+q} = 1$. Therefore after the $j$-th iteration there are totally $(j+1)k$ nonzero indicator variables (nonzero probability to select the corresponding trees in $\mathcal{T}$). Finally, there are $|\mathcal{T}| = (N+1)k$ non-zero indicator variables.

However, in practice it is not easy to estimate the exact value of $N$ in advance as there is only an upper bound $O(m(\ln m + \varepsilon^{-2} \ln \varepsilon^{-1}))$ for $N$. In our implementation, we set $N = 100$. If it is insufficient we will double it, until the value of $N$ suffices. In fact according to our implementation results the setting $N = 100$ is enough as for all of our test instances there are only $O(k)$ Steiner trees generated (See Section 5).

## 4 Heuristics

### 4.1 Choose the Step Length

In Algorithm $\mathcal{MC}$ the step length $\tau$ is set as $t\theta\nu/(2m(w^T f(x) + w^T f(\hat{x})))$ as in [13, 14], where $t$ and $\theta$ are parameters for computing the price vector, and $\nu = (w^T f(x) - w^T f(\hat{x}))/(w^T f(x) + w^T f(\hat{x}))$ is a parameter for stopping rules. In the last coordination steps of $\mathcal{MC}$, we have that $t = O(\varepsilon)$ and $\nu = O(\varepsilon)$ according to the scaling phase and the stopping rules, respectively. Assuming that $\theta/(w^T f + w^T \hat{f}) = O(1)$, we notice that $\tau = O(\varepsilon^2/m)$ is very small. It means that the contribution of the block solution is very tiny and the iterate moves to the desired neighbourhood of the optimum very slowly, which results in a large number of iterations (though the bound in Theorem 1 still holds). In fact in our implementation we find that even at the beginning of the iterative procedure the value of $\tau$ defined in [13, 14] is too small. In [11, 13] it is mentioned that any $\tau \in (0, 1)$ can be employed as the step length. We test several feasible settings of $\tau$ such as $\tau = 1 - t\theta\nu/(2m(w^T f + w^T \hat{f}))$, $\tau = 1 - \nu$ and $\tau = \nu$. Experimental results show that $\tau = \nu$ is the best among them. With this heuristic, the number of iterations is reduced significantly (see Section 5).

### 4.2 Remove the Scaling Phase

In our implementation we set $\varepsilon = 10^{-5}$. In this way we are able to estimate the number of scaling phases $N_s = -\log \varepsilon = 5 \log 10 \approx 16.61$. Therefore in the total computation there should be 17 scaling phases. In fact we find that in many cases in our implementation there is only one iteration in each scaling phase. Thus, the number of scaling phases dominates the overall number of iterations and there are only $O(1)$ iterations in a scaling phase.

We notice that in [13] the algorithm without scaling phase is also mentioned and the corresponding coordination complexity is $O(mc^2(\ln m + \varepsilon^{-2} + \varepsilon^{-3} \ln c))$. In our implementation $c = 2$ is a constant so the complexity does not increase much. In practice with this strategy the algorithm could run faster, especially

when there are only very few iterations in each scaling phase. Therefore we use this approach and the number of iterations is reduced.

### 4.3 Add Only One Steiner Tree in Each Iteration

Algorithm $\mathcal{MC}$ calls the block solver $\mathcal{MSTS}$ $k$ times independently for the $k$ requests in each iteration. We now consider Example 1 which leads to hardness for finding an optimum solution. The instance is as follows: In the graph $G$, $|V| = 4$ and $|E| = 5$. The edges are $(1,2)$, $(1,3)$, $(2,3)$, $(1,4)$ and $(2,4)$ (see Figure 1(a)). There are 3 identical requests $S_q = \{1,2\}$ for $q = 1,2,3$. In general we can also study the graphs with $|V| = p$, $|E| = 2p-3$, with edge set $E = \{(1,2),(1,i),(i,2)|i = 3,\ldots,p\}$ and identical requests $S_q = \{1,2\}$ for $q = 1,\ldots,p-1$ for $p \in N$ and $p \geq 4$.

In the initialization step of Algorithm $\mathcal{MC}$, each edge is assigned an identical length $1/5$. For all requests, the minimum Steiner trees $T_q$, $q = 1,2,3$ are all the path containing only edge $(1,2)$, with a total length $1/5$. After $T_1$ is generated for the first request $S_1$, Algorithm $\mathcal{MC}$ is not aware of the change of the congestion on edge $(1,2)$, and still assign the identical trees $T_2$ and $T_3$ to requests $S_2$ and $S_3$. After the initialization congestions of edges are all zero except for edge $(1,2)$, which has a congestion 3 (see Figure 1(b)). In the first iteration, the edge lengths changes and the length on edge $(1,2)$ is the maximum, and other edges have very small lengths. Therefore Algorithm $\mathcal{MC}$ will choose the path $\{(1,3),(3,2)\}$ as $T_4$ for $S_1$. With the same arguments, other requests are also assigned the path $\{(1,3),(3,2)\}$ as their corresponding minimum Steiner trees (see Figure 1(c)). In the second iteration all requests are assigned the path $\{(1,4),(4,1)\}$ (see Figure 1(d)) and in the third iteration the solution returns back to the case in Figure 1(b). This procedure continues and in each iteration only one path is used for all requests, which leads to a wrong solution with always a maximum congestion 3. It is also verified by our implementation.

This problem does not result from Algorithm $\mathcal{MC}$ itself but from the data structure (the indices of the vertices and edges). An intuitive approach is to re-index the nodes (and hence edges) after each Steiner tree is generated. However, this approach causes large computational cost of re-indexing. The strategy we apply here is to establish a permutation of the requests. In each iteration only one request is chosen according to the permutation, and a Steiner tree is

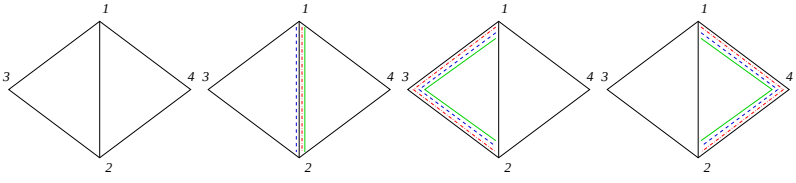



**Fig. 1.** Examples 1

generated by $\mathcal{MSTS}$ for the chosen request. This method is applied in [23] to solve the packing problems with block structured variables but with a standard (not weak) approximate block solver. The bound on the number of iterations is $O(k \ln m(\ln \min\{k, m\} + \varepsilon^{-2}))$, where $k$ is the number of blocks of variables. There is also a randomized algorithm for such problems [11] with a number of iterations bounded by $O(k \ln m(\ln k + \varepsilon^{-2}))$. In our problem there are also $k$ blocks of indicator variables corresponding to the $k$ requests. But there are only weak block solvers with the approximation ratio $c > 1$. So we apply this method as a heuristic in our implementation. Furthermore, in our implementation we find it is not necessary to construct and maintain the permutation. We can just choose the requests according to their indices. In this way the optimum solution can be attained in only 3 iterations for Example 1 such that the three requests are realized by the three disjoint paths between vertex 1 and 2.

It is interesting that when this heuristic is employed, not only the quality of the solution but also the running time are improved. In fact for many instances with symmetric topology structure, such a problem due to data structure can happen without our heuristic or the re-indexing approach.

### 4.4 Punish the Edges with Large Congestions

We use a 2-approximate minimum Steiner tree solver ($c = 2$) here as the block solver in our implementation. We notice that with the above heuristics we can only obtain a solution bounded by (1) as indicated in our implementation results (see Section 5). In fact our implementation shows that as soon as the solution fulfils (1) for $c = 2$, the algorithm halts immediately. In order to obtain a better approximate solution still with $\mathcal{MSTS}$, we could modify the stopping rules to force the algorithm to continue running with more iterations. However, here we use another heuristic without changing the stopping rules in order to avoid more running time.

The price vector is used as edge length in our algorithm for the Steiner tree problem. It is obvious that a large congestion leads to a large length on the edge. Thus we can add extra punishment to edges with large congestions to balance the edge congestions over the whole graph. We apply the following strategy:

First we define an edge $e_i$ *high-congested* if its congestion $f_i$ fulfils the following inequality:

$$\lambda - f_i \leq r(\lambda - \hat{\lambda}). \tag{3}$$

Here $\lambda$ is the maximum congestion in current iteration, $\hat{\lambda}$ is the average congestion defined as the sum of congestions over all edges divided by the number of edges with nonzero congestions, and $r$ is a ratio depending on the quality of the current solution defined as follows:

$$r = \sqrt{1 - (\lambda_0 - \lambda)^2/\lambda_0^2}, \tag{4}$$

where $\lambda_0 \geq \lambda$ is the maximum congestion of the initial solution. According to (4), $r \in (0, 1]$. In addition, (4) is an ellipse function. At the beginning $\lambda \approx \lambda_0$ so $r \approx 1$. With the maximum congestion being reduced, the value of $r$ also

decreases. Furthermore, according to the property of the ellipse function, at the beginning of the iterative procedure the value of $r$ decreases slowly. When the congestions are well distributed, $r$ reduces quickly. This formulation guarantees that at the beginning of the iterative procedure there is a large portion of high-congested edges while later there is only a small portion.

Next we re-assign length function to all edges in the graph. For any edge not high-congested, we keep its length as computed by the method in [13, 14]. For a high-congested edge, we set its length as its current congestion. Afterwards we normalize all edge lengths such that the sum of lengths of all edges is exactly one. Our implementation shows that this technique can not only improve the quality of solution but also speed up the convergence (with less iterations).

## 5 Experimental Results

Our test instances are two-dimensional rectilinear lattices (grid graphs with certain rectangular holes). These instances typically arise in VLSI logic chip design problems and the holes represent big arrays on the chips. These instances are regarded hard for path- or tree-packing problems. The instances have the following sizes:

*Example 1.* $n = 2079$ and $m = 4059$; $k = 50$ to 2000.
*Example 2.* $n = 500$ and $m = 940$; $k = 50$ to 300.
*Example 3.* $n = 4604$ and $m = 9058$; $k = 50$ to 500.
*Example 4.* $n = 1277$ and $m = 2464$; $k = 50$ to 500.

We first demonstrate the influence of the heuristics mentioned in Section 4 by a hard instance. The instance belongs to Instance 3 with 4604 vertices, 9058 edges and 100 requests. The sizes of requests varies and the smallest request has 5 vertices. We test our algorithm without or with heuristics and the results are shown in Table 1.

We refer Algorithm 1 the original Algorithm $\mathcal{MC}$ without any heuristics. Algorithm 2 is referred to Algorithm $\mathcal{MC}$ with the heuristic to add only one Steiner tree in each iteration. For Algorithm 3, we refer the algorithm similar to Algorithm 2 but with step length $\tau = \nu$. Algorithm 4 is similar to Algorithm 3 but with extra punishment to high-congested edges. It is worth noting that in Algorithm 1, the block solver $\mathcal{MST}$ is called $k$ times in each iteration, while

**Table 1.** Numerical results of Algorithm $\mathcal{MC}$ without and with heuristics

| | Alg. 1 | Alg. 2 | Alg. 3 | Alg. 4 |
|---|---|---|---|---|
| Initial Congestion | 17 | 17 | 17 | 17 |
| Final Congestion | 17 | 13 | 6 | 4 |
| Number of Calls | – | 44 | 85 | 90 |

**Table 2.** Numerical results of Instance 1 compared with Garg-Könemann's algorithm and Baltz-Srivastav's heuristic

| # req.(# term.) | G-K | B-S | Alg. 3 | Alg. 4 |
|---|---|---|---|---|
| 50(4) | 2.5(5000) | 2(50) | 4(111) | 2(67) |
| 100(4) | 4.4(10000) | 3(100) | 7(207) | 3(180) |
| 150(4) | 6.1(15000) | 4(300) | 9(314) | 5(131) |
| 200(4) | 8.0(20000) | 5(400) | 11(594) | 6(260) |
| 300(4) | 11.5(30000) | 7(900) | 15(826) | 8(492) |
| 500(4) | 19.9(50000) | 12(1000) | 23(1389) | 13(977) |
| 1000(4) | 36.5(100000) | 21(69000) | 48(2786) | 24(2955) |
| 2000(4) | 76.1(200000) | 44(4000) | 96(5563) | 54(3878) |
| 500($\geq 2$) | 69.1(50000) | 32(4500) | 39(1381) | 37(501) |
| 1000($\geq 2$) | 100.5(100000) | 65(3000) | 78(2933) | 72(1004) |

in Algorithm 2, 3 and 4 $\mathcal{MSTS}$ is called only once in each iteration. In order to compare the running time fairly, we count the number of calls to $\mathcal{MSTS}$ as the measurement of running time. In fact according to our implementation, the running time of $\mathcal{MSTS}$ dominates the overall running time. From Table 1 it is obvious that the heuristics improve the quality of solution much. Since the value of $\tau$ is too small in Algorithm 1, the iterate does not move after long time and we manually terminate the program.

In [4] Instance 1 was implemented to test their heuristic based on an online algorithm in [2] and a well-known approximation algorithm for packing problems in [10] based on an approximation algorithm for the fractional multicommodity flow problem. Here, we also use the same instances to test our Algorithm 3 and 4. The results are shown in Table 2. In the first column of Table 2 the number of requests and the number of terminals per request are given. The solution delivered by the algorithms and heuristics are presented in other columns, together with the number of calls to $\mathcal{MSTS}$ in brackets. The results of Garg and Könemann's algorithm are only for the LP relaxation.

It is clear that Algorithm 4 is superior to Algorithm 3 in the examples of regular requests (with 4 terminals per request). Furthermore, it is worth noting that our Algorithm 4 delivers better solutions than the algorithm by Garg and Könemann [10] with much less number of calls to $\mathcal{MSTS}$. In fact the fractional solutions of Algorithm 3 are also better than those of the algorithm by Garg and Könemann. Our results are not as good as those of the heuristic proposed in [4] for these instances. However, there is no performance guarantee of their heuristic, while our solutions are always bounded by (1). A possible reason of this case is that we use a 2-approximate block solver, which leads to a low accuracy. We believe that a better approximate minimum Steiner tree solver and some more strict stopping rules can result in better performance of our algorithm.

We also test our Algorithm 4 by Instances 2, 3 and 4, which are not implemented in [4]. The results are listed in Table 3. Our algorithm can always generate satisfactory solutions for these hard instances in short running times.

**Table 3.** Numerical results of Instance 2, 3 and 4

| Inst. | # req.(# term.) | Alg. 4 | Inst. | # req.(# term.) | Alg. 4 |
|---|---|---|---|---|---|
| 2 | 50(≥ 10) | 7(37) | 2 | 150(≥ 30) | 25(116) |
| 2 | 50(≥ 5) | 6(41) | 2 | 200(≥ 10) | 34(404) |
| 2 | 100(≥ 5) | 12(83) | 2 | 200(≥ 30) | 32(148) |
| 2 | 100(≥ 10) | 14(77) | 2 | 300(≥ 10) | 38(559) |
| 2 | 150(≥ 10) | 19(131) | 2 | 300(≥ 30) | 47(231) |
| 3 | 50(≥ 5) | 2(146) | 3 | 200(≥ 20) | 13(320) |
| 3 | 50(≥ 20) | 4(105) | 3 | 300(≥ 5) | 8(565) |
| 3 | 100(≥ 5) | 4(90) | 3 | 300(≥ 20) | 19(285) |
| 3 | 100(≥ 20) | 7(186) | 3 | 500(≥ 5) | 13(962) |
| 3 | 200(≥ 5) | 6(210) | 3 | 500(≥ 20) | 30(483) |
| 4 | 50(≥ 5) | 3(103) | 4 | 200(≥ 20) | 24(165) |
| 4 | 50(≥ 20) | 7(36) | 4 | 300(≥ 5) | 14(560) |
| 4 | 100(≥ 5) | 6(83) | 4 | 300(≥ 20) | 34(280) |
| 4 | 100(≥ 20) | 13(86) | 4 | 500(≥ 5) | 24(475) |
| 4 | 200(≥ 5) | 10(173) | 4 | 500(≥ 20) | 56(390) |

For any request of all these instances, the corresponding $\mathcal{MSTS}$ is called at most 3 times.

## 6 Conclusion

We have implemented the approximation algorithm for the multicast congestion problem in communication networks in [14] based on [13] with some heuristics to improve the quality of solution and reduce the running time. The numerical results for hard instances are reported and are compared with the results of the approximation algorithm in [10] and a heuristic in [4]. It shows that the algorithm in [13] is practical and efficient for packing problems with a provably good approximation ratio.

There could be some interesting techniques to further improve the experimental performance of the algorithm. A possible method is to use a better approximate minimum Steiner tree solver (e.g. the algorithm in [21]), though the running time will be significantly increased. Another technique is to use the line search for the step length to reduce the number of iterations. However, the running time in each iteration increases so the improvement of the overall running time could be not significant. More heuristics and techniques are to be designed and implemented in our further work.

## Acknowledgment

The authors thank Andreas Baltz for providing the data of testing instances. We also thank Klaus Jansen for his helpful discussion.

## References


1. S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy, Proof verification and hardness of approximation problems, *Journal of the ACM*, 45 (1998), 501-555.
2. J. Aspnes, Y. Azar, A. Fiat, S. Plotkin and O. Waarts, On-line routing of virtual circuits with applications to load balancing and machine scheduling, *Journal of the Association for Computing Machinery*, 44(3) (1997), 486-504.
3. A. Baltz and A. Srivastav, Fast approximation of multicast congestion, *manuscript* (2001).
4. A. Baltz and A. Srivastav, Fast approximation of minimum multicast congestion - implementation versus theory, *Proceedings of the 5th Conference on Algorithms and Complexity*, CIAC 2003.
5. M. Bern and P. Plassmann, The Steiner problem with edge lengths 1 and 2, *Information Professing Letters*, 32 (1989), 171-176.
6. R. Carr and S. Vempala, Randomized meta-rounding, *Proceedings of the 32nd ACM Symposium on the Theory of Computing*, STOC 2000, 58-62.
7. S. Chen, O. Günlük and B. Yener, The multicast packing problem, *IEEE/ACM Transactions on Networking*, 8 (3) (2000), 311-318.
8. M. Chlebík and J. Chlebíková, Approximation hardness of the Steiner tree problem, *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, SWAT 2002, LNCS 2368, 170-179.
9. R. Floren, A note on "A faster approximation algorithm for the Steiner problem in graphs", *Information Processing Letters*, 38 (1991), 177-178.
10. N. Garg and J. Könemann, Fast and simpler algorithms for multicommodity flow and other fractional packing problems, *Proceedings of the 39th IEEE Annual Symposium on Foundations of Computer Science*, FOCS 1998, 300-309.
11. M. D. Grigoriadis and L. G. Khachiyan, Fast approximation schemes for convex programs with many blocks and coupling constraints, *SIAM Journal on Optimization*, 4 (1994), 86-107.
12. M. D. Grigoriadis and L. G. Khachiyan, Coordination complexity of parallel price-directive decomposition, *Mathematics of Operations Research*, 2 (1996), 321-340.
13. K. Jansen and H. Zhang, Approximation algorithms for general packing problems with modified logarithmic potential function, *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science*, TCS 2002, 255-266.
14. K. Jansen and H. Zhang, An approximation algorithm for the multicast congestion problem via minimum Steiner trees, *Proceedings of the 3rd International Workshop on Approximation and Randomized Algorithms in Communication Networks*, ARACNE 2002, 77-90.
15. R. M. Karp, Reducibility among combinatorial problems, *in R. E. Miller and J. W. Thatcher (Eds.), Complexity of Computer Computations*, Plenum Press, NY, (1972), 85-103.
16. P. Klein, S. Plotkin, C. Stein and E. Tardos, Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts, *SIAM Journal on Computing*, 23 (1994), 466-487.
17. K. Mehlhorn, A faster approximation algorithm for the Steiner problem in graphs, *Information Processing Letters*, 27 (1988), 125-128.
18. C. A. S. Oliveira and P. M. Pardolos, A survey of combinatorial optimization problems in multicast routing, *Computers and Operations Research*, 32 (2005), 1953-1981.

19. P. Raghavan, Probabilistic construction of deterministic algorithms: Approximating packing integer programs, *Journal of Computer and System Science*, 37 (1988), 130-143.
20. P. Raghavan and C. Thompson, Randomized rounding: a technique for provably good algorithms and algorithmic proofs, *Combinatorica*, 7 (1987), 365-374.
21. G. Robins and A. Zelikovsky, Improved Steiner tree approximation in graphs, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2000, 770-779.
22. S. Vempala and B. Vöcking, Approximating multicast congestion, *Proceedings of the tenth International Symposium on Algorithms and Computation*, ISAAC 1999, LNCS 1741, 367-372.
23. J. Villavicencio and M. D. Grigoriadis, Approximate structured optimization by cyclic block-coordinate descent, *Applied Mathematics and Parallel Computing, H. Fisher et al. (Eds), Physica Verlag*, (1996), 359-371.
24. J. Villavicencio and M. D. Grigoriadis, Approximate Lagrangian decomposition with a modified Karmarkar logarithmic potential, *Network Optimization, P. Pardalos, D. W. Hearn and W. W. Hager (Eds.), Lecture Notes in Economics and Mathematical Systems 450, Springer-Verlag, Berlin*, (1997), 471-485.

# Frequency Assignment and Multicoloring Powers of Square and Triangular Meshes

Mustapha Kchikech and Olivier Togni

LE2I, UMR CNRS,
Université de Bourgogne,
21078 Dijon Cedex, FRANCE
{kchikech, olivier.togni}@u-bourgogne.fr

**Abstract.** The static frequency assignment problem on cellular networks can be abstracted as a multicoloring problem on a weighted graph, where each vertex of the graph is a base station in the network, and the weight associated with each vertex represents the number of calls to be served at the vertex. The edges of the graph model interference constraints for frequencies assigned to neighboring stations. In this paper, we first propose an algorithm to multicolor any weighted planar graph with at most $\frac{11}{6}W$ colors, where $W$ denotes the weighted clique number. Next, we present a polynomial time approximation algorithm which garantees at most $2W$ colors for multicoloring a power square mesh. Further, we prove that the power triangular mesh is a subgraph of the power square mesh. This means that it is possible to multicolor the power triangular mesh with at most $2W$ colors, improving on the known upper bound of $4W$. Finally, we show that any power toroidal mesh can be multicolored with strictly less than $4W$ colors using a distributed algorithm.

**Keywords:** Graph multioloring; power graph; approximation algorithm; distributed algorithm; frequency assignment, cellular networks.

## 1 Introduction

A cellular network covers a certain geographic area, which is divided into regions called *cells*. Each cell contains a base station equipped with radio transceivers. Users in a cell are served by a base station and cells can communicate with their neighbors via radio transceivers. This communication consists in assigning a frequency to each call in a manner that avoids interference between two distincts calls. However, cellular networks use a fixed spectrum of radio frequencies and the efficient shared utilisation of the limited available bandwidth is critical to the viability and efficiency of the network. The *static frequency assignment problem with reuse distance d* (also called interference constraints), therefore, consists in designing an interference frequency allocation for a given network where the number of calls per cell is known, and it is assumed that a frequency can be *reused* without causing interference in two cells if the distance between them is at least $d$. This forms the motivation for the problems studied in this paper.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 165–176, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Cellular networks are often modeled as finite portions of the infinite triangular mesh embedded in the plane. Vertices represent cells and edges correspond to interference constraints for frequencies assigned to neighboring stations.

The frequency assignment problem described above is a *multicoloring* problem in the triangular mesh, and it can be abstracted as follows. Let $G = (V, E)$ be a finite undirected subgraph of the triangular mesh. Each vertex $v \in V$ has an associated nonnegative integer *weight*, noted $\omega(v)$. A *multicoloring* of $G$ is an assignment of sets of colors to the vertices such that each vertex $v$ is assigned a set of $\omega(v)$ distinct colors, any pair of adjacent vertices $u, v$ in $G$ are assigned *disjoint sets* of colors. Note that the weight of a vertex is the number of calls in the corresponding cell and the assigned colors are the allocated frequencies. The frequency assignment problem with reuse distance $d = p + 1$ is thus equivalent to the problem of multicoloring the $p^{th}$ *power of* $G$.

There is a vast literature on algorithms for the multicoloring problem (also known as weighted coloring [7] or $\omega$-coloring [9]) on graphs (especially triangular mesh) [3, 4, 5, 6, 7, 9]. McDiarmid and Reed proved [7] that this problem is NP-hard. Hence, it would be interesting to find algorithms that approximate the number of colors used. But generally there are no proven bounds on the approximation ratio of the proposed algorithms in terms of the number of colors used in relationship to the weighted clique number. In this work, we give special attention to some powers of the graphs with an embedding into the plane, in particular square mesh and triangular mesh. Also, we study the multicoloring problem on the power toroidal mesh.

In the next section, we present some definitions of basic terminology. In Section 3, we describe an algorithm for multicoloring any planar graph in which the number of colors used is within a factor $\frac{11}{6}$ to the weighted clique number. Our main results presented in Section 4 and in Section 5 concern the multicoloring the power square mesh and the power triangular mesh with an algorithm using a number of colors at most 2 times the weighted clique number. Finally, for multicoloring a power toroidal mesh, we propose in Section 6 a distributed algorithm with guaranteed approximation ratio of 4.

## 2 Preliminaries

In this paper, we denote by $G = (V, E)$ a finite and simple graph with vertex set $V$ and edge set $E$. The length of a path between two vertices is the number of edges on that path. The distance in $G$ between two vertices $u, v \in V$, noted $d_G(u, v)$, is the length of a shortest path between them. Given a positive integer $p$, the $p^{th}$ power $G^p$ of a graph $G$ is a graph with the same set of vertices as $G$ and an edge between two vertices if and only if there is a path of length at most $p$ between them in $G$. A proper coloring of $G$ is an assignment of colors to its vertices such that no two adjacent vertices receive the same colors. The minimum number of colors for which a coloring of $G$ exists is called the chromatic number and is denoted by $\chi(G)$.

Given a graph $G$, a *weighted graph* $G_\omega$ associated with $G$ is a pair $G_\omega = (G, \omega)$ where $\omega$ is a weight function that assigns a non-negative integer to each vertex $v$ of $G$, $\omega(v)$ is called the *weight* of $v$. A multicoloring of the weighted graph $G_\omega$ consists of a set of colors $\mathcal{C}$ and a function $f$ that assigns to each $v \in V$ a subset of colors $f(v) \subset \mathcal{C}$ such that:

**i)** $\forall\ v \in V$, $|f(v)| = \omega(v)$, i.e. the vertex $v$ gets $\omega(v)$ distinct colors.
**ii)** If $(u, v) \in E$ then $f(u) \cap f(v) = \emptyset$, i.e. two adjacent vertices get disjoint sets of colors.

The *weighted chromatic number*, denoted $\chi_\omega(G)$, of $G_\omega$ is the minimum number of colors needed to multicolor all vertices of $G_\omega$ so that conditions i) and ii) above are satisfied. An algorithm is an $\alpha$-approximation algorithm for the multicoloring problem if the algorithm runs in polynomial time and it always produces a solution that is within a factor of $\alpha$ of the optimal solution.

A subgraph $K$ of $G_\omega$ is called a *clique* if every pair of vertices in $K$ is connected by an edge. The weight of any clique in $G_\omega$ is defined as the sum of the weights of the vertices forming that clique. The *weighted clique number* of $G_\omega$, denoted $W_G$ (for short, we will sometimes use $W$), is defined to be the maximum over the weights of all cliques in $G_\omega$. Clearly, $\chi_\omega(G) \geq W$.

## 3 Planar Graphs

The Frequency assignment problem on planar graphs have been studied in [1, 8]. In this section we consider the multicoloring problem on planar graphs. Before, we present a result of Narayanan and Shende [9] showing that there exists an efficient algorithm to optimally multicolor any outerplanar graph. A graph is *planar* if it can be drawn in a plane without edge crossings. A graph is said to be *outerplanar* if it is a planar graph so that all vertices may lie on the outer face.

**Theorem 1.** ([9]) *Let $G$ be an arbitrary outerplanar graph, then its associated weighted graph $G_\omega$ can be multicolored optimally using $\chi_\omega(G)$ colors in linear time.*

Now, we consider the problem of computing an approximate multicoloring of an arbitrary planar graph.

**Theorem 2.** *Let $G$ be a planar graph, then*

$$\chi_\omega(G) \leq \frac{11}{6} \cdot W_G$$

*Proof.* Suppose that $G$ is connected, since disconnected components of $G$ can be multicolored independently. $G$ is a planar graph of order $n$, then using the $\mathcal{O}(n^2)$ algorithm described by Robertson et al. in [11], we color $G$ with 4 colors from $\{1, 2, 3, 4\}$. We call these colors *base colors*. We denote by $s_i$ any vertex $s \in V_G$ which has color $i \in \{1, 2, 3, 4\}$, and by $[1, x]_i$ an interval of $x$ (nonnegative integer)

distinct *hues* associated with base color $i$ so that if $i \neq j$ then for every integers $x, y \geq 0$, we have $[1, x]_i \cap [1, y]_j = \emptyset$.

Let $G_\omega$ be a weighted graph associated to $G$. $W_G$ denotes the weighted clique number of $G_\omega$. We fix $\ell = \frac{1}{3}W_G$ (to simplify, we consider the case where $W_G \equiv 0 \pmod 3$, otherwise the result will almost be same), and we let $\mathcal{C} = \bigcup_{i=1}^{4} [1, \ell]_i$ denote a set of available colors. Consider the multicoloring function $f$ of $G_\omega$ defined as follows:

$$\begin{aligned} f : V_G &\longrightarrow \mathcal{P}(\mathcal{C}) \\ s_i &\longmapsto f(s_i) = [1, min(\omega_G(s_i), \ell)]_i \end{aligned}$$

where each vertex $s_i$ with weight $\omega_G(s_i)$ is assigned the hues of $[1, min(\omega_G(s_i), \ell)]_i$. Note that a vertex $v$ (with weight $\omega(v)$) is called *heavy* if $\omega(v) > \ell$ and is called *light* if $\omega(v) \leq \ell$. Hence, only the heavy vertices remain to be completely colored and their weights may be decreased by $\ell$. All light vertices are colored completely and are deleted from $G$.

Let $H = (V_H, E_H)$ denote the remaining graph obtained after this process. Thus $H$ is such that

- $u \in V_H \Leftrightarrow \omega_G(u) > \ell$,
- $\omega_H(u) = \omega_G(u) - \ell$,
- $(u, v) \in E_H \Rightarrow (u, v) \in E_G$.

It is easily seen that $H$ has no clique of size 3. In fact, if $(u_1, u_2, u_3)$ is a triangle in $H$, then these vertices must have been heavy in $G$. Hence, there exist positive integers $\varepsilon_1$, $\varepsilon_2$ and $\varepsilon_3$ such that $\omega_G(u_1) = \ell + \varepsilon_1$, $\omega_G(u_2) = \ell + \varepsilon_2$ and $\omega_G(u_3) = \ell + \varepsilon_3$. As $\omega_G(u_1) + \omega_G(u_2) + \omega_G(u_3) \leq W_G = 3\ell$, we obtain $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 \leq 0$ a contradiction. Consequently, every clique $K$ in $H$ has size at most 2. Furthermore, if $(u, v) \in K$ then $\omega_H(u) + \omega_H(v) = \omega_G(u) - \ell + \omega_G(v) - \ell \leq W_G - 2\ell = \ell$. In addition, if $u$ is an isolated vertex in $H$, then there exists a positive integer $\varepsilon$ such that $\omega_G(u) = \ell + \varepsilon$, and all neighbors of $u$ must be light vertices in $G$. Suppose that $u$ has $i$ for its base color and let $N_j$ be the set of all its neighbors having $j \neq i$ for there base color in $G$. Let $v_j \in N_j$ such that $\omega_G(v_j) = \max_{s \in N_j} \omega(s)$. Further, we have $\omega_G(v_j) = \ell - \varepsilon_{v_j}$ with $0 \leq \varepsilon_{v_j} \leq \ell$. Then $u$ can borrow $\varepsilon_{v_j}$ available colors from $[1, \ell]_j$, which are unused by all vertices of $N_j$. As $\omega_G(u) + \omega_G(v_j) \leq W_G = 3\ell$, we get $\varepsilon - \varepsilon_{v_j} \leq \ell$. For this reason, we consider that each isolated vertex $u \in H$ has $\omega_H(u) = \omega_G(u) - (\ell + \varepsilon_{v_j}) = \varepsilon - \varepsilon_{v_j}$. Thus, $W_H \leq \ell$ and we can therefore distinguish two cases

1. If $H$ is a bipartite graph, then $H$ can be multicolored optimally with exactly $W_H$ colors (see [4, 7]). In this case, to avoid color conflicts, we use a new set $\mathcal{C}'$ of $W_H$ distinct colors. Thus, multicoloring all vertices of $G_\omega$ requires at most $|\mathcal{C}| + |\mathcal{C}'| \leq 4\ell + \ell = \frac{5}{3}W_G$ colors.
2. If $H$ is not a bipartite graph, then $H$ is a planar graph without 3-cycles (triangle-free). Then, using an $\mathcal{O}(nlogn)$ algorithm (See [12]) we color $H$ with at most 3 colors. Thus, there is an algorithm for multicoloring $H$ that requires

at most $\frac{3}{2}W_H$ colors [4]. Similarly, we use a new set $\mathcal{C}'$ of $\frac{3}{2}W_H$ distinct colors. Thus, multicoloring all vertices of $G_\omega$ requires at most $|\mathcal{C}| + |\mathcal{C}'| \leq 4\ell + \frac{3}{2}\ell = \frac{11}{6}W_G$ colors.

## 4 $p^{th}$ Power of a Mesh

In this section, we denote by $M_{n,m} = (V_{n,m}, E_{n,m})$ the square mesh of order $n * m$, and by $M^p_{n,m} = (V^p_{n,m}, E^p_{n,m})$ the $p^{th}$ power of $M_{n,m}$ such that:

- $V^p_{n,m} = \{(i,j) \,|\, 0 \leq i \leq n-1;\ 0 \leq j \leq m-1\}$,
- $E^p_{n,m} = \{((i,j),(i \pm l, j \pm r)) \in (V^p_{n,m})^2 \mid 1 \leq l + r \leq p\}$.

Suppose that $M^p_{n,m}$ is a weighted graph with weighted clique number $W$. For the multicoloring of $M^p_{n,m}$, the method we use is based on a greedy algorithm. In fact, in the beginning, we fix a set of $W$ distinct colors. Next, according to a preset order, each vertex $u$ is multicolored in a greedy manner by assigning any colors currently unused by its neighbors. If the multicoloring of $u$ remains incomplete, we use new colors to complete it. This method prompts Lemma 1.

Let $H = (V, E)$ be a weighted graph and let $x$ be a vertex of $H$ such that $V = K_1 \cup K_2 \cup \{x\}$, where $K_1$ and $K_2$ are two disjoint subsets and $K_1 \cup \{x\}$ and $K_2 \cup \{x\}$ are cliques in $H$ (See Figure 1). We denote by $\omega(K_i)$ the weight of $K_i$ with $i = 1, 2$, by $\omega(x)$ the weight of $x$ and by $\mathcal{C}$ the set of available colors.

**Lemma 1.** *Assume that all vertices of $K_1 \cup K_2$ are already multicolored using colors from $\mathcal{C}$ and assume that the vertex $x$ is not yet multicolored, then the total number of colors obtainable after multicoloring $x$ is at most $max(|\mathcal{C}|, W + \frac{1}{2}|\mathcal{C}|)$.*

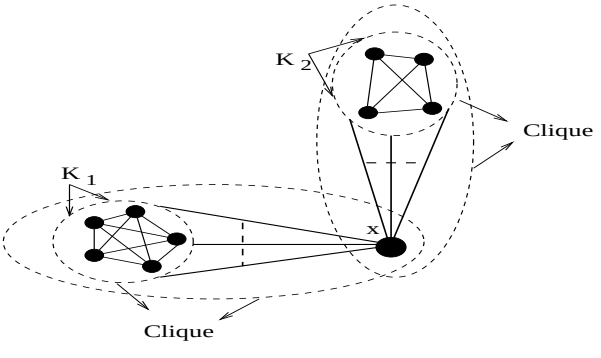



**Fig. 1.** The graph $H$

*Proof.* Suppose that, for each $i \in \{1,2\}$, $K_i \cup \{x\}$ is a clique in $H$. As $W$ is weighted clique number of $H$, we have

$$\begin{cases} \omega(x) + \omega(K_1) \leq W \\ \omega(x) + \omega(K_2) \leq W \end{cases} \tag{1}$$

(1) gives :

$$\omega(x) \leq W - \frac{1}{2}(\omega(K_1) + \omega(K_2)) \tag{2}$$

Let $S$ be the number of colors of $\mathcal{C}$ used only on the vertices of $K_1$ or $K_2$. Let $D$ be the number of colors of $\mathcal{C}$ used on both vertices of $K_1$ and $K_2$. Thus, we get $\omega(K_1) + \omega(K_2) = S + 2D$.

1. If all colors of $\mathcal{C}$ have been used i.e. $|\mathcal{C}| = S + D$, according to (2) we obtain $\omega(x) \leq W - \frac{1}{2}(S + 2D) \leq W - \frac{1}{2}(S + D) = W - \frac{1}{2}|\mathcal{C}|$. Then, to multicolor the vertex $x$ without having a colors conflict, we use $\omega(x) \leq W - \frac{1}{2}|\mathcal{C}|$ new colors. Consequently, the total number of colors used is $\omega(x) + |\mathcal{C}| \leq W + \frac{1}{2}|\mathcal{C}|$ colors.
2. If $|\mathcal{C}| > S + D$, let $A$ be the number of unused colors of $|\mathcal{C}|$. If $\omega(x) \leq A$, we can use some of these $A$ colors to multicolor vertex $x$. In this case, the total number of colors used cannot exceeds $|\mathcal{C}|$. If $\omega(x) > A$, we assign $A$ colors to vertex $x$ and we consider that the new weight of vertex $x$ is $\omega(x) - A$, thus we are in the previous case.

**Theorem 3.** *For any $p \geq 2$, there exists a polynomial time greedy algorithm which multicolors all vertices of the weighted $p^{th}$ power square mesh using at most $2W$ colors.*

*Proof.* Let $M^p_{n,m}$ be the $p^{th}$ power square mesh as defined previously. In what follows, let $\omega_{ij}$ denote the weight of vertex $(i,j)$ and $\mathcal{C}_0$ denote a set of colors with $|\mathcal{C}_0| = W$.

Consider $\mathcal{P}_N = \{(i,j) \in V^p_{n,m} \mid i + j = N\}$, a subset of $V^p_{n,m}$. We observe that $V^p_{n,m} = \bigcup_{N=0}^{n+m-2} \mathcal{P}_N$.

The idea of the proof is to multicolor each subset $\mathcal{P}_N$ starting from $\mathcal{P}_0$ to $\mathcal{P}_{n+m-2}$. At each stage $N$, we multicolor in a greedy manner the vertices of $\mathcal{P}_N$ using Lemma 1. In fact,

- For $N = 0$, let $\mathcal{C}_0$ be the set of available colors. It is easy to see that $\mathcal{P}_0 = \{(0,0)\}$ and we can multicolor the vertex $(0,0)$ with $\omega_{00}$ colors from $\mathcal{C}_0$.
- For $N = 1$, we also consider $\mathcal{C}_0$ as the set of available colors. We have $\mathcal{P}_1 = \{(1,0),(0,1)\}$. As $(0,0)$, $(1,0)$ and $(0,1)$ belong to the same clique, we can easily multicolor the vertices $(1,0)$ and $(0,1)$ with colors from $\mathcal{C}_0$ without having any conflict.
- At stage $N$, let $(i,j) \in \mathcal{P}_N$ be a vertex not yet multicolored and let $\mathcal{C}$ be the set of available colors. We suppose that all vertices of $\mathcal{P}_{N'}$ (with $N' < N$)

and the vertices $(i+r, j-r) \in \mathcal{P}_N$ with $1 \leq r \leq min(N-i-1, j)$ are already multicolored and use colors from $\mathcal{C}$. Let $K_{ij}$ be the subset of $V^p_{n,m}$ that contains part of these vertices which are at distance less than or equal to $p$ from vertex $(i,j)$ in $M_{n,m}$. The goal is to partition $K_{ij}$ into two cliques $K^1_{ij}$ and $K^2_{ij}$ and applying Lemma 1 (See Figure 2).
Now, we are going to show that

$$\begin{aligned} K_{ij} = &\{(i+r, j-l) \in V^p_{n,m} \,|\, r+l \leq p\,; 0 \leq r \leq l\} \\ &\cup \{(i-r, j-l) \in V^p_{n,m} \,|\, r+l \leq p\,; 1 \leq r \leq p\,; 0 \leq l \leq p\} \\ &\cup \{(i-r, j+l) \in V^p_{n,m} \,|\, r+l \leq p\,; 1 \leq l < r\} \end{aligned}$$

Let $(i', j') \in K_{ij}$, it is easy to see that $d_{M_{n,m}}((i,j),(i',j')) \leq p$.
In addition, we have

$$i'+j' = \begin{cases} i+j-(l-r), \text{ if } 0 \leq r \leq l; \\ i+j-(l+r), \, 1 \leq r, \, 0 \leq l; \\ i+j-(r-l), \, 1 \leq l < r. \end{cases} \tag{3}$$

If $N' = i'+j'$ then, (3) gives us $N' \leq N$, because $N = i+j$. However, we have $i'+j' = N$ only if $i' = i+r$ and $j' = j-r$ with $r \leq j$. Hence, according to the above assumption, we have that $(i', j')$ is already multicolored.
Reciprocally, consider $(i \pm r, j \pm l)$, with $r, l \geq 0$, a vertex of $V^p_{n,m}$ already multicolored such that $d_{M_{n,m}}((i,j),(i \pm r, j \pm l)) \leq p$. That means $r+l \leq p$ and there exists $N' \leq N$ such that $(i \pm r, j \pm l) \in \mathcal{P}_{N'}$. So, $i \pm r + j \pm l \leq N = i+j$ this gives $r \leq l$ or $l \leq r$. Hence, $(i \pm r, j \pm l) \in K_{ij}$.
Moreover, we can partition $K_{ij}$ into two cliques $K^1_{ij}$ and $K^2_{ij}$ where:

- $K^1_{ij} = \{(i+r, j-l) \,|\, 1 \leq r+l \leq p\,; 0 \leq r \leq l\} \cup \{(i-r, j-l) \,|\, r+l \leq p\,; 1 \leq r < l\}$.
- $K^2_{ij} = \{(i-r, j-l) \,|\, 1 \leq r+l \leq p\,; 0 \leq l < r\} \cup \{(i-r, j+l), \,|\, r+l \leq p\,; 1 \leq l < r\}$

Finally, by applying Lemma 1, we multicolor $(i,j)$ and the total number of colors used until this stage is $max(|\mathcal{C}|, W + \frac{1}{2}|\mathcal{C}|)$. Then, it is clear that $2W$ colors are sufficient for multicoloring all vertices of $M^p_{n,m}$.

In fact, the algorithm of the above proof can be applied to every subgraph of the power square mesh.

**Proposition 1.** *For any subgraph $G$ of the weighted square mesh, there exists a polynomial time greedy algorithm which multicolors all vertices of $G$ using at most $2W$ colors.*

*Proof.* The proof is similar to that of Theorem 3, because, keeping the same order on the vertices, when we multicolor vertex $(i,j)$ of $G$, the set of neighbors of $(i,j)$ which are already multicolored can be partitioned in two cliques. Thus the same algorithm used in proof of Theorem 3 gives the result.

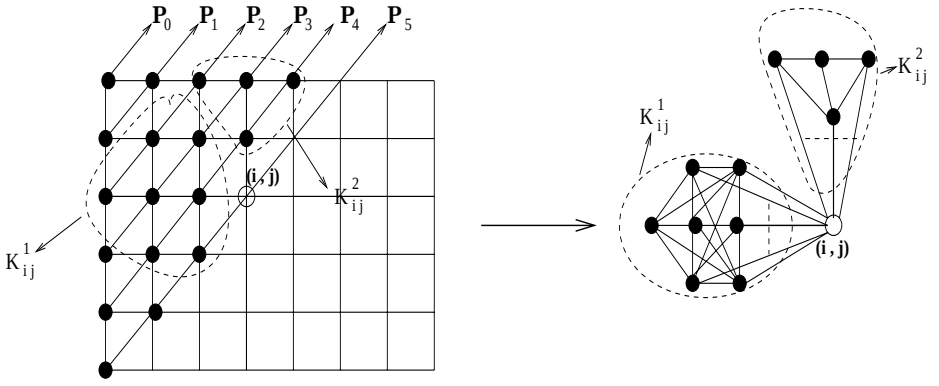



**Fig. 2.** The multicoloring of graph $M^3$

## 5 $p^{th}$ Power of a Triangular Mesh

We define a triangular mesh as a mesh formed by tiling the plane regularly with equilateral triangles (See Figure 3). The multicoloring problem on a weighted triangular mesh has been extensively studied and proved to be NP-hard by Mcdiarmid and Reed [7]. If the triangular mesh considered is of power $p \geq 2$ then the problem models frequency allocation in cellular networks with reuse distance $d$, where $d = p - 1$. Some authors independently gave approximation algorithms for this problem. In case where $d = 2$, a $\frac{4}{3}$-approximation algorithm has been described both in [7, 9]. For $d = 3$, [3] gives a simple algorithm that has a guaranteed approximation ratio of $\frac{7}{3}$. For $d \geq 4$, the best known upper bound on the number of colors needed is $4W$ [6]. In contrast, the best known lower bound on the number of colors needed is $\frac{9}{8}W$ if $d = 2$ [9] and is $\frac{5}{4}W$ if $d \geq 3$ [10].

In the following, we present an improvement of the upper bound of $4W$ by showing that for the $p^{th}$ power triangular mesh noted $H^p$, there exists a polynomial time algorithm that multicolors all vertices of $H^p$ using at most $2W$ colors. The method used is based on the multicoloring of the $p^{th}$ power mesh.

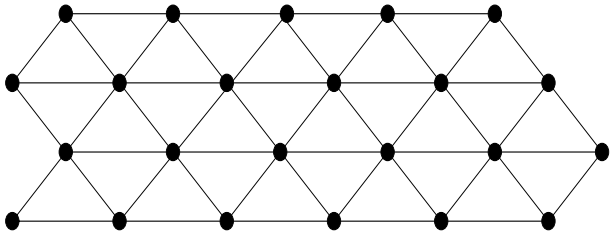


**Fig. 3.** A triangular mesh

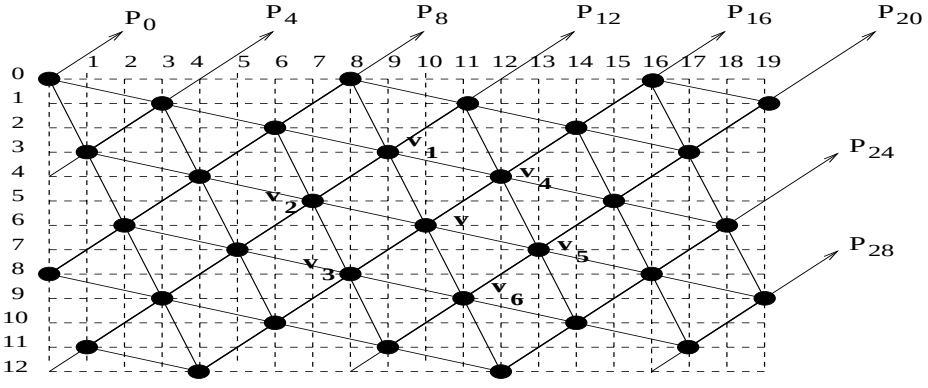



**Fig. 4.** A triangular mesh induced in the $4^{th}$ power of a mesh

**Theorem 4.** *$H^p$ is a subgraph of $M^{4p}$, where $M^{4p}$ is the $(4p)^{th}$ power of a mesh.*

*Proof.* Let $M = (V, E)$ be an arbitrary square mesh, and let $M^p = (V^p, E^p)$ be the $p^{th}$ power of $M$.
We fix $p = 4$. We know that there exists a pair of integers $(q, r)$ such that $n + m - 2 = 4q + r$ where $0 \leq r < 4$.

Let $V_H^1 = \bigcup_{k=0}^{q} \mathcal{P}_{4k}$ be a vertex subset of $V^4$, where for $0 \leq k \leq q$ we have:

- $\mathcal{P}_{4k} = \{(2i, 2j) \in V^4 \mid i + j = 2k\}$ when $k$ is even.
- $\mathcal{P}_{4k} = \{(2i + 1, 2j + 1) \in V^4 \mid i + j = 2k - 1\}$ when $k$ is odd.

Consider $v, v' \in V_H^1$ two adjacent vertices in $M^4$ (i.e. $d_M(v, v') \leq 4$). Thus, there exist $k, k' \in \{0, 1, ..., q\}$ such that $v \in \mathcal{P}_{4k}$ and $v' \in \mathcal{P}_{4k'}$.
Without loss of generality, we assume that $k$ is even. Then there exists $(i, j)$ such that $v = (2i, 2j) \in V^4$ and $i + j = 2k$.

*case 1:* $k'$ is even. Then, there exists $(i', j')$ such that $v' = (2i', 2j') \in V^4$ and $i' + j' = 2k'$. As $d_M(v, v') \leq 4$, we get $|2(i' - i)| + |2(j' - j)| \leq 4$. Moreover, $|2(i' - i) + 2(j' - j)| \leq |2(i' - i)| + |2(j' - j)| \leq 4$, this gives $|4(k' - k)| \leq 4$. As $k$ and $k'$ are even we obtain $k = k'$. Then, $i' - i = j - j'$ so $|i' - i| \leq 1$. Thus, $i = i'$ or $i = i' \pm 1$. If $i = i'$ we obtain $v = v'$, but if $i = i' \pm 1$ we obtain $v' = (2i + 2, 2j - 2)$ or $v' = (2i - 2, 2j + 2)$.

*case 2:* $k'$ is odd. Then, there exists $(i', j')$ such that $v' = (2i' + 1, 2j' + 1) \in V^4$ and $i' + j' = 2k' - 1$. $d_M(v, v') \leq 4$ implies that $|2(i' - i) + 1| + |2(j' - j) + 1| \leq 4$. As $|2(i' - i) + 2(j' - j) + 2| \leq |2(i' - i) + 1| + |2(j' - j) + 1|$, we obtain $|4(k' - k)| \leq 4$. So, $k' = k \pm 1$ because they are of different parity.

If $k' = k + 1$, then $i' - i = j - j' + 1$. So, the inequality $|2(i' - i) + 1| + |2(j' - j) + 1| \leq 4$ gives $j - j' \leq \frac{1}{2}$ or $j' - j \leq \frac{3}{2}$. Hence, we obtain $j' = j$ or $j' = j + 1$.

Therefore, $v' = (2i+3, 2j+1)$ or $v' = (2i+1, 2j+3)$. In the same way if we take $k' = k-1$, we obtain $v' = (2i-3, 2j-1)$ or $v' = (2i-1, 2j-3)$.

Thanks to the above, we see that each inner vertex $v \in \mathcal{P}_{4k}$ with $k$ is even has exactly six neighbors and they form a hexagon, and the same holds when $k$ is odd.

Consequently, we define $H^1$ to be the subgraph of $M^4$ induced by $V_H^1$. It is clear that $H^1$ is a triangular mesh (See Figure 4).

In addition, we remark that the distance in $M$ between two adjacent vertices of $H^1$ is 4. Thus, we easily verify that for any $p \geq 1$, the $p^{th}$ power of the triangular mesh $H^p$ is a subgraph of the $(4p)^{th}$ power of the mesh $M^{4p}$.

**Corollary 1.** *There exists a polynomial time greedy algorithm which multicolor all vertices of the weighted $p^{th}$ of a triangular mesh $H^p$ using at most $2W$ colors.*

## 6 $p^{th}$ Power of a Toroidal Mesh

Let $G = (V, E)$ be an arbitrary graph and let $s$ be a vertex of $G$. We denote by $N(s)$ the neighborhood of vertex $s$.

**Definition 1.** *A connected graph $G = (V, E)$ has Property $(P_k)$ iff for any vertex $s$, the set $N(s) \cup \{s\}$ can be covered with $k$ distinct cliques each containing $s$.*

With the above definition, we provide the following theorem, where the proof is based on an idea given in [3].

**Theorem 5.** *If $G$ is a weighted connected graph verifying Property $(P_4)$ then $\chi_\omega(G) < 4W$.*

*Proof.* Let $G = (V, E)$ be a connected graph verifying Property $(P_4)$. Consider a proper coloring of $G$ with $k$ colors. We associate with each base color $\ell$ distinct hues, where $\ell$ is a constant to be determined later. Thus, we obtain $k \cdot \ell$ available colors. Now, we assign each vertex $s \in V$ $\omega(s)$ colors in the interval $[1, min(\omega(s), \ell)]$ of colors associated to the base color of $s$. Then, we note that the heavy vertices (with weight larger than $\ell$) are not completely colored. In order to complete the coloring of the heavy vertices, we proceed in the following manner. First, we consider a heavy vertex $s$ not yet completely colored and let $N(s)$ be the vertex subset of $V$ adjacent with $s$. As $G$ verifies Property $(P_4)$, $N(s)$ can be covered with four cliques, each of them containing $s$. Assume that $W$ is the weighted clique number. Then each clique has weight at most $W$. Thus, the total weight of $N(s)$ is at most $\omega(s) + 4(W - \omega(s)) = 4W - 3\omega(s) < 4W - 3\ell$ because $s$ is a heavy vertex. In order for $k \cdot \ell$ colors to be sufficient to color all vertices of $N(s)$ we must have $4W - 3\ell = k \cdot \ell$. This gives $4W = (3+k)\ell$. So, $\ell = \frac{4}{3+k}W$. Thus, we are able to color all vertices of $G$ using at most $k \cdot \ell = \frac{4k}{3+k}W < 4W$ colors.

Now, we consider the $p^{th}$ power toroidal mesh $M_{n,m}^p = (V^p, E^p)$ of order $n * m$ where:

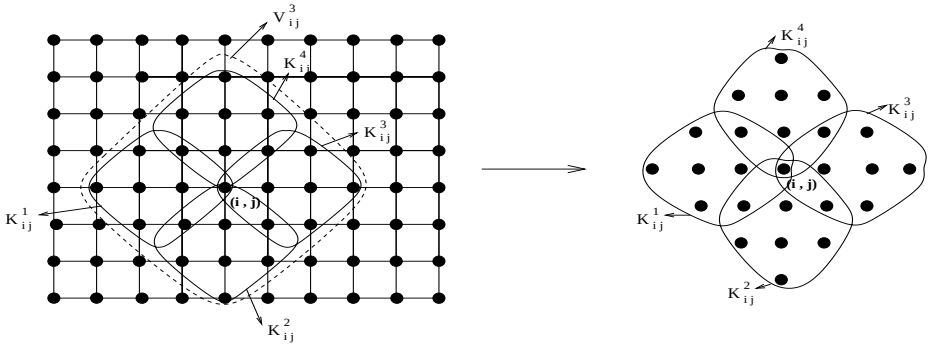



**Fig. 5.** The graph $M^3$ and the vertex subset $V^p_{(i,j)}$ covered by 4 cliques $K^1_{ij}$, $K^2_{ij}$, $K^3_{ij}$, $K^4_{ij}$

- $V^p = \{(i,j) \,|\, 0 \leq i \leq n-1;\, 0 \leq j \leq m-1\}$.
- $E^p = \{((i,j),(i \pm l \ mod\ n\ , j \pm r \ mod\ m)) \,|\, 1 \leq l + r \leq p\}$.

Let $W$ be the weighted clique number of $M^p_{n,m}$.

Using Theorem 5, we show in the following theorem that the weighted chromatic number of the $p^{th}$ power of a toroidal mesh $M^p_{n,m}$ does not exceed $4W$.

**Theorem 6.** *There exists a polynomial time algorithm which multicolors any vertices of the weighted $p^{th}$ power toroidal mesh of order $n*m$ if $p < min(\lceil \frac{n}{2} \rceil, \lceil \frac{m}{2} \rceil)$ using at most $4W$ colors.*

*Proof.* For the proof, with Theorem 5, we only have to show that $M^p = (V^p, E^p)$ verifies Property $(P_4)$.

Let $s = (i,j)$ be a vertex of $V^p$, and let $V^p_s = N(s) \cup \{s\}$ be the vertex subset of $V^p$ which contains all vertices of $V^p$ at distance less or equal to $p$ from $s$. Now, we construct four subsets of $V^p_s$ defined as follow:

- $K^1_{ij} = \{(i \pm r, j - l) \in V^p_s \,|\, 0 \leq r + l \leq p;\, 0 \leq r \leq l\}$.
- $K^2_{ij} = \{(i + r, j \pm l) \in V^p_s \,|\, 0 \leq r + l \leq p;\, 0 \leq l \leq r\}$.
- $K^3_{ij} = \{(i \pm r, j + l) \in V^p_s \,|\, 0 \leq r + l \leq p;\, 0 \leq r \leq l\}$.
- $K^4_{ij} = \{(i - r, j \pm l) \in V^p_s \,|\, 0 \leq r + l \leq p;\, 0 \leq l \leq r\}$.

It is easy to see that each subset $K^l_{ij}$ with $l \in \{1,2,3,4\}$ is a clique in $M^p_{n,m}$ and $V^p_s = K^1_{ij} \cup K^2_{ij} \cup K^3_{ij} \cup K^4_{ij}$ i.e. $V^p_s$ is covered by these 4 cliques (See Figure 5). Then, $M^p_{n,m}$ verifies Property $(P)$.

## 7 Conclusions

In this work, we have studied the problem of frequency assignment in cellular networks with reuse distance $d$ as a multicoloring problem on powers of graphs.

We proposed some techniques of multicoloring based on approximation and distributed algorithms, and we provided some approximation ratios in terms of the number of colors used in relationship to the weighted clique number, denoted by $W$. For a planar graph, we have shown that $\frac{11}{6}W$ colors are sufficient to multicolor it. For the power square mesh and triangular mesh, we described a greedy multicoloring algorithm that uses at most $2W$ colors. We also presented a distributed algorithm for multicoloring a toroidal mesh using at most $4W$ colors. It would be interesting to see if the algorithm proposed in Section 4 can be improved to have an approximation ratio less than 2, and can be adapted to every power $n$-dimensional mesh and every power planar graph.

## Acknowledgments

We wish to thank the anonymous referees for many useful comments and references.

## References

1. M. I. Andreou, S. E. Nikoletseas, P. G. Spirakis. Algorithms and Experiments on Colouring Squares of Planar Graphs. In Proc. *of the 2nd International Workshop on Experimental and Efficient Algorithms (WEA 2003), LNCS 2647, Springer*, 15-32, 2003.
2. I. Caragiannis, C. Kaklamanis, E. Papaioannou. Efficient On-Line Frequency Allocation and Call Control in Cellular Networks. In Proc. *Theory of Computing Systems*, 35(5):521-543, 2002.
3. T. Feder and S.M. Shende. Online channel allocation in FDMA networks with reuse constraints. *Inform. Process. Lett.*, 67(6):295–302, 1998.
4. J. Janssen, K. Kilakos and O. Marcotte. Fixed preference frequency allocation for cellular telephone systems. *IEEE Transactions on Vehicular Technology*, 48(2):533-541, March 1999.
5. J. Janssen, D. Krizanc, L. Narayanan, et S. Shende. Distributed On–Line Frequency Assignment in Cellular Networks. *Journal of Algorithms*, 36(2):119-151, 2000.
6. S. Jordan and E.J. Schwabe. Worst-case preference of cellular channel assignment policies. *Wireless networks*, 2:265-275, 1996.
7. C. McDiarmid and B. Reed. Channel Assignment and Weighted coloring. *Networks*, 36:114-117, 2000.
8. M. Molloy, M. R. Salavatipour. Frequency Channel Assignment on Planar Networks. In Proc. *of the 10th Annual European Symposium (ESA 2002), LNCS 2461, Springer*, 736-747, 2002.
9. L. Narayanan, and S.M. Shende. Static Frequency Assignment In Cellular Networks. *Algoritmica*, 29, 396-409, 2001.
10. L. Narayanan, and Y. Tang. Worst-case analysis of a dynamic channel assignment strategy. *Discrete Applied Mathematics*, 140:115-141, 2004.
11. N. Robertson , D. Sanders , P. Seymour et R. Thomas. The four-colour theorem. *Journal of Combinatorial Theory*, 70(1):2-44, 1997.
12. M. R. Salavatipour. The Three color problem for planar graphs. *Technical Report CSRG-458, Department of Computer Science, University of Toronto*, 2002.

# From Static Code Distribution to More Shrinkage for the Multiterminal Cut

Bram De Wachter*, Alexandre Genon*, and Thierry Massart

Université Libre de Bruxelles,
Département d'Informatique,
Bld du Triomphe, B-1050 Bruxelles
{bdewacht, agenon, tmassart}@ulb.ac.be

**Abstract.** We present the problem of statically distributing instructions of a common programming language, a problem which we prove equivalent to the multiterminal cut problem. We design efficient shrinkage techniques which allow to reduce the size of an instance in such a way that optimal solutions are preserved. We design and evaluate a fast local heuristics that yields remarkably good results compared to a well known $2 - \frac{2}{k}$ approximation algorithm. The use of the shrinkage criterion allows us to increase the size of the instances solved exactly, or to augments the precision of any particular heuristics.

## 1 Introduction

We present the problem of automatic distribution of a programming language, motivated by our research in automatic distributed industrial control systems [17]. This problem consists in distributing a program code among different sites, minimizing the total communications between these sites during its execution. We show that this problem is NP-hard. Furthermore, we show that it is equivalent to the *multiterminal cut* presented in [7], and therefore concentrate on finding new ways to attack the problem described in terms of multiterminal cut.

The key concept used in this paper is based on *shrinkage*, a notion presented by Dahlhaus *et al.* in [7] where an instance $I$ is transformed into a smaller instance $I'$ in such a way that all optimal solutions in $I'$ can be easily transformed into optimal solutions for $I$.

In this paper, we generalize the shrinkage criterion of Dahlhaus et al. which is based on `st` cuts, to all nodes in the instance graph and prove its correctness. Then, we present an implementation of a fast local heuristics taking advantage of this new shrinkage operation. The heuristics combines both the new shrinkage based reduction and an *unshackle* operation which operates on graphs where no more shrinkage is possible.

We also introduce *maximum size* minimum `st` cuts, and prove (theorems 5, 6 and 7) some unexpected properties on the structure of these cuts. We exploit these results to obtain a more efficient implementation for our shrinkage

* Work supported by the *Region de Bruxelles Capitale*, grant no. RBC-BR 227/3298.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 177–188, 2005.
© Springer-Verlag Berlin Heidelberg 2005

algorithm, and prove (theorem 8) that the procedure of Goldberg and Tarjan, presented in [10] for max-flow/min-cut actually computes these cuts. A practical evaluation is presented showing that our heuristics yields generally better results than the approximation algorithm designed by Dahlhaus et al. To the best of our knowledge, we perform the first experimental study of the approximation algorithm of [7].

## 2 Optimal Static Code Distribution Is Hard

Our problem consists in finding, at compile time, an optimal distribution of an imperative regular program. Such a program contains instructions (assignments, loops and tests), a set of static global internal variables and a set of static global I/O variables. The distributed environment in which the program runs is composed of several sites, each of which contains some of the global I/O variables. A correct distribution is an assignment of all variables and all instructions to the set of sites such that the following distribution constraints are satisfied : (1) the I/O variables are on the predefined sites, (2) each variable and instruction is on exactly one site, (3) each instruction using a variable is on the site of that variable.

The assignment of instructions to sites influences the performance of the program during execution: each time control flows from an instruction assigned to one site to an instruction assigned to another site, the executed distribution environment must synchronize (e.g. by sending a message over a network) in order to continue the execution on the other site. The optimal distribution is such that the expected number of (synchronization) messages exchanged during execution is minimum. In order to evaluate this performance criterion, we suppose that a *realistic* control flow frequency function $\mathcal{W}$ is given, expressing the expected number of times control flows from one instruction to another.

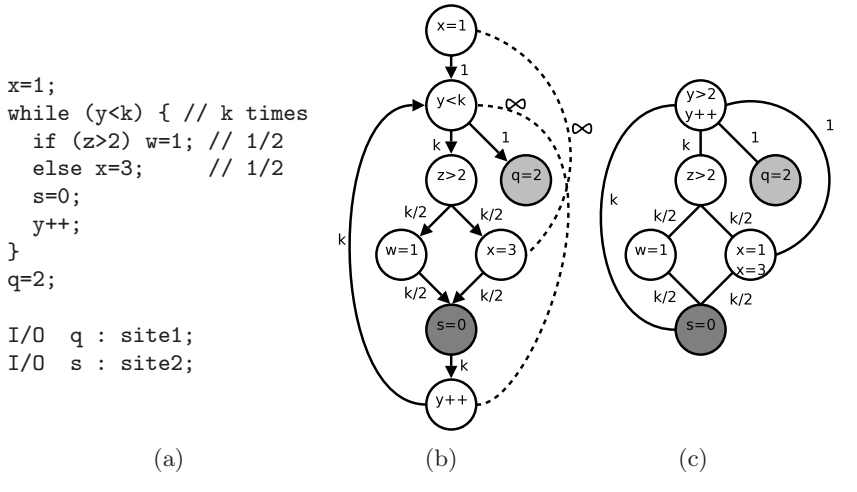



**Fig. 1.** The distribution problem

A graphical presentation of the optimal distribution problem can be found in figure 1. The program of figure 1(a) can be graphically modeled by its control flow graph (figure 1(b)) where the nodes are its instructions and edges model the control flow between instructions with weights defined by $\mathcal{W}$. The graph of figure 1(c) is the undirected graph where all nodes using the same variables are merged. We say that this graph is the result of the merging of `x=1` and `x=3` and of the merging of `y>k` and `y++` in the first graph. More formally, when two nodes $n$ and $n'$ are merged, $n$ and $n'$ are replaced by one new node $n''$, and all edges $\{v, n\}$ and $\{u, n'\}$ are changed to $\{v, n''\}$ and $\{u, n''\}$. Note that when more than one edge exists between two nodes, all of the edges between those nodes can be replaced by a single edge weighted by the sum of the weights of these edges. Remark that both representations of figure 1 are equivalent with respect to the optimal distribution problem. The merging operation is sometimes called *contraction* if an edge exists between two merged nodes, since that edge would disappear from the graph. We now give a formal definition of the multiterminal cut problem on weighted undirected graphs.

**Definition 1 (Multiterminal cut problem).** *Given a weighted undirected graph* $G(V, E, w) : E \subseteq \{\{u, v\} | u, v \in V \wedge u \neq v\}$ [1], $w : E \mapsto \mathbb{N}$ *and a set of terminals* $T = \{s_1, ..., s_k\} \subseteq V$*, find a partition of* $V$ *into* $V_1, ..., V_k$ *such that* $s_i \in V_i \ \forall i \in [1, k]$ *and* $\sum_{v \in V_i, v' \in V_j, i \neq j} w(v, v')$ *is minimized.*

We know that the multiterminal cut problem is NP-Hard [7] for fixed $k > 2$, even when all weights are equal to 1. As shown in [1], optimal distribution is an NP-hard problem. The following theorems, proved in [1], state that the optimal distribution problem and the multiterminal cut are equivalent.

**Theorem 1.** *There exists a polynomial time reduction from the optimal distribution problem to the multiterminal cut.*

**Theorem 2.** *There exists a polynomial time reduction from multiterminal cut on unweighted graphs to the optimal distribution problem.*

With these two theorems, we can conclude that the optimal distribution problem is polynomially equivalent to the multiterminal cut. Thus, to solve the optimal distribution problem, we can concentrate on the multiterminal cut in the program's control flow graph.

## 3 Related Works

The multiterminal cut problem has first been studied by Dahlhaus *et al.* in [7]. In this paper, the authors prove that this problem is NP-hard for $k > 2$ even when $k$ is fixed where $k$ is the number of terminals. The problem is polynomially solvable when $k = 2$, a well known result proved by Ford and Fulkerson [8], and in the case of planar graphs. The authors also present a $2 - \frac{2}{k}$

[1] For technical reasons looping edges $(v, v)$ will be omitted in all graphs considered here.Note that their presence does not change the problem.

polynomial time approximation algorithm that relies on *isolating* cuts, a technique that is detailed further on. Moreover, they proved that this problem is MAX SNP-hard, i.e. there is no polynomial time approximation scheme unless P=NP. In [2], Calinescu, Karloff, and Rabani, presented a linear programming relaxation. Using this technique and a well chosen rounding procedure, they obtain an approximation factor of $1.5 - \frac{1}{k}$. This factor was lowered to 1.3438 by Karger *et al.* in [13] who give better approximations when $k \geq 14$. These improvements were found by studying carefully the integrality gap and giving a more precise rounding procedure. A polyhedral approach [3, 15, 5] and a non-linear formulation [6] have also been studied for the multiterminal cut problem.

Shrinkage has also been studied by Högstedt and Kimelman in [11]. In this paper, the authors give some optimality-preserving heuristics that allow to reduce the size of the input graph by contracting some edges. The shrinkage technique presented here generalizes some of their criteria (such as independent nets and articulation points).

In this paper, we consider the multiterminal cut problem on undirected graphs, but work has also been done on directed graphs. Naor and Zosin presented a 2-approximation algorithm for this problem in [14]. On the other hand, Costa, Letocart and Roupin proved in [4] that multiterminal cuts on acyclic graphs could be computed in polynomial time using a simple flow algorithm. A generalization of multiterminal cut is *minimum multicut* where a list of pairs of terminals is given and we must find a set of edges such that these pairs of terminals are disconnected. Garg *et al.* [9] give a $O(\log k)$-approximation algorithm for this minimum multicut. A survey on multiterminal cuts and its variations can be found in [4].

The applications that rely on the multiterminal cut fall mainly into two domains : the domain of parallel computation and the partitioning of distributed applications. The problems encountered in parallel computation are concerned with the allocation of tasks on different processors. The total load must be partitioned in roughly equal sized pieces, characterized by some load balancing criterion, and this subject to some interconnection criterion that must be minimized ([12] and [16]). These problems can be formulated using the strongly related $k$-cut problem, which asks to partition the graph in $k$ subsets such that crossing edges are minimized. Since this problem has no fixed terminals, it is polynomially solvable, for any fixed $k \geq 3$ [7] and is thus considerably easier than the problem addressed here.

For the distributed applications, the problem is similar, only that it is the several application's components that must be distributed among different processors. Several criteria are studied, such as the inter object communication load of [11]. However, we are not aware of other works that are based on the static distribution of the instructions where the control flow is used to minimize the expected communications load. Because of this fine grain distribution, the scale of our problem is considerably larger than the studies on the partitioning of objects or functions as is the case in *classical* distributed systems. Therefore, we believe that the results of the heuristics presented here are applicable on these smaller instances as well.

## 4 A Generalized Global Criterion

In [7] the authors design a $2-\frac{2}{k}$ approximation algorithm based on the isolation heuristics which uses `st` cuts. An `st` cut (multiterminal cut with $k=2$) divides the graph into two sets $(C,\overline{C})$ where $s \in C$ and $t \in \overline{C}$. The heuristics consists in finding an optimal isolating cut for each of the $k$ terminals $\{s_1,...,s_k\}$ and taking the union of the $k-1$ smallest of these cuts. An optimal isolating cut is a minimum `st` cut where s=$s_i$ and t is the node resulting of the merging of $s_{j\neq i}$. We now introduce the original shrinkage theorem proved by Dahlhaus et al :

**Theorem 3 (Shrinkage).** *Given graph $G(V,E,w)$ with terminals $T=\{s_1,...,s_k\} \subseteq V$. Let $G'_i$ be the graph where all terminals in $T \setminus \{s_i\}$ are merged into t, and $(C,\overline{C})$ the* `st` *cut between $s_i$ and t, then there exists an optimal multiterminal cut $(V_1,...,V_k)$ of G such that $\exists \ell : C \subseteq V_\ell$.*

Theorem 3 allows us to shrink (i.e. to merge) all nodes in $C$ into one node. Shrinkage is clearly an interesting way to attack the multiterminal cut problem. Indeed, we can apply theorem 3 to all terminal nodes in order to shrink the graph. And if one can obtain a relatively small instance, then there may be hope to find the optimal solution by exhaustive search. It can also be used independently of any other algorithm designed to approximate the multiterminal cut problem. We extend theorem 3 to handle more shrinkage as follows :

**Theorem 4 (More shrinkage).** *Given graph $G(V,E,w)$ with terminals $T=\{s_1,...,s_k\} \subseteq V$. Let $v \in V$, and $G'_v$ be the graph where all terminals in $T \setminus \{v\}$ are merged into t, and $(C,\overline{C})$ the minimum* `st` *cut between v and t in $G'_v$ then there exists an optimal multiterminal cut of G $(V_1,...,V_k)$ such that $\exists \ell : C \subseteq V_\ell$.*

*Proof.* Outline (a detailed proof can be found in [1]). Figure 2 illustrates the proof for $l=1$. Proof by contradiction. Take any minimum multiterminal cut $C^*=(V_1,V_2,...,V_k)$ and suppose that $v \in V_1$ but $C \not\subseteq V_1$. Take $C^{*'}=(V_1 \cup C, V_2 \setminus C,...,V_k \setminus C)$. Then we can show that the weight of $C^{*'} \leq$ the weight of $C^*$ by using the weights of the edges between $C$ and $\overline{C}$ and between $C \cap V_1$ and $\overline{C \cap V_1}$ : since $(C,\overline{C})$ is a minimum `st` cut, the border of $V_1$ can be extended to $V_1 \cup C$ without increasing the multiterminal cut's weight. ■

Theorem 3 differs from this theorem because we can apply the former only on terminal nodes, while the latter can be applied to all nodes in the graph, resulting in more shrinkage and therefore smaller graphs.

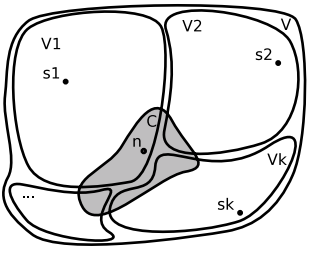



**Fig. 2.** Theorem 4

We now explain how to use theorem 4 to shrink an instance of the multiterminal cut problem. Let $s \in V$, we compute the `st` cut were $t$ is the result of the merging of all terminals in $T \setminus s$. The nodes that are in the same partition as $s$ are merged together, with theorem 4 assuring that this preserves opimality. A chain of graphs $G_1, ..., G_l$ can therefore be calculated where each graph is the result of the optimal merging with respect to its predecessor, and where $G_l$ can not be reduced any further. To compute these `st` cuts, one can use the algorithm of Goldberg and Tarjan [10], with complexity $O(nm \log \frac{n^2}{m})$ (where $n = |V|, m = |E|$). With the results contained in the next section, we can show that when this well known algorithm is used, then $l \leq n$, resulting in a total complexity in $O(n^2 m \log \frac{n^2}{m})$.

Once a graph cannot be reduced any further, two options remain, either search exhaustively and find an optimal solution, or *unshackle* the graph. Unshackling means contracting one or more edges that likely connect nodes from the same partition in the optimal cut. Note that if an edge is picked that is in every optimal multiterminal cut, this operation will not preserve optimality. Once the graph is *unshackled*, the resulting graph may be ready for further optimal reductions. In the following section, we study an implementation using the shrinkage technique combined with a fast local unshackling heuristics.

## 5 A Fast Local Heuristics

As said in previous section, we can use the shrinkage technique in combination with an unshackling heuristics. Figure 3 gives a graphical overview of this technique and figure 4 presents an implementation. We first perform shrinkage until the graph cannot be reduced any further. Then, we use an unshackling heuristics to contract one edge from this graph. The shrinkage technique may thereupon be reused on this *unshackled* graph. This process is repeated until the graph contains only terminal nodes. While the resulting multiterminal cut may not be optimal, due to the unshackling heuristics, we will see that this technique generally computes a fairly good multiterminal cut and is quite efficient, provided that the unshackling is easy to compute.

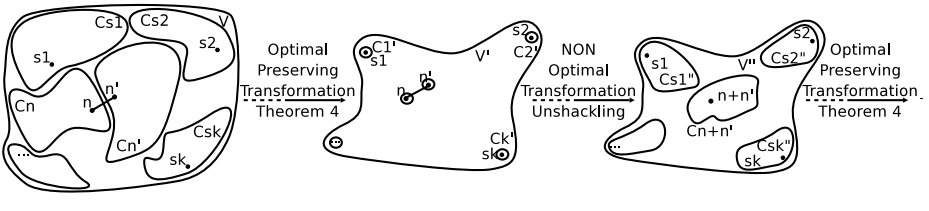



**Fig. 3.** Optimal and non optimal reductions

### 5.1 Definition and Complexity

**Definition 2 (MAX-MIN-`st` cut).** *Given graph $G(V, E, w)$ and two different nodes $s, t \in V$. Define min-ST$(s, t)$ as the set of cuts separating $s$ and $t$ with*

```
reduce();
while(non-terminals exist) {
    unshackle(); // Contract 1 edge
    reduce();
}
```

**Fig. 4.** Unshackling heuristics

*minimum weight. We define the set* MAX-MIN-`st` $(s,t)$ *as the set of nodes left connected to* $s$ *by the cut* $(C,\overline{C}) \in$ *min-ST*$(s,t)$ *such that* $|C|$ *is maximal. We can easily extend these definitions for sets of nodes. For a set* $T$, MAX-MIN-`st` $(s,T)$ *is equivalent to* MAX-MIN-`st` $(s,t)$ *in the graph* $G$ *where all nodes in* $T$ *have been merged into the new node* $t$.

We now prove some interesting properties related to maximum size minimum cuts. Theorems 5, 6 and 7 give some remarkable insights on the structure of these cuts, which leads to a more efficient implementation of our heuristics. We use the following notation :
$w(A,B) = \sum_{\{(x,y)|x\in A, y\in B\}} w(x,y)$, and let $w(X) = w(X,\overline{X})$, where $\overline{X} = V \backslash X$.

**Theorem 5.** *Given graph* $G(V,E,w)$ *and two nodes* $s,t \in V$, MAX-MIN-`st`$(s,t)$ *is uniquely defined, i.e. there is only one maximal size minimum* `st` *cut for any couple* $(s,t)$.

*Proof.* Outline (a full proof can be found in [1]). By contradiction : suppose $S$ and $S'(S \neq S')$ both satisfy the definition of MAX-MIN-`st`$(s,t)$. Let $I = S \cap S'$ and $T = V \setminus (S \cup S')$, it is easy to see that $S$ (resp. $S'$) $\neq I$, else $S$ (resp. $S'$) would not be a maximal size minimum `st` cut. First, since $S$ is a min `st` cut we start from $w(I) \geq w(S)$ to prove that $w(S \setminus I, I) \geq w(S \setminus I, T)$. Next, we compute $w(S \cup S')$, note that $S \cup S'$ is also an `st` cut for $(s,t)$. We show that $w(S \cup S') = w(S \setminus I, T) + w(S', T) \leq w(S')$. Finally, observe that $S \cup S'$ is therefore a minimum `st` cut for $(s,t)$ with larger cardinality than $S$ or $S'$. ■

**Theorem 6.** *For any three nodes* $s, s', t$ *of* $V$, *if* $s' \in$ MAX-MIN-`st`$(s,t)$, *then* MAX-MIN-`st` $(s',t) \subseteq$ MAX-MIN-`st` $(s,t)$.

*Proof.* By contradiction : let $S =$ MAX-MIN-`st`$(s,t), S' =$ MAX-MIN-`st`$(s',t)$ and suppose that $S' \not\subseteq S$. We have that $|S \cup S'| > |S|$. Let $I = S \cap S'$ and $T = V \setminus (S \cup S')$, we define the following :

$$\begin{array}{lll} A \equiv w(S \setminus I, T) & B \equiv w(I,T) & C \equiv w(S' \setminus I, T) \\ D \equiv w(I, S \setminus I) & E \equiv w(I, S' \setminus I) & F \equiv w(S \setminus I, S' \setminus I) \end{array}$$

$$\Longrightarrow \quad \begin{array}{ll} w(S \cup S') = A + B + C & w(S) = A + B + E + F \\ w(S') = B + C + D + F & w(I) = B + D + E \end{array}$$

From the definition of $S$ and $S'$ we have $w(S \cup S') > w(S)$ (as $S \subsetneq S \cup S'$) and $w(I) \geq w(S')$, which implies that $C > E + F$ and $E \geq C + F$. This leads to a contradiction. ■

**Theorem 7.** *Given graph $G(V, E, w)$ and three distinct nodes $s, s', t \in V$. Let $S =$ MAX-MIN-st$(s,t)$, $S' =$ MAX-MIN-st$(s',t)$, $I = S \cap S'$, and $T = V \setminus (S \cup S')$. If $I \neq \emptyset$ and $S \neq I$ and $S' \neq I$, then $w(I, S \setminus I) = w(I, S' \setminus I)$. Moreover, we have that $w(I, V \setminus (S \cup S')) = 0$. The same results hold when $S =$ MAX-MIN-st$(s, \{t \cup s'\})$, $S' =$ MAX-MIN-st$(s', t)$ or $S =$ MAX-MIN-st$(s, \{t \cup s'\})$, $S' =$ MAX-MIN-st$(s', \{s \cup t\})$.*

*Proof.* By contradiction : let's reuse the equations from proof of theorem 6, to compute $w(S \setminus I)$ and $w(S)$ :

$$w(S \setminus I) = A + D + F \qquad w(S) = A + B + E + F$$

As $S$ is the MAX-MIN-st cut$(s,t)$, we have $A + B + E + F \leq A + D + F$ and $B + E \leq D$. By applying a similar reasoning with $S'$ and $S' \setminus I$, we can prove that $B + D \leq E$. In conclusion, we have $E = D (\Rightarrow w(I, S \setminus I) = w(I, S' \setminus I))$ and $B (\equiv w(I,T)) = 0$. The two other propositions are proved likewise. ■

Theorems 5, 6 and 7 allow us to efficiently calculate the reduction phases of our unshackling heuristics. We know that the order in which we calculate the cuts has no effect on the outcome of the algorithm. Moreover, we can calculate the MAX-MIN-st cut for a given node $n$ and immediately merge all nodes on the same side of $n$ in the cut, thus reducing the number of nodes before calculating the next MAX-MIN-st cut for the remaining unmodified nodes. After the calculation and merging of all MAX-MIN-st cuts, we have for all nodes in the reduced graph and terminals $s_1, \ldots, s_k$, MAX-MIN-st cut $(s, \cup_i s_i \setminus \{s\}) = \{s\}$. The only missing link is how to calculate MAX-MIN-st:

**Theorem 8.** *The algorithm of Goldberg and Tarjan [10] calculating the maximum flow in $O(nm \log(\frac{n^2}{m}))$-time also calculates* MAX-MIN-st

*Proof.* By contradiction. As for prerequisites, the reader is expected to be familiar with [10], where the authors prove that it is possible to calculate a minimum st cut $(S_g, \overline{S}_g)$ with $s \in S_g \wedge t \in \overline{S}_g$ in $O(nm \log(\frac{n^2}{m}))$-time. We will use their notations to prove that the min st cut calculated by their algorithm is in fact the unique minimum st cut of maximal size.

Let $g(v,w) : E \mapsto \mathbb{R}^+$ be the preflow function (here we may suppose that the algorithm terminated and that the preflow is a legal flow). $G_g$ is used to indicate the residual graph and $c(v,w) : E \mapsto \mathbb{R}^+$ indicates the capacities of the edges in $E$. In addition, $(S_g, \overline{S}_g)$ is defined as the partition of $V$ such that $\overline{S}_g$ contains all nodes from which $t$ is reachable in $G_g$ and $S_g = V \setminus \overline{S}_g$.
We use the following lemma by Golberg and Tarjan from [10]:

> *When the first stage terminates, $(S_g, \overline{S}_g)$ is a cut such that every pair $v, w$ with $v \in S_g$ and $w \in \overline{S}_g$ satisfies $g(v,w) = c(v,w)$.*

Suppose that there exists another minimum cut $(C', \overline{C'})$ such that $|C'| > |S_g|$ which is maximal in size.
Remark that $S_g \subseteq C'$ because of theorem 6 and $s \in C' \cap S_g$.
Let $I = C' \cap \overline{S}_g$. Note that $I \neq \emptyset$ since $|C'| > |S_g|$. We split the boundaries between $S_g, I$ and $\overline{S}_g$ in three sets :

- Old Boundary: $O \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in I$
- New Boundary: $N \subseteq E = (v, w) : v \in I \wedge w \in \overline{S}_g \setminus I$
- Common Boundary: $C \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in \overline{S}_g \setminus I$

By definition of $(S_g, \overline{S}_g)$, we know that $g(O) = c(O)$. We also know that since $(C', \overline{C'})$ and $(S_g, \overline{S}_g)$ are both minimum cuts : $w(O) + w(C) = w(N) + w(C) \Rightarrow w(O) = w(N)$. Remark that since $g$ is a legal flow, the flow entering $I$ must be equal to the flow getting out of $I$, which means that $g(O) = g(N)$.

The combination of these tree equations leads to a contradiction: since the edges in $N$ are saturated, $t$ is not reachable from any $n \in I$ in $G_g$ which means that $I = \emptyset$. ■

Finally, we can prove that the worst execution time for the unshackling heuristics stays within the complexity of the reduction algorithm :

**Theorem 9.** *The unshackling algorithm from figure 4 can be implemented with worst case complexity* $O(n^2 m \log(\frac{n^2}{m}))$ *if the complexity of* `unshackle()` *is in* $O(nm \log(\frac{n^2}{m}))$.

*Proof.* Consider an irreducible graph in which one and only one edge $\{v_1, v_2\}$ is contracted. Before contraction, $\forall v \in V$ : MAX-MIN-`st` $(v, T) = \{v\}$. It is easy to see (proof by contradiction) that after contraction $\forall v \in V \setminus \{v_1, v_2\}$ : MAX-MIN-`st` $(v, T) = \{v\}$, i.e. the contraction only affects the resulting node from the contraction, which means that after each contraction only one MAX-MIN-`st` cut has to be calculated. Since at most $n$ contractions are possible, the number of MAX-MIN-`st` calculations needed can be bounded by $2n$ ($n$ for the initial reduction and $n$ for all subsequent reductions). The worst case complexity is therefore as stated, if the complexity of `unshackle()` is $O(nm \log(\frac{n^2}{m}))$. ■

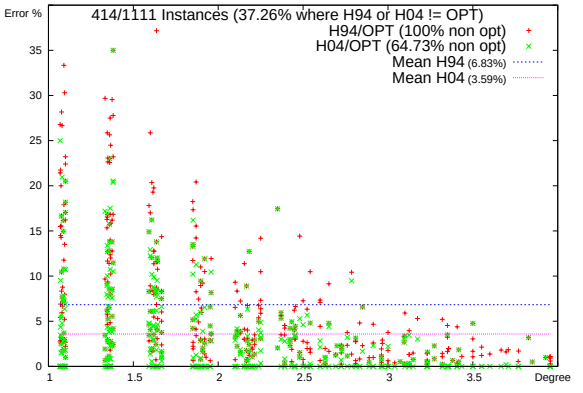
### 5.2 Results

It remains to define the way we will unshackle the graph. We tried several local procedures, among which :

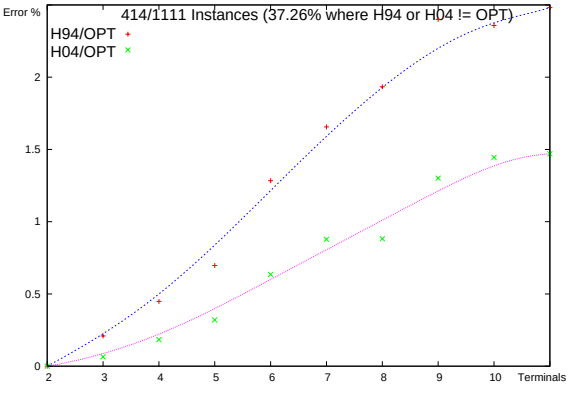
- *Greedy* : take an edge with maximal weight.
- *Error-reduction* : take an edge where the expected error is small
- *Balanced weight* : take an edge $\{n_1, n_2\}$ such that $\sum_{\{\{n_1, n'\} \in E\}} w(n_1, n') + \sum_{\{\{n_2, n'\} \in E\}} w(n_2, n')$ is maximal
- *Max-unshackle* : take an edge that has high reduction rate

Surprisingly, we found that *balanced weight* works much better than the others. Unfortunately, we discovered that none of these heuristics have a fixed approximation bound, however, since our calculations include the ones from the $k - \frac{2}{k}$ approximation algorithm, we can compare the results and take the better of both, resulting in the same bound without any extra cost. In order to compare both heuristics, this has not been done in the following experiments.

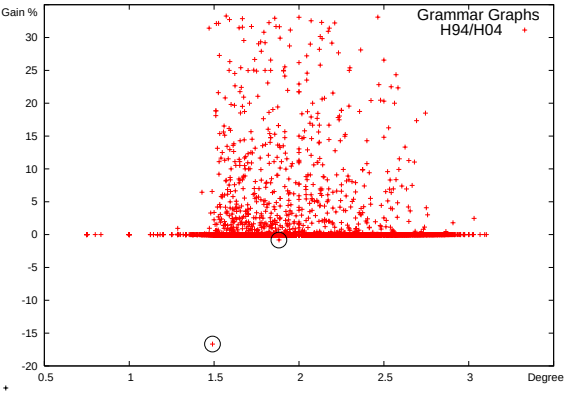
Two sets of experiments were conducted : figures 5(a), 5(b) show the results on random graphs, while 5(c) shows the results on graphs obtained from auto generated programs.




(a) Unshackling heuristics on random graphs




(b) Unshackling heuristics on random graphs




(c) Unshackling heuristics on grammar graphs

**Fig. 5.** Results for the unshackling heuristics

In figure 5(a) we compare the results of our heuristics (indicated by $\mathcal{H}04$) with the approximation algorithm (called $\mathcal{H}94$) from Dahlhaus et al. 1111 experiments where conducted on sufficiently small graphs (ranging from 20-40 nodes), allowing us to compare with the optimal solution. For 411 *hard* cases (37%), one of the heuristics failed to find the optimal. We can see that for increasing mean degree (X-axis), the error rate (Y-axis, in percent w.r.t. the optimal) for both algorithms drops rapidly, caused by the randomness in the graph. For sparse graphs however, error rates can be as high as 35%. The mean error rate for $\mathcal{H}04$, for these *hard* cases, is 3.6% while it raises to 6.8% for $\mathcal{H}94$. Remark that the failure rate for $\mathcal{H}94$ is 100% of the hard cases, while our algorithm failed in 65% of these cases. In these experiments, there was no instance where $\mathcal{H}04$ performed worse compared to $\mathcal{H}94$.

Figure 5(b) shows the mean error rate (Y-axis, in % w.r.t. the optimal solution) for the experiments of figure 5(a), with increasing number of terminals (X-axis). We can clearly see the gain of our algorithm.

Figure 5(c) shows results for 25.000 grammar graphs of moderate size (600 nodes, 3 to 10 terminals), where the two algorithms are compared to each other. X-axis gives the mean degree. We can observe a difference of as high as +35% (Y-axis) for some cases, meaning that our algorithm improves the other by the same amount. For only 2 instances, our algorithm performed worse (1.3% worse and 14% worse).

## 6 Conclusions

In this paper, we studied the problem of optimal code distribution of an imperative regular program, a problem equivalent to the multiterminal cut problem.

We presented a criterion that allows to perform shrinkage on a given graph such that optimal solutions for the multiterminal cut problem on the resulting graph are optimal solutions for the original graph. This criterion is a generalization of the criterion of Dahlhaus *et al.* presented in [7]. Using this shrinkage criterion, we designed a heuristics that, in practice, finds near optimal solutions for the multiterminal cut problem. In our search for an efficient implementation for our shrinkage algorithm, we defined maximum size minimum cuts for wich we prove some interesting structural properties.

*Future works.* We hope that structural properties can lead to a more thorough understanding of the combinatorial structure of optimal solutions for the multiterminal cut. Furthermore, we are currently searching other unshackling procedures that could be combined with our shrinkage technique. Moreover, we are trying to determine why the *balanced weight* unshackling performs well in practice.

We are also considering the use of our shrinkage technique in the branch and bound context. We are currently integrating our shrinkage technique into the CPLEX solver in order to speed up the general branch and bound phase of the mixed integer optimizer.

## References


1. T. Massart B. DeWachter, A. Genon. From static code distribution to more shrinkage for the multiterminal cut. Technical Report 007, U.L.B., December 2004.
2. Gruia Calinescu, Howard Karloff, and Yuval Rabani. An improved approximation algorithm for multiway cut. *J. Comput. Syst. Sci.*, 60(3):564–574, 2000.
3. Sunil Chopra and Jonathan H. Owen. Extended formulations for the a-cut problem. *Math. Program.*, 73:7–30, 1996.
4. M. Costa, L. Letocart, and F. Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 162(1):55–69, 2005.
5. W.H. Cunningham. The optimal multiterminal cut problem. *DIMACS series in discrete mathematics and theoretical computer science*, 5:105–120, 1991.
6. R. Vohra D. Bertsimas, C. Teo. Nonlinear formulations and improved randomized approximation algorithms for multicut problems. In *Proc. 4th conference on integer programming and combinatorial optimization*, volume 920 of *LNCS*, pages 29–39, 1995.
7. Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.
8. L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.
9. Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway cuts in directed and node weighted graphs. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, pages 487–498. Springer-Verlag, 1994.
10. Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, October 1988. ISSN:0004-5411.
11. K. Hogstedt and D. Kimelman. Graph cutting algorithms for distributed applications partitioning. *SIGMETRICS Performance Evaluation Review*, 28(4):27–29, 2001.
12. Lisa Hollermann, Tsan sheng Hsu, Dian Rae Lopez, and Keith Vertanen. Scheduling problems in a practical allocation model. *J. Comb. Optim.*, 1(2):129–149, 1997.
13. David R. Karger, Philip Klein, Cliff Stein, Mikkel Thorup, and Neal E. Young. Rounding algorithms for a geometric embedding of minimum multiway cut. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 668–678, 1999.
14. J. Naor and L. Zosin. A 2-approximation algorithm for the directed multiway cut problem. *SIAM J. Comput.*, 31(2):477–482, 2001.
15. M.R. Rao S. Chopra. On the multiway cut polyhedron. *Networks*, 21:51–89, 1991.
16. Tsan sheng Hsu, Joseph C. Lee, Dian Rae Lopez, and William A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.
17. Bram De Wachter, Thierry Massart, and Cédric Meuter. dsl : An environment with automatic code distribution for industrial control systems. In *Principles of Distributed Systems, 7th International Conference, OPODIS 2003*, volume 3144 of *LNCS*, pages 132–145. Springer, 2004.

# Partitioning Graphs to Speed Up Dijkstra's Algorithm

Rolf H. Möhring[1], Heiko Schilling[1], Birk Schütz[2], Dorothea Wagner[2], and Thomas Willhalm[2]

[1] Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany
[2] Universität Karlsruhe, Fakultät für Informatik, Postfach 6980,76128 Karlsruhe, Germany

**Abstract.** In this paper, we consider Dijkstra's algorithm for the point-to-point shortest path problem in large and sparse graphs with a given layout. In [1], a method has been presented that uses a partitioning of the graph to perform a preprocessing which allows to speed-up Dijkstra's algorithm considerably.

We present an experimental study that evaluates which partitioning methods are suited for this approach. In particular, we examine partitioning algorithms from computational geometry and compare their impact on the speed-up of the shortest-path algorithm. Using a suited partitioning algorithm speed-up factors of 500 and more were achieved.

Furthermore, we present an extension of this speed-up technique to multiple levels of partitionings. With this multi-level variant, the same speed-up factors can be achieved with smaller space requirements. It can therefore be seen as a compression of the precomputed data that conserves the correctness of the computed shortest paths.

## 1 Introduction

We consider the problem of repeatedly finding shortest paths in a large but sparse graph with given arc weights. Dijkstra's algorithm [2] solves this problem efficiently with a sub-linear running time as the algorithm can stop once the target node is reached. If the graph is static, a further reduction of the search space can be achieved with a preprocessing that creates additional information. More precisely, we consider the approach to enrich the graph with arc labels that mark, for each arc, possible target nodes of a shortest paths that start with this arc. Dijkstra's algorithm can then be restricted to arcs whose label mark the target node of the current search, because a sub-path of a shortest path is also a shortest path.

This concept has been introduced in [3] for the special case of a timetable information system. There, arc labels are angular sectors in the given layout of the train network. In [4], the approach has been studied for general weighted graphs. Instead of the angular sectors, different types of convex geometric objects are implemented and compared.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 189–202, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A different variation has been presented in [1], where the graph is first partitioned into regions. Then an arc-flag then consists of a bit-vector that marks the regions containing target nodes of shortest paths starting with this arc. Usually, arc-flags result in a much smaller search space for the same amount of preprocessed data. Furthermore, the generation of arc-flags can be realized without the computation of all-pairs shortest paths in contrast to the geometric objects of [4]. In fact, only the distances to nodes are needed that lie on the boundary of a region. (See [1] for details.) However in [5], it has been shown that partitioning of the graph with METIS [6] generally results in a better reduction than for the partitioning algorithms of [1].

The first contribution of this paper is a computational study whether partitioning algorithms from computational geometry can be used for the arc-flag approach and how they compare to the results of METIS. The algorithms are evaluated with large road networks, the typical application for this problem.

As a second contribution of this paper, we present a multi-level version of arc-flags that produces the same speed-up with lower space consumption. Therefore, these multi-level arc-flags can be seen as a (lossy) compression of arc-flags. Note that the compression still guarantees the correctness of a shortest-path query but may be slower than the uncompressed arc-flags.

We start in Sect. 2 with some basic definitions and a precise description of the problem and the pruning of the search space of Dijkstra's algorithm with arc-flags. In Sect.3, we present the selection of geometric partitioning algorithms that we used for our analysis. We discuss the two-level variant of the arc-flags in Sect. 4. In Sect. 5, we describe our experiments and we discuss the results of the experiments in Sect. 6. We conclude the paper with Sect. 7.

## 2 Definitions and Problem Description

### 2.1 Graphs

A directed simple *graph* $G$ is a pair $(V, A)$, where $V$ is a finite set of *nodes* and $A \subseteq V \times V$ are the *arcs* of the graph $G$. Throughout this paper, the number of nodes $|V|$ is denoted by $n$ and the number of arcs $|A|$ is denoted by $m$. A *path* in $G$ is a sequence of nodes $u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in A$ for all $1 \leq i < k$. A path with $u_1 = u_k$ is called a *cycle*. A graph (without multiple arcs) can have up to $n^2$ arcs. We call a graph *sparse*, if $m \in O(n)$. We assume that we are given a *layout* $L : V \to \mathbb{R}^2$ of the graph in the Euclidean plane. For ease of notation, we will identify a node $v \in V$ with its location $L(v) \in \mathbb{R}^2$ in the plane.

### 2.2 Shortest Path Problem

Let $G = (V, A)$ be a directed graph whose arcs are *weighted* by a function $l : A \to \mathbb{R}$. We interpret the weights as *arc lengths* in the sense that the *length of a path* is the sum of the weights of its arcs. The *(single-source single-target) shortest-path problem* consists in finding a path of minimum length from a given

source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if $G$ does not contain negative cycles. If there are negative weights but not negative cycles, it is possible, using Johnson's algorithm [7], to convert in $O(nm + n^2 \log n)$ time the original arc weights $l : A \to \mathbb{R}$ to non-negative arc weights $l' : A \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, in the rest of the paper, we can safely assume that arc weights are non-negative. Throughout the paper we also assume that for all pairs $(s, t) \in V \times V$, the shortest path from $s$ to $t$ is unique.[1]

### 2.3 Dijkstra's Algorithm with Arc-Flags

The classical algorithm for computing shortest paths in a directed graph with non-negative arc weights is that of Dijkstra [2]. For the general case of arbitrary non-negative arc lengths, it still seems to be the fastest algorithm with $O(m + n \log n)$ worst-case time. However, in practice, speed-up techniques can reduce the running time and often result in a sub-linear running time. They crucially depend on the fact that Dijkstra's algorithm is label-setting and that it can be terminated when the destination node is settled. (Therefore, the algorithm does not necessarily search the whole graph.)

If one admits a preprocessing, the running time can be further reduced with the following insight: Consider, for each arc $a$, the set of nodes $S(a)$ that can be reached by a shortest path starting with $a$. It is easy to verify that Dijkstra's algorithm can be restricted to the sub-graph with those arcs $a$ for which the target $t$ is in $S(a)$. However, storing all sets $S(a)$ requires $O(nm)$ space which results in $O(n^2)$ space for sparse graphs with $m \in O(n)$ and is thus prohibitive in our case. We will therefore use a partition of the set of nodes $V$ into $p$ regions for an approximation of the set $S(a)$. Formally, we will use a function $r : V \to \{1, \ldots, p\}$ that assigns to each node the number of its region. (Given a 2D layout of the graph, a simple method to partition a graph is to use a regular grid as illustrated in Figure 1 and assign all nodes inside a grid cell the same number.) We will now use a bit-vector $b_a : \{1, \ldots, p\} \to \{\texttt{true}, \texttt{false}\}$ with $p$ bits, each of which corresponds to a region. For each arc $a$, we therefore set the bit $b_a(i)$ to `true` iff $a$ is the beginning of a shortest path to at least one node in region $i \in \{1, \ldots, p\}$. For a specific shortest-path query from $s$ to $t$, Dijkstra's algorithm can be restricted to the sub-graph $G_t$ with those arcs $a$ for which the bit of the target-region is set to `true`. (For all edges on the shortest path from $s$ to $t$ the arc-flag for the target region is set, because a sub-path of a shortest path is also a shortest path.)

The sub-graph $G_t$ can be computed on-line during Dijkstra's algorithm. In a shortest-path search from $s$ to $t$, while scanning a node $u$, the modified algorithm considers all outgoing arcs but ignores those arcs which have not set the bit for the region of the target node $t$. All possible partitions of the nodes lead to a correct solution but most of them would not lead to the desired speed-up of the computation.

[1] This can be achieved by adding a small fraction to the arc weights, if necessary.

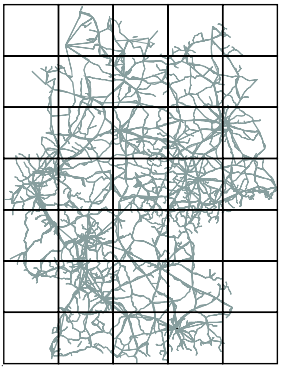


**Fig. 1.** A $5 \times 7$ grid partitioning of Germany

The space requirement for the preprocessed data is $\mathcal{O}(p \cdot m)$ for $p$ regions because we have to store one bit for each region and arc. If $p = n$ and we assign to every node its own region number, we store in fact all-pairs shortest paths: if a node is assigned to its own, specific region, the modified shortest-path algorithm will find the direct path without regarding unnecessary arcs or nodes. Note however, that in practice even for $p \ll n$ we achieved an average search space that is only 4 times the number of nodes in the shortest path. Furthermore, it is possible within the framework of the arc-flag speed-up technique to use a specific region only for the most important nodes. Storing the shortest paths to important nodes can therefore be realized without any additional implementation effort. (It is common practice to cache the shortest paths to the most important nodes in the graph.)

## 3 Partitioning Algorithms

The arc-flag speed-up technique uses a partitioning of the graph to precompute information on whether an arc may be part of a shortest path. Any possible partitioning can be used and the algorithm returns a shortest path, but most partitions do not lead to an acceleration. In this section, we will present the partitioning algorithms that we examined. Most of these algorithms need a 2D layout of the graph.

### 3.1 Grid

Probably the easiest way to partition a graph with a 2D layout is to use regions induced by a grid of the bounding box. Each grid cell defines one region of the graph. Nodes on a grid line are assigned to an arbitrary but fixed grid cell. Figure 1 shows an example of a $5 \times 7$ grid.

Arc-flags for a grid can be seen as a *raster image* of $S(u, v)$, where $S(u, v)$ represents the set of nodes $x$ for which the shortest $u$-$x$ path starts with the arc

$(u, v)$. The pixel $i$ in the image is set, iff $(u, v)$ is the beginning of a shortest path to a node in region $i \in \{1, \ldots, p\}$. A finer grid (i.e., an image with higher resolution) provides a better image of $S(u, v)$, but requires more memory. (On the other hand, the geometric objects in [4] approximate $S(u, v)$ by a single convex object of constant size.)

The grid partitioning method uses only the bounding box of the graph—all other properties like the structure of the graph or the density of nodes are ignored and hence it is not surprising that the grid partitioning always has the worst results in our experiments. Since [1, 5] include this partitioning method, we use the grid partitioning as a baseline and compare all other partitioning algorithms with it.

### 3.2 Quadtrees

A *quadtree* is a data structure for storing points in the plane. Quadtrees are typically used in computational geometry for range queries and have applications in computer graphics, image analysis, and geographic information systems.

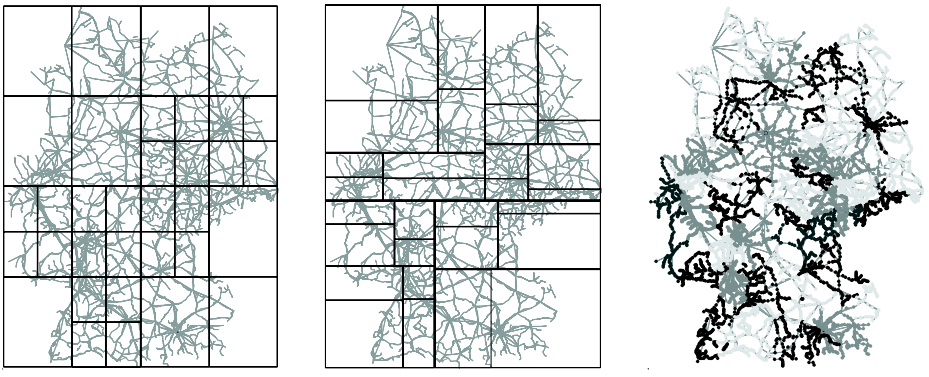
**Definition 1 (Quadtree).** *Let $P$ be a set of points in the plane and $R_0$ its bounding-box. Then, the data structure* quadtree *is a rooted tree of rectangles, where*

- *the root is the bounding region $R_0$, and*
- *$R_0$ and all other regions $R_i$ are recursively divided into the four quadrants, while they contain more than one point of $P$.*

The leaves of a quadtree form a subdivision of the bounding-box $R_0$. Even more, the leaves of every sub-tree containing the root form such a subdivision. Since, for our application, we do not want to create a separate region for each node, we use a sub-tree of the quadtree. More precisely, we define an upper bound $b \in \mathbb{N}$ of points in a region and stop the division if a region contains less points than this bound $b$. This results in a partition of our graph where each region contains at most $b$ nodes. Fig. 2(a) shows such a partition with 34 regions. In contrast to the grid-partition, this partitioning reflects the geometry of the graph—dense parts will be divided into more regions than sparse parts.

### 3.3 Kd-Trees

In the construction of a quadtree, a region is recursively divided into four equally-sized sub-regions. However, equally-sized sub-regions do not take into account the distribution of the points. This leads to the definition of a *kd-tree*. In the construction of a *kd-tree*, the plane is recursively divided in a similar way as for a quadtree. The underlying rectangle is decomposed into *two halves* by a straight line parallel to an axis. The directions of the dividing line alternate. The positions of the dividing line can depend on the data. Frequently used positions are given by the center of the rectangle (*standard* kd-tree), the *average*, or the *median* of the points inside. (Fig. 2(b) shows a result for the median and 32 regions.) If



(a) Quadtree (34 regions) (b) Kd-Tree (32 regions) (c) METIS (32 regions)

**Fig. 2.** Germany with three different partitions

the median of points in general position is used, the partitioning has always $2^l$ regions.

The median of the nodes can be computed in linear time with the *median of medians* algorithm [8]. Since the running time of the preprocessing is dominated by the shortest-path computations after the partitioning of the graph, we decided to use a standard sorting algorithm instead. (As a concrete example, the *kd-tree* partitioning with 64 regions for one of our test graphs with one million nodes was calculated in 175s, calculating the arc-flags took seven hours.)

### 3.4 METIS

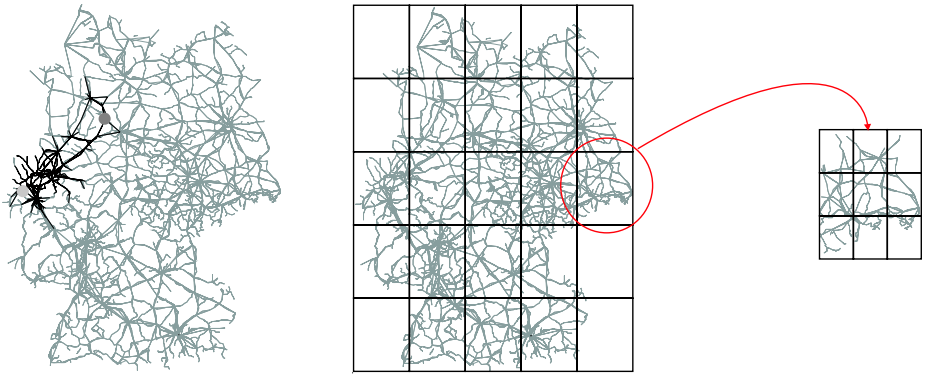
A fast method to partition a graph into $k$ almost equally-sized sets with a small cut-set is presented in [9]. An efficient implementation can be obtained free-of-charge from [6]. There are two advantages of this method for our application. The METIS partitioning does not need a layout of the graph and the preprocessing is faster because the number of arcs in the cut is noticeable smaller than in the other partitioning methods. Fig. 2(c) shows a partitioning of a graph generated by METIS.

## 4 Two-Level Arc-Flags

An analysis of the calculated arc-flags reveals that there might exist possibilities to compress the arc-flags. For 80% of the arcs either almost none or nearly all bits of their arc-flags are set. Table 1 shows an excerpt of the analysis we made. The column "= 1" shows the number of arcs, which are only responsible for shortest paths inside their own region (only one bit is set). Arcs with more than 95% bits set could be important roads. This justifies ideas for (lossy) compression

**Table 1.** Some statistics about the number of regions that are marked in arc-flags

| graph | #arcs | algorithm | # marked regions | | |
|---|---|---|---|---|---|
| | | | = 1 | < 10% | > 95% |
| road_network_1 | 920,000 | KdTree(32) | 351,255 | 443,600 | 312,021 |
| road_network_1 | 920,000 | KdTree(64) | 334,533 | 470,818 | 294,664 |
| road_network_1 | 920,000 | METIS(80) | 346,935 | 468,101 | 290,332 |
| road_network_4 | 2,534,000 | KdTree(32) | 960,779 | 1,171,877 | 854,670 |
| road_network_4 | 2,534,000 | KdTree(64) | 913,605 | 1,209,353 | 799,206 |



(a) Without two-level arc-flags a search visits almost all arcs in the target region.

(b) For each arc a bit-vector is stored for the coarse $5 \times 5$ grid and a bit-vector for a fine $3 \times 3$ grid *in the same coarse region as the arc.*

**Fig. 3.** Illustrations for two-level vectors

of the arc-flags, but it is important that the decompression algorithm is very fast—otherwise the speed-up of time will be lost.

Let us have a closer look at a search space to get an idea of how to compress the arc-flags. As illustrated in Figure 3(a) for a search from the dark grey node to the light grey node, the modified DIJKSTRA search reduces the search space next to the beginning of the search but once the target region has been reached, almost all nodes and arcs are visited. This is not very surprising if you consider that usually all arcs of a region have set the region-bit of their own region. We could handle this problem if we used a finer partition of the graph but this would lead to longer arc-flags (requiring more memory and a longer preprocessing). Take the following example, if we used a $15 \times 15$ grid instead of a $5 \times 5$ grid, each region would be split in 9 additional regions but the preprocessing data increases from 25 to 225 bits per arc. However, the additional information for the fine grid is mainly needed for arcs in the target region of the coarse grid. This leads to the idea that we could split each region of the coarse partition but store this

additional data (for the fine grid) only for the arcs inside the same coarse region. Therefore, each arc gets two bit-vectors: one for the coarse partition and one for the associated region of the fine partition.

The advantage of this method is that the preprocessed data is smaller than for a fine one-level partitioning, because the second bit-vector exists only for the target region (34 bits per arc instead of 225). It is clear that the $15 \times 15$ grid would lead to better results. However, the difference for the search spaces is small because we expect that entries in arc-flags of neighboring regions are similar for regions far away. Thus, we can see this two-level method as a compression of the first-level arc-flags. We summarize the bits for remote regions. If one bit is set for a fine region, the bit is set for the whole group.

Only a slight modification of the search algorithm is required. Until the target region is reached, everything will remain unaffected, unnecessary arcs will be ignored with the arc-flags of level one. If the algorithm has entered the target region, the second-level arc-flags provide further information on whether an arc can be ignored for the search of a shortest path to the target-node.

Experiments showed (Section 6) that this method leads to the best results concerning the reduction of the search space, but an increased preprocessing effort is needed. Note however, that it is not necessary in the preprocessing to compute the complete shortest-path trees for all boundary nodes of the fine partitioning. The computation can be stopped if all nodes in the same coarse region are finished.

## 5 Experimental Setup

The main goal of this section is to compare the different partitioning algorithms with regard to their resulting search space and speed-up of time during the accelerated Dijkstra search. We tested the algorithms on German road networks, which are directed and have a 2D layout and positive arc weights. Table 2 shows some characteristics of the graphs. The column "shortest path" is the average number of nodes on a shortest path in the graph. For the unmodified Dijkstra's algorithm, the average running time and number of nodes touched by the algorithm is given for 5000 runs.

All experiments are performed with an implementation of the algorithms in C++ using the GCC compiler 3.3. We used the graph data structure from

**Table 2.** Characteristics of tested road networks. The columns "shortest paths" provides the average number of nodes on a shortest path

| Graph | #nodes | #arcs | shortest path | Dijkstra's algorithm time [$s$] | #touched nodes |
|---|---|---|---|---|---|
| road_network_1 | 362,000 | 920,000 | 250 | 0.26 | 183,509 |
| road_network_2 | 474,000 | 1,169,000 | 440 | 0.27 | 240,421 |
| road_network_3 | 609,000 | 1,534,000 | 580 | 0.30 | 306,607 |
| road_network_4 | 1,046,000 | 2,534,000 | 490 | 0.78 | 522,850 |

LEDA 4.4 [10]. As we do not have real-world shortest-path queries we considered random queries for our experiments. For each graph, we generated a demand file with 5000 random shortest-path requests so that all algorithms use the same shortest-path demands. All runtime measurements were made on a single AMD Opteron Processor with 2.2 GHz and 8 GB RAM. Note that our algorithms are not optimized with respect to space consumption. However, even for the largest graphs considered in this study significant less than 8 GB RAM would suffice.

Our speed-up method reduces the complete graph for each search to a smaller sub-graph and this leads to a smaller search space. We sampled the average size of the search space by counting the number of visited nodes and measured the average CPU time per query. Dijkstra's algorithm is used as a reference algorithm to compare search space and CPU time. Fortunately, Dijkstra's algorithm with arc-flags only tests a bit of a bit-vector and does not lead to a significant overhead. In graphs we tested, there is a strong linear correlation between the search space and the CPU time. This justifies that in the analysis it is sufficient to consider the search space only.

Arc-flags can be combined with a bidirectional search. In principle, arc-flags can be used independently for the forward search, the backward search, or both of them. In our experiments the best results (with a fixed total number of bits per arc) achieved a forward and backward accelerated bidirectional search, which means that we applied the partition-based speed-up technique on both search directions (with half of the bits for each direction).

## 6 Computational Results

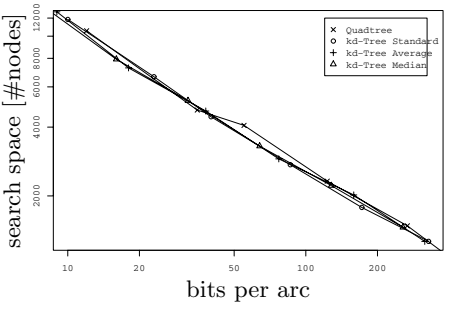
### 6.1 Quadtrees and Kd-Trees

We first compared the four geometric partitioning methods quadtrees and kd-trees for the center (standard), average, and median. Figure 4(a) shows the average search space for a road network for an increasing number of bits per arc. As the differences are indeed very small, we will use only kd-trees with median in the rest of this section as a representative for this partitioning class.

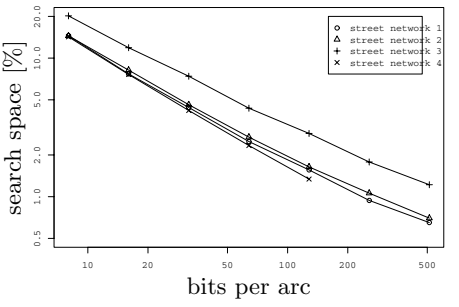
We now compare the average search space for different graphs. For an easy comparison we consider the search space *relative* to the average search space of Dijkstra's algorithm in this graph. Figure 4(b) provides the relative average search space for an increasing number of bits per arc. It is remarkable that for arc-flags in this range of size all curves follow a power law.

### 6.2 Two-Level Partitionings

The main reason for the introduction of the second-level partitions was that no arc is excluded from the shortest-path search inside the region of the target node $t$. Therefore, the second-level arc-flags reduce the shortest-path search mainly if the search already approaches the target. Figure 5 compares the search spaces of the one-level and two-level accelerated searches. Although only very few bits are added, the average search space is reduced to about half of its size.




(a) Partitioning with quadtree and three kd-tree partitions (standard, average, and median). The difference for the resulting search space is marginal.




(b) Partitioning with median kd-tree for road network 1-4. The search space is plotted relative to the search space of Dijkstra's algorithm.

**Fig. 4.** Average search space for different sizes of arc-flags. With an increasing number of bits per arc, the search space gets smaller

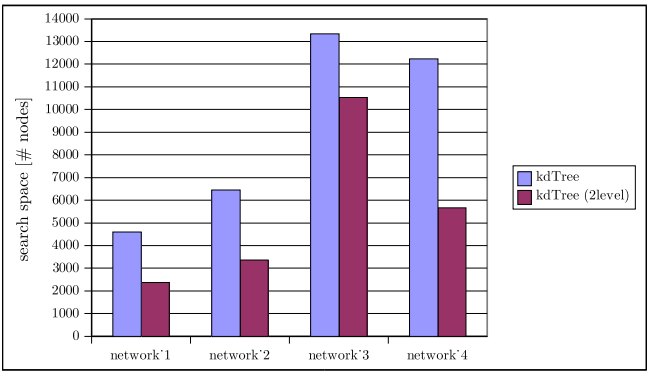



**Fig. 5.** Comparison of one-level (64 regions) and two-level (64 first-level regions, 8 second-level regions) arc-flags with kd-trees

Using a bidirectional search, the two-level strategy becomes less important, because the second-level arc-flags will not be used in most of the shortest-path searches: the second-level arc-flags are only used, if the search enters the region of the target. During a bidirectional search the probability is high that the two search horizons meet in a different region than the source or target region. Therefore, the second-level arc-flags are mainly used, if both nodes are lying in the same region. Figure 6 confirms this estimation. Only for large partitions in the first level is a speed-up recognizable. If more than 50 bits for the first level are used, the difference is very small. We conclude that the second-level strategy does not seem to be useful in a bidirectional search.

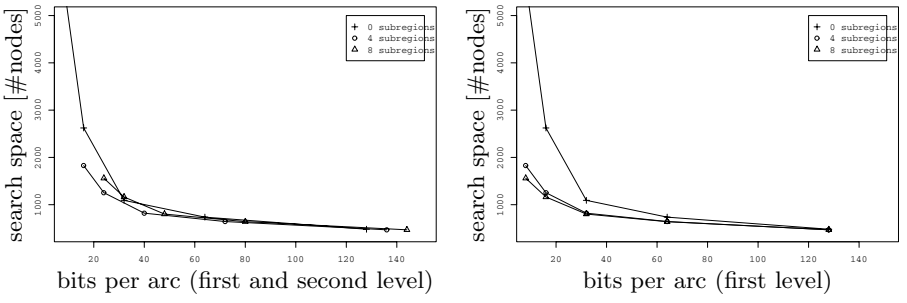



**Fig. 6.** Average search space for a bidirectional search using arc-flags by kd-trees. The two-level strategy becomes irrelevant for the bidirectional search. If more than 50 regions are used for the first-level, the two-level acceleration does not provide any noticeable improvement

### 6.3 Comparison of the Partitioning Methods

Finally, we want to compare the different algorithms directly. We have four orthogonal dimensions in our algorithm tool-box:

1. The base partitioning method: Grid, KdTree or METIS
2. The number of partitions
3. Usage of one-level partitions or two-level partitions
4. Unidirectional or bidirectional search

Since computing all possible combinations on all graphs takes way too much time, we selected the algorithms that are listed in Table 3. (We refrained from implementing Bi2Metis, because usually the two-level arc-flags in a bidirectional search hardly performed better than the one-level variant.) Furthermore, we fix the size of the preprocessed data to nearly the same number for all algorithms.

**Table 3.** Partitionings with nearly the same preprocessed data size of 80 bit

| Name of partitioning | forward | | backward | | bits per arc |
|---|---|---|---|---|---|
| | $1^{st}$ level | $2^{nd}$ level | $1^{st}$ level | $2^{nd}$ level | |
| Grid | $9 \times 9$ | - | - | - | 81 |
| KdTree | 64 | - | - | - | 64 |
| METIS | 80 | - | - | - | 80 |
| 2LevelGrid | $8 \times 8$ | $4 \times 4$ | - | - | 80 |
| 2LevelKd | 64 | 16 | - | - | 80 |
| 2LevelMETIS | 72 | 8 | - | - | 80 |
| BiGrid | $7 \times 7$ | - | $6 \times 6$ | - | 85 |
| BiKd | 32 | - | 32 | - | 64 |
| BiMETIS | 40 | - | 40 | - | 80 |
| Bi2LevelGrid | $6 \times 6$ | $2 \times 2$ | $6 \times 6$ | $2 \times 2$ | 80 |
| Bi2LevelKd | 32 | 8 | 32 | 8 | 80 |

(The same size can not be realized due to the restrictions by the construction of the partitioning algorithms.) Figure 7 compares the results of our partitioning methods on the four road networks.

For the unidirectional searches, the two-level strategies yield the best results (a factor of 2 better than for their corresponding one-level partitioning). For the bidirectional search, we can see some kind of saturation: the differences between the partitioning techniques are very small.

Figure 8 shows the search space for the four road networks. Note that in the case of a bidirectional search, a large number of bits per arc already shows some

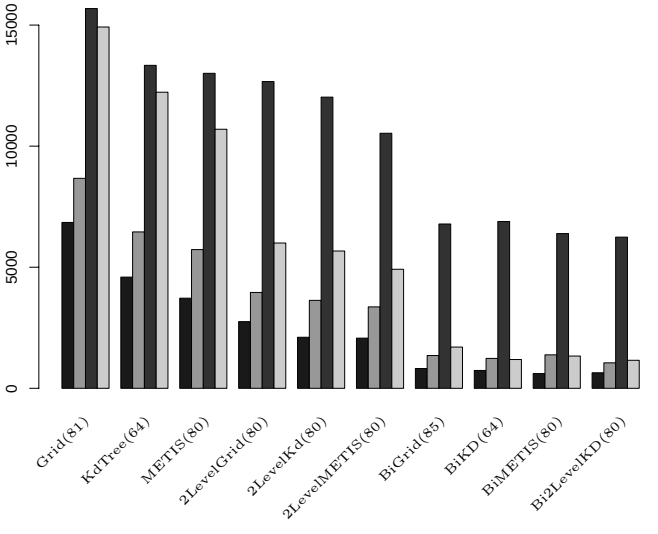



**Fig. 7.** Average search space for most of the implemented algorithms in road networks 1-4. The number of bits are noted in brackets

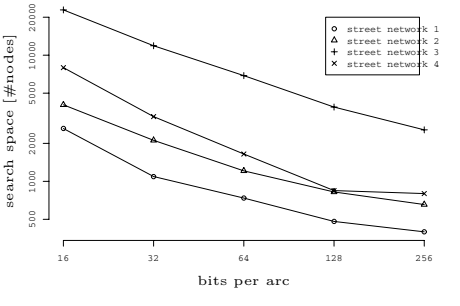



**Fig. 8.** Search space for road networks 1-4 with a bidirectional accelerated search using kd-tree partitions

effects of saturation as the curves are bent. In contrast, in Fig. 4(b) the curves follow a power-law.

## 7 Conclusion and Outlook

The best partition-based speed-up method we tested is a bidirectional search, accelerated in both directions with kd-tree or METIS partitions. We can measure speed-ups of more than 500. (In general, the speed-up increases with the size of the graph.) Even with the smallest preprocessed data (16 bit per arc), we get a speed-up of more than 50. The accelerated search on network_4 is 545 times faster than plain Dijkstra's algorithm using 128 bits per arc preprocessed data ($1.3ms$ per search). Of the tested partitioning methods, we can recommend the kd-tree used for forward and backward acceleration. The partitionings with kd-trees and METIS yield the highest speed-up factors, but kd-trees are easier to implement.

It would be particularly interesting to develop a specialized partitioning method that is optimized for the arc-flags approach. Our experiments showed that our intuition is right that regions should be equally sized and nodes should be grouped together if their graph-theoretic distance is small. However, we cannot prove that our intuition is theoretically the best partitioning method for arc-flags. Although many further techniques are known from graph clustering, all optimization criteria that we are aware of either result in a large running time or their use for the arc-flags approach cannot be motivated.

For an unidirectional search the two-level arc-flags lead to a considerable speed-up. The reduction of the search space outweighs by far the overhead to "uncompress" two-level arc-flags. It would, however, be interesting to evaluate whether this effect can be repeated with a third or fourth level of compression (especially for very large graphs like the complete road network of Europe).

There are further known speed-up techniques [11, 12, 13, 14]. Although the speed-up factors of these speed-up techniques are not competitive, experimental studies [15, 16] with similar techniques suggest that combinations outperform a single speed-up technique. A systematic evaluation of combinations with current approaches would therefore be of great value.

## References

1. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In Raubal, M., Sliwinski, A., Kuhn, W., eds.: Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung. Volume 22 of IfGI prints., Institut für Geoinformatik, Münster (2004) 219–230
2. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
3. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics **5** (2000)

4. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In Battista, G.D., Zwick, U., eds.: Proc. 11th European Symposium on Algorithms (ESA 2003). Volume 2832 of LNCS., Springer (2003) 776–787
5. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. (2005) Submitted to WEA'05.
6. Karypis, G.: METIS: Family of multilevel partitioning algorithms. `http://www-users.cs.umn.edu/~karypis/metis/` (1995)
7. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. Journal of the ACM (JACM) **24** (1977) 1–13
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge Massachusetts (2001)
9. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing archive **20** (1998) 359–392
10. Mehlhorn, K., Näher, S.: LEDA, A platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
11. Jung, S., Pramanik, S.: HiTi graph model of topographical road maps in navigation systems. In: Proc. 12th IEEE Int. Conf. Data Eng. (1996) 76–84
12. Holzer, M.: Hierarchical speed-up techniques for shortest-path algorithms. Technical report, Dept. of Informatics, University of Konstanz, Germany (2003) http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.
13. Goldberg, A.V., Harrelson, C.: Computing the shortest path: $a^*$ search meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research (2003) Accepted at SODA 2005.
14. Gutman, R.: Reach-based routing: A new approach to shortest path algortihms optimized for road networks. In Arge, L., Italiano, G.F., Sedgewick, R., eds.: Proc. Algorithm Engineering and Experiments (ALENEX'04), SIAM (2004) 100–111
15. Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In Ribeiro, C.C., Martins, S.L., eds.: Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004). Volume 3059 of LNCS., Springer (2004) 269–284
16. Wagner, D., Willhalm, T.: Drawing graphs to speed up shortest-path computations. In: Proc. 7th Workshop Algorithm Engineering and Experiments (ALENEX'05). LNCS, Springer (2005) To appear.

# Efficient Convergence to Pure Nash Equilibria in Weighted Network Congestion Games*

Panagiota N. Panagopoulou and Paul G. Spirakis

Computer Technology Institute, Riga Feraiou 61, 26221, Patras, Greece
Computer Engineering and Informatics Department, Patras University, Greece
panagopp@hermes.cti.gr; spirakis@cti.gr

**Abstract.** In large-scale or evolving networks, such as the Internet, there is no authority possible to enforce a centralized traffic management. In such situations, Game Theory and the concepts of Nash equilibria and Congestion Games [8] are a suitable framework for analyzing the equilibrium effects of selfish routes selection to network delays.
We focus here on *layered* networks where selfish users select paths to route their loads (represented by arbitrary integer *weights*). We assume that individual link delays are equal to the total load of the link. We focus on the algorithm suggested in [2], i.e. a potential-based method for finding *pure* Nash equilibria (PNE) in such networks. A superficial analysis of this algorithm gives an upper bound on its time which is polynomial in $n$ (the number of users) and the sum of their weights. This bound can be exponential in $n$ when some weights are superpolynomial. We provide strong experimental evidence that this algorithm actually converges to a PNE in strong *polynomial time* in $n$ (independent of the weights values). In addition we propose an initial allocation of users to paths that dramatically accelerates this algorithm, compared to an arbitrary initial allocation. A by-product of our research is the discovery of a weighted potential function when link delays are *exponential* to their loads. This asserts the existence of PNE for these delay functions and extends the result of [2].

## 1 Introduction

In large-scale or evolving networks, such as the Internet, there is no authority possible to employ a centralized traffic management. Besides the lack of central regulation, even cooperation of the users among themselves may be impossible due to the fact that the users may not even know each other. A natural assumption in the absence of central regulation and coordination is to assume

* This work was partially supported by the EU within the Future and Emerging Technologies Programme under contract IST200133135 (CRESCCO) and within the 6th Framework Programme under contract 001907 (DELIS).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 203–215, 2005.
© Springer-Verlag Berlin Heidelberg 2005

that network users behave selfishly and aim at optimizing their own individual welfare. Thus, it is of great importance to investigate the selfish behavior of users so as to understand the mechanisms in such non-cooperative network systems.

Since each user seeks to determine the shipping of its own traffic over the network, different users may have to optimize completely different and even conflicting measures of performance. A natural framework in which to study such multi-objective optimization problems is (non-cooperative) game theory. We can view network users as independent agents participating in a non-cooperative game and expect the routes chosen by users to form a Nash equilibrium in the sense of classical game theory: a Nash equilibrium is a state of the system such that no user can decrease his individual cost by unilaterally changing his strategy.

Users selfishly choose their private strategies, which in our environment correspond to paths from their sources to their destinations. When routing their traffics according to the strategies chosen, the users will experience an expected latency caused by the traffics of all users sharing edges (i.e the latency on the edges depends on their congestion). Each user tries to minimize his private cost, expressed in terms of his individual latency. If we allow as strategies for each user all probability distributions on the set of their source-destination paths, then a Nash equilibrium is guaranteed to exist. It is very interesting however to explore the existence of *pure* Nash equilibria (PNE) in such systems, i.e. situations in which each user is deterministically assigned on a path from which he has no incentive to unilaterally deviate.

Rosenthal [8] introduced a class of games, called *congestion games*, in which each player chooses a particular subset of resources out of a family of allowable subsets for him (his strategy set), constructed from a basic set of primary resources for all the players. The *delay* associated with each primary resource is a non-decreasing function of the number of players who choose it, and the total delay received by each player is the sum of the delays associated with the primary resources he chooses. Each game in this class possesses at least one Nash equilibrium in pure strategies. This result follows from the existence of a real-valued function (an *exact potential* [6]) over the set of pure strategy profiles with the property that the gain of a player unilaterally shifting to a new strategy is equal to the corresponding increment of the potential function.

In a *multicommodity network congestion game* the strategy set of each player is represented as a set of origin-destination paths in a network, the edges of which play the role of resources. If all origin-destination pairs of the users coincide we have a *single commodity network congestion game*. In a *weighted congestion game* we allow users to have different demands for service, and thus affect the resource delay functions in a different way, depending on their own weights. Hence weighted congestion games are not guaranteed to possess a PNE.

*Related Work.* As already mentioned, the class of (unweighted) congestion games is guaranteed to have at least one PNE. In [1] it is proved that a PNE for any (unweighted) single commodity network congestion game can be constructed in polynomial time, no matter what resource delay functions are considered (so long as they are non-decreasing functions with loads). On the other hand, it is

shown that even for an unweighted multicommodity network congestion game it is PLS-complete to find a PNE, though it certainly exists.

For the special case of single commodity network congestion games where the network consists of parallel edges from a unique origin to a unique destination and users have varying demands, it was shown in [3] that there is always a pure Nash equilibrium which can be constructed in polynomial time.

[5] deals with the problem of weighted parallel-edges congestion games with user-specific costs: each allowable strategy of a user consists of a single resource and each user has his own private cost function for each resource. It is shown that all such games involving only two users, or only two possible strategies for all the users, or equal delay functions, always possess a PNE. However, it is shown that even a 3-user, 3-strategies, weighted parallel-edges congestion game may not possess a PNE.

In [2] it is proved that even for a weighted single commodity network congestion game with resource delays being either linear or 2-wise linear functions of their loads, there may be no PNE. Nevertheless, it is proved that for the case of a weighted single commodity $\ell$-layered network congestion game (to be defined later) with resource delays identical to their loads, at least one PNE exists and can be computed in pseudo-polynomial time.

*Our Results.* We focus our interest on weighted $\ell$-layered network congestion games with resource delays equal to their loads. As already mentioned, any such game possesses a PNE, and the algorithm suggested in [2] requires at most a pseudo-polynomial number of steps to reach an equilibrium; this bound however has not yet been proven to be tight. The algorithm starts with any initial allocation of users on paths and iteratively allows each unsatisfied user to switch to any other path where he could reduce his cost. We experimentally show that the algorithm actually converges to a PNE in polynomial time for a variety of networks and distributions of users' weights. In addition, we propose an initial allocation of users onto paths that, as our experiments show, leads to a significant reduction of the total number of steps required by the algorithm, as compared to an arbitrary initial allocation.

Moreover, we present a **b**-potential function for any single commodity network congestion game with resource delays exponential to their loads, thus assuring the existence of a PNE in any such game (Theorem 2).

## 2 Definitions and Notation

*Games, Congestion Games and Weighted Congestion Games.* A *game* $\Gamma = \langle N, (\Pi_i)_{i\in N}, (u_i)_{i\in N}\rangle$ in strategic form is defined by a finite set of *players* $N = \{1, \ldots, n\}$, a finite set of *strategies* $\Pi_i$ for each player $i \in N$, and a *payoff function* $u_i : \Pi \to \mathbb{R}$ for each player, where $\Pi \equiv \times_{i\in N}\Pi_i$ is the set of *pure strategy profiles* or *configurations*. A game is *symmetric* if all players are indistinguishable, i.e. all $\Pi_i$'s are the same and all $u_i$'s, considered as a function of the choices of the other players, are identical symmetric functions of $n-1$ variables. A *pure*

*Nash equilibrium* (PNE) is a configuration $\pi = (\pi_1, \ldots, \pi_n)$ such that for each player $i$, $u_i(\pi) \geq u_i(\pi_1, \ldots, \pi'_i, \ldots, \pi_n)$ for any $\pi'_i \in \Pi_i$. A game may not possess a PNE in general. However, if we extend the game to include as strategies for each $i$ all possible probability distributions on $\Pi_i$ and if we extend the payoff functions $u_i$ to capture expectation, then an equilibrium is guaranteed to exist [7].

A *congestion model* $\langle N, E, (\Pi_i)_{i\in N}, (d_e)_{e\in E}\rangle$ is defined as follows. $N$ denotes the set of players $\{1, \ldots, n\}$. $E$ denotes a finite set of resources. For $i \in N$ let $\Pi_i$ be the set of strategies of player $i$, where each $\varpi_i \in \Pi_i$ is a nonempty subset of resources. For $e \in E$ let $d_e : \{1, \ldots, n\} \to \mathbb{R}$ denote the delay function, where $d_e(k)$ denotes the cost (e.g. delay) to each user of resource $e$, if there are exactly $k$ players using $e$. The *congestion game* associated with this congestion model is the game in strategic form $\langle N, (\Pi_i)_{i\in N}, (u_i)_{i\in N}\rangle$, where the payoff functions $u_i$ are defined as follows: Let $\Pi \equiv \times_{i\in N}\Pi_i$. For all $\varpi = (\varpi_1, \ldots, \varpi_n) \in \Pi$ and for every $e \in E$ let $\sigma_e(\varpi)$ be the number of users of resource $e$ according to the configuration $\varpi$: $\sigma_e(\varpi) = |\{i \in N : e \in \varpi_i\}|$. Define $u_i : \Pi \to \mathbb{R}$ by $u_i(\varpi) = -\sum_{e\in\varpi_i} d_e(\sigma_e(\varpi))$.

In a *weighted congestion model* we allow the users to have different demands, and thus affect the resource delay functions in a different way, depending on their own weights. A weighted congestion model $\langle N, (w_i)_{i\in N}, E, (\Pi_i)_{i\in N}, (d_e)_{e\in E}\rangle$ is defined as follows. $N$, $E$ and $\Pi_i$ are defined as above, while $w_i$ denotes the demand of player $i$ and for each resource $e \in E$, $d_e(\cdot)$ is the delay per user that requests its service, as a function of the total usage of this resource by all the users. The *weighted congestion game* associated with this congestion model is the game in strategic form $\langle (w_i)_{i\in N}, (\Pi_i)_{i\in N}, (u_i)_{i\in N}\rangle$, where the payoff functions $u_i$ are defined as follows. For any configuration $\varpi \in \Pi$ and for all $e \in E$, let $\Lambda_e(\varpi) = \{i \in N : e \in \varpi_i\}$ be the set of players using resource $e$ according to $\varpi$. The cost $\lambda^i(\varpi)$ of user $i$ for adopting strategy $\varpi_i \in \Pi_i$ in a given configuration $\varpi$ is equal to the cumulative delay $\lambda_{\varpi_i}(\varpi)$ on the resources that belong to $\varpi_i$: $\lambda^i(\varpi) = \lambda_{\varpi_i}(\varpi) = \sum_{e\in\varpi_i} d_e(\theta_e(\varpi))$ where, for all $e \in E$, $\theta_e(\varpi) \equiv \sum_{i\in\Lambda_e(\varpi)} w_i$ is the load on resource $e$ with respect to the configuration $\varpi$. The payoff function for player $i$ is then $u_i(\varpi) = -\lambda^i(\varpi)$. A configuration $\varpi \in \Pi$ is a PNE if and only if, for all $i \in N$, $\lambda_{\varpi_i}(\varpi) \leq \lambda_{\pi_i}(\varpi_{-i}, \pi_i) \quad \forall \pi_i \in \Pi_i$, where $(\varpi_{-i}, \pi_i)$ is the same configuration as $\varpi$ except for user $i$ that has now been assigned to path $\pi_i$. Since the payoff functions $u_i$ can be implicitly computed by the resource delay functions $d_e$, in the following we will denote a weighted congestion game by $\langle (w_i)_{i\in N}, (\Pi_i)_{i\in N}, (d_e)_{e\in E}\rangle$.

In a *network congestion game* the families of subsets $\Pi_i$ are presented implicitly as paths in a network. We are given a directed network $G = (V, E)$ with the edges playing the role of resources, a pair of nodes $(s_i, t_i) \in V \times V$ for each player $i$ and the delay function $d_e$ for each $e \in E$. The strategy set of player $i$ is the set of all paths from $s_i$ to $t_i$. If all origin-destination pairs $(s_i, t_i)$ of the players coincide with a unique pair $(s, t)$ we have a *single commodity network congestion game* and then all users share the same strategy set, hence the game is symmetric. If users have different demands, we refer to *weighted network con-*

*gestion games* in the natural way. In the case of a *weighted single commodity network congestion game* however the game is not necessarily symmetric, since the users have different demands and thus their cost functions will also differ.

*Potential Functions.* Fix some vector $\mathbf{b} \in \mathbb{R}^n_{>0}$. A function $F : \times_{i\in N} \Pi_i \to \mathbb{R}$ is a $\mathbf{b}$-*potential* for the weighted congestion game $\Gamma = \langle (w_i)_{i\in N}, (\Pi_i)_{i\in N}, (d_e)_{e\in E}\rangle$ if $\forall \varpi \in \times_{i\in N}\Pi_i, \forall i \in N, \forall \pi_i \in \Pi_i,\ \lambda^i(\varpi) - \lambda^i(\varpi_{-i}, \pi_i) = b_i \cdot (F(\varpi) - F(\varpi_{-i}, \pi_i))$. $F$ is an *exact potential* for $\Gamma$ if $b_i = 1$ for all $i \in N$. It is well known [6] that if there exists a $\mathbf{b}$-potential for a game $\Gamma$, then $\Gamma$ possesses a PNE.

*Layered Networks.* Let $\ell \geq 1$ be an integer. A directed network $(V, E)$ with a distinguished source-destination pair $(s, t)$, $s, t \in V$, is $\ell$-layered if every directed $s-t$ path has length exactly $\ell$ and each node lies on a directed $s-t$ path. The nodes of an $\ell$-layered network can be partitioned into $\ell + 1$ layers, $V_0$, $V_1$, ..., $V_\ell$, according to their distance from the source node $s$. Since each node lies on directed $s-t$ path, $V_0 = \{s\}$ and $V_\ell = \{t\}$. Similarly we can partition the edges $E$ of an $\ell$-layered network in $\ell$ subsets $E_1, \ldots, E_\ell$ where for all $j \in \{1, \ldots, \ell\}$, $E_j = \{e = (u, v) \in E : u \in V_{j-1} \text{ and } v \in V_j\}$.

## 3 The Problem

We focus our interest on the existence and tractability of pure Nash equilibria in weighted $\ell$-layered network congestion games with resource delays identical to their loads. Consider the $\ell$-layered network $G = (V, E)$ with a unique source-destination pair $(s, t)$ and the weighted single commodity network congestion game $\langle (w_i)_{i\in N}, \mathcal{P}, (d_e)_{e\in E}\rangle$ associated with $G$, such that $\mathcal{P}$ is the set of all directed $s-t$ paths of $G$ and $d_e(x) = x$ for all $e \in E$. Let $\varpi = (\varpi_1, \ldots, \varpi_n)$ be an arbitrary configuration and recall that $\theta_e(\varpi)$ denotes the load of resource $e \in E$ under configuration $\varpi$. Since resource delays are equal to their loads, for all $i \in N$ it holds that $\lambda^i(\varpi) = \lambda_{\varpi_i}(\varpi) = \sum_{e\in\varpi_i} \theta_e(\varpi) = \sum_{e\in\varpi_i} \sum_{j\in N | e\in\varpi_j} w_j$.

A user $i \in N$ is *satisfied* in the configuration $\varpi \in \mathcal{P}^n$ if he has no incentive to unilaterally deviate from $\varpi$, i.e. if for all $s-t$ paths $\pi \in \mathcal{P}$, $\lambda_{\varpi_i}(\varpi) \leq \lambda_\pi(\varpi_{-i}, \pi)$. The last inequality can be written equivalently

$$\lambda_{\varpi_i}(\varpi_{-i}) + \ell w_i \leq \lambda_\pi(\varpi_{-i}) + \ell w_i \quad \Longleftrightarrow \quad \lambda_{\varpi_i}(\varpi_{-i}) \leq \lambda_\pi(\varpi_{-i}) \ ,$$

hence user $i$ is satisfied if and only if he is assigned to the shortest $s-t$ path calculated with respect to the configuration $\varpi_{-i}$ of all the users except for $i$. The configuration $\varpi$ is a PNE if and only if all users are satisfied in $\varpi$. In [2] it was shown that any such weighted $\ell$-layered network congestion game possesses a PNE that can be computed in pseudo-polynomial time:

**Theorem 1 ([2]).** *For any weighted $\ell$-layered network congestion game with resource delays equal to their loads, at least one PNE exists and can be computed in pseudo-polynomial time.*

*Proof (sketch).* The **b**-potential function establishing the result is

$$\Phi(\varpi) = \sum_{e \in E} (\theta_e(\varpi))^2$$

where, $\forall i \in N, b_i = \frac{1}{2w_i}$. □

In Sect. 4 we present the pseudo-polynomial algorithm *Nashify*() for the computation of a PNE for a weighted $\ell$-layered network congestion game, while in Sect. 6 we experimentally show that such a PNE can actually be computed in polynomial time, as our following conjecture asserts:

*Conjecture 1.* Algorithm *Nashify*() converges to a PNE in polynomial time.

## 4 The Algorithm

The algorithm presented below converts any given non-equilibrium configuration into a PNE by performing a sequence of greedy selfish steps. A greedy selfish step is a user's change of his current pure strategy (i.e. path) to his best pure strategy with respect to the current configuration of all other users.

Algorithm *Nashify*$(G, (w_i)_{i \in N})$
*Input:* $\ell$-layered network $G$ and a set $N$ of users, each user $i$ having weight $w_i$
*Output:* configuration $\varpi$ which is a PNE

1. begin
2. select an initial configuration $\varpi = (\varpi_1, \ldots, \varpi_n)$
3. while $\exists$ user $i$ such that $\lambda_{\varpi_i}(\varpi_{-i}) > \lambda_s(\varpi_{-i})$ where $s = \text{Shortest\_Path}(\varpi_{-i})$
4. $\varpi_i := \text{Shortest\_Path}(\varpi_{-i})$
5. return $\varpi$
6. end

The above algorithm starts with an initial allocation of each user $i \in N$ on an $s-t$ path $\varpi_i$ of the $\ell$-layered network $G$. The algorithm iteratively examines whether there exists any user that is unsatisfied. If there is such a user, say $i$, then user $i$ performs a greedy selfish step, i.e. he switches to the shortest $s-t$ path according to the configuration $\varpi_{-i}$. The existence of the potential function $\Phi$ assures that the algorithm will terminate after a finite number of steps at a configuration from which no user will have an incentive to deviate, i.e. at a PNE.

*Complexity Issues.* Let $W = \sum_{i \in N} w_i$. Note that in any configuration $\varpi \in \mathcal{P}^n$ and for all $j \in \{1, \ldots, \ell\}$ it holds that $\sum_{e \in E_j} \theta_e(\varpi) = W$. It follows that

$$\Phi(\varpi) = \sum_{e \in E} (\theta_e(\varpi))^2 = \sum_{j=1}^{\ell} \sum_{e \in E_j} (\theta_e(\varpi))^2 \leq \sum_{j=1}^{\ell} \left( \sum_{e \in E_j} \theta_e(\varpi) \right)^2 = \ell W^2 \ .$$

Without loss of generality assume that the users have integer weights. At each iteration of the algorithm *Nashify*() an unsatisfied user performs a greedy selfish

step, so his cost must decrease by at least 1 and thus the potential function $\Phi$ decreases by at least $2 \min_i w_i \geq 2$. Hence the algorithm requires at most $\frac{1}{2}\ell W^2$ steps so as to converge to a PNE.

**Proposition 1.** *Suppose that* $\frac{(\max_i w_i)^2}{\min_i w_i} = O(n^k)$ *for some constant* $k$. *Then algorithm Nashify() will converge to a PNE in polynomial time.*

*Proof.* Observe that $\Phi(\varpi) \leq \ell W^2 \leq \ell(n \max_i w_i)^2 = \ell n^2 \min_i w_i \cdot O(n^k)$, which implies that the algorithm will reach a PNE in $O(\ell n^{k+2})$ steps. □

## 5 The Case of Exponential Delay Functions

In this section we deal with the existence of pure Nash equilibria in weighted single commodity network congestion games with resource delays being exponential to their loads. Let $G = (V, E)$ be any single commodity network (not necessarily layered) and denote by $\mathcal{P}$ the set of all $s - t$ paths in it from the unique source $s$ to the unique destination $t$. Consider the weighted network congestion game $\Gamma = \langle (w_i)_{i \in N}, \mathcal{P}, (d_e)_{e \in E} \rangle$ associated with $G$, such that for any configuration $\varpi \in \mathcal{P}^n$ and for all $e \in E$, $d_e(\theta_e(\varpi)) = \exp(\theta_e(\varpi))$. We next show that $F(\varpi) = \sum_{e \in E} \exp(\theta_e(\varpi))$ is a **b**-potential for such a game and some positive $n$-vector **b**, assuring the existence of a PNE.

**Theorem 2.** *For any weighted single commodity network congestion game with resource delays exponential to their loads, at least one PNE exists.*

*Proof.* Let $\varpi \in \mathcal{P}^n$ be an arbitrary configuration. Let $i$ be a user of demand $w_i$ and fix some path $\pi_i \in \mathcal{P}$. Denote $\varpi' \equiv (\varpi_{-i}, \pi_i)$. Observe that

$$\lambda^i(\varpi) - \lambda^i(\varpi') = \sum_{e \in \varpi_i \setminus \pi_i} \exp(\theta_e(\varpi_{-i}) + w_i) - \sum_{e \in \pi_i \setminus \varpi_i} \exp(\theta_e(\varpi_{-i}) + w_i)$$

$$= \exp(w_i) \cdot \left( \sum_{e \in \varpi_i \setminus \pi_i} \exp(\theta_e(\varpi_{-i})) - \sum_{e \in \pi_i \setminus \varpi_i} \exp(\theta_e(\varpi_{-i})) \right) .$$

Note that, for all $e \notin \{\{\varpi_i \setminus \pi_i\} \cup \{\pi_i \setminus \varpi_i\}\}$, it holds that $\theta_e(\varpi) = \theta_e(\varpi')$. Now

$$F(\varpi) - F(\varpi') = \sum_{e \in \varpi_i \setminus \pi_i} \exp\left(\theta_e(\varpi_{-i}) + w_i\right) - \exp(\theta_e(\varpi_{-i}))$$

$$+ \sum_{e \in \pi_i \setminus \varpi_i} \exp(\theta_e(\varpi_{-i})) - \exp(\theta_e(\varpi_{-i}) + w_i)$$

$$= \frac{\exp(w_i) - 1}{\exp(w_i)} \left(\lambda^i(\varpi) - \lambda^i(\varpi')\right) .$$

Thus, $F$ is a **b**-potential for our game, where $\forall i \in N, b_i = \frac{\exp(w_i)}{\exp(w_i) - 1}$, assuring the existence of at least one PNE. □

## 6 Experimental Evaluation

*Implementation Details.* We implemented algorithm *Nashify*() in C++ programming language using several advanced data types of LEDA [4]. In our implementation, we considered two initial allocations of users on paths: (1) Random allocation: each user assigns its traffic uniformly at random on an $s-t$ path and (2) Shortest Path allocation: users are sorted according to their weights, and the maximum weighted user among those that have not been assigned a path yet selects a path of shortest length, with respect to the loads on the edges caused by the users of larger weights.

Note that, in our implementation, the order in which users are checked for satisfaction (line 3 of algorithm *Nashify*()) is the worst possible, i.e. we sort users according to their weights and, at each iteration, we choose the minimum weighted user among the unsatisfied ones to perform a greedy selfish step. By doing so, we force the potential function to decrease as less as possible and thus we maximize the number of iterations, so as to be able to better analyze the worst-case behavior of the algorithm.

### 6.1 Experimental Setup

*Networks.* Figure 1 shows the $\ell$-layered networks considered in our experimental evaluation of algorithm *Nashify*(). Network 1 is the simplest possible layered network and Network 2 is a generalization of it. Observe that the number of possible $s-t$ paths of Network 1 is 3, while the number of possible $s-t$ paths for Network 2 is $3^5$. Network 3 is an arbitrary dense layered network and Network 4 is the $5\times 5$ grid. Network 5 is a 4-layered network with the property that layers $1, 2, 3$ form a tree rooted at $s$ and layer 4 comprises all the edges connecting the leaves of this tree with $t$.

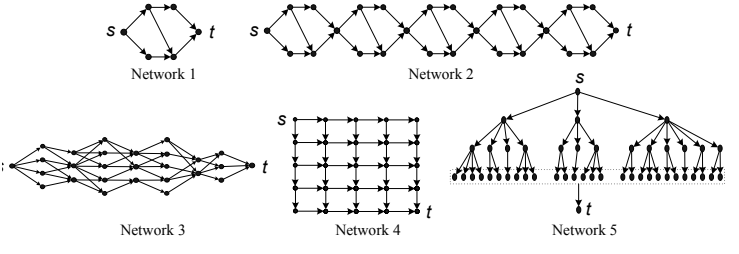



**Fig. 1.** The $s-t$ layered networks considered

*Distribution of weights.* For each network, we simulated the algorithm *Nashify*() for $n = 10, 11, \ldots, 100$ users. Obviously, if users' weights are polynomial in $n$ then the algorithm will definitely terminate after a polynomial number of steps. Based on this fact, as well as on Proposition 1, we focused on instances

where some users have exponential weights. More specifically, we considered the following four distributions of weights: (a) 10% of users have weight $10^{n/10}$ and 90% of users have weight 1, (b) 50% of users have weight $10^{n/10}$ and 50% of users have weight 1, (c) 90% of users have weight $10^{n/10}$ and 10% of users have weight 1, and (d) users have uniformly at random selected weights in the interval $[1, 10^{n/10}]$. Distributions (a)–(c), albeit simple, represent the distribution of service requirements in several communication networks, where a fraction of users has excessive demand that outweighs the demand of the other users.

### 6.2 Results and Conclusions

Figures 2–6 show, for each network and each one of the distributions of weights (a)–(d), the number of steps performed by algorithm *Nashify*() over the number of users (#steps/$n$) as a function of the sum of weights of all users $W$. For each instance we considered both random and shortest path initial allocation.

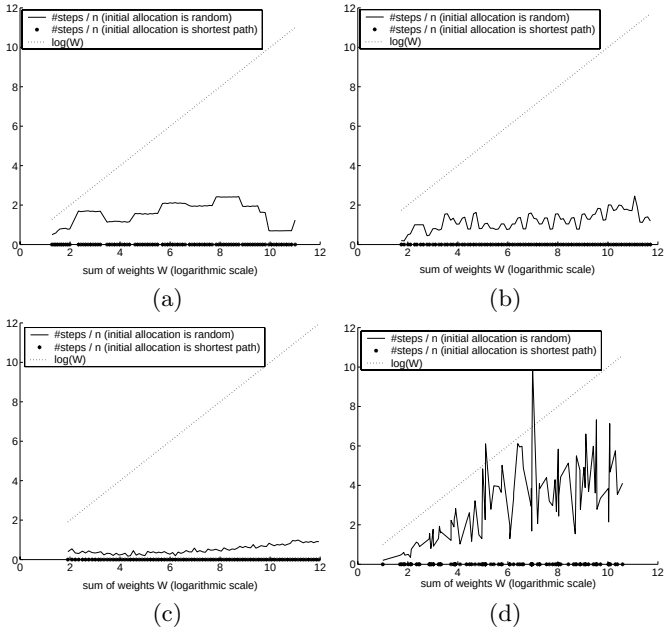



**Fig. 2.** Experimental results for Network 1

Observe that the shortest path initial allocation significantly outperforms any random initial allocation, no matter what networks or distributions of weights are considered. In particular, the shortest path initial allocation appears to be a PNE for sparse (Networks 1 and 2), grid (Network 4) and tree-like (Network 5) networks, while for the dense network (Network 3) the number of steps over the number of users seems to be bounded by a small constant.

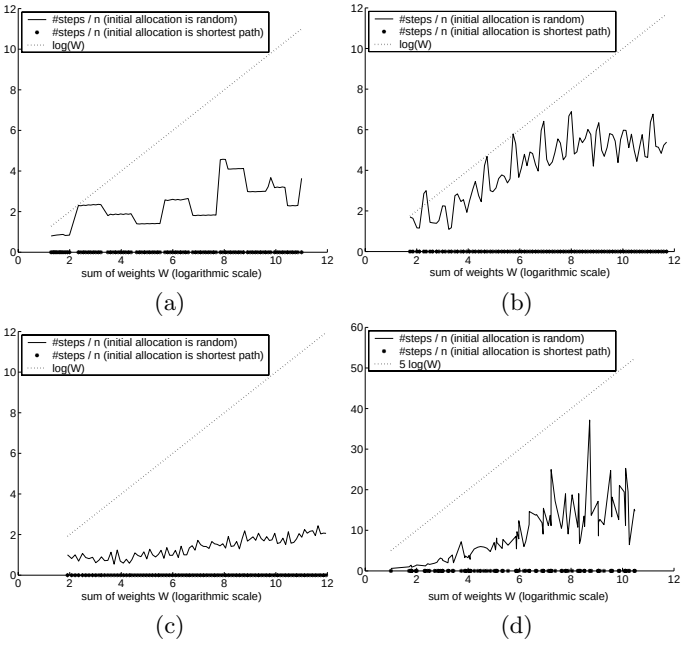



**Fig. 3.** Experimental results for Network 2

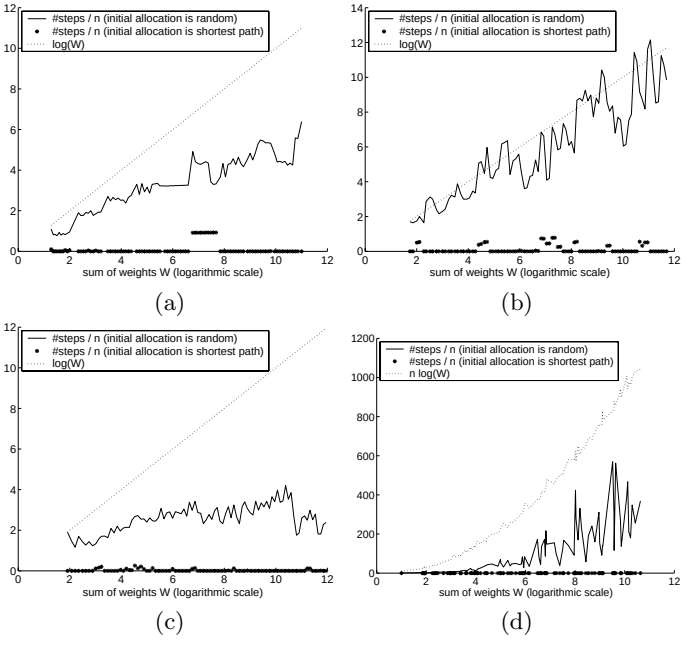



**Fig. 4.** Experimental results for Network 3

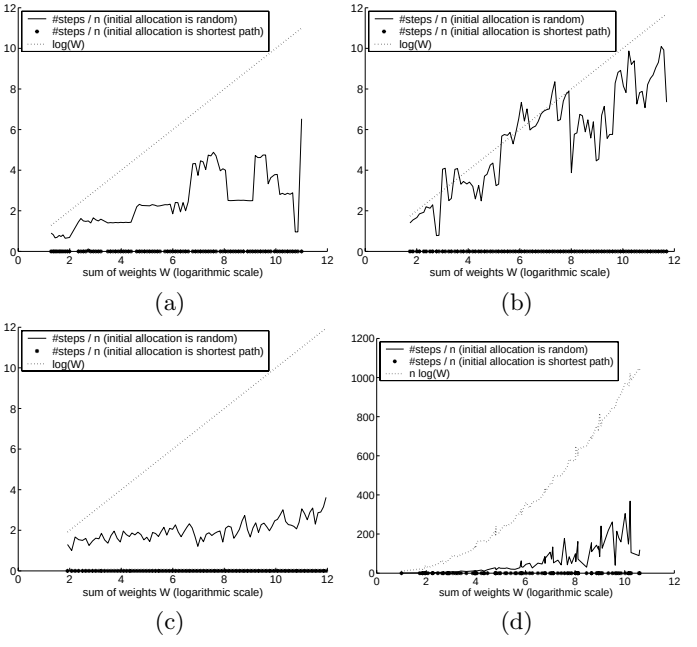



**Fig. 5.** Experimental results for Network 4

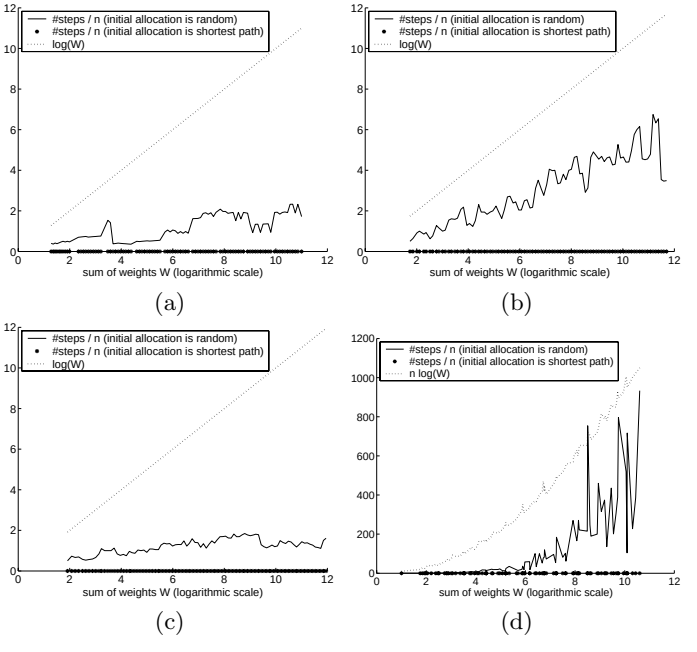



**Fig. 6.** Experimental results for Network 5

On the other hand, the behavior of the algorithm when starting with an arbitrary allocation is sensibly worse. First note that, in this case, the fluctuations observed at the plots are due to the randomization of the initial allocation. On the average however we can make safe conclusions as regards the way #steps/$n$ increases as a function of $W$. For the distributions of weights (a)–(c) it is clear that the number of steps over the number of users is asymptotically upper bounded by the logarithm of the sum of all weights, implying that #steps $= O(n \cdot \log(W))$. Unfortunately, the same does not seem to hold for randomly selected weights (distribution (d)). In this case however, as Figs. 2–6(d) show, $n \log(W)$ seems to be a good asymptotic upper bound for #steps/$n$, suggesting that #steps $= O(n^2 \cdot \log(W))$.

Note that, for all networks, the maximum number of steps over the number of users occurs for the random distribution of weights. Also observe that, for the same value of the sum of weights $W$, the number of steps is dramatically smaller when there are only 2 distinct weights (distributions (a)–(c)). Hence we conjecture that the complexity of the algorithm does actually depend not only on the sum of weights, but also on the number of distinct weights of the input.

Also note that the results shown in Figs. 2 and 3 imply that, when starting with an arbitrary allocation, the number of steps increases as a linear function of the size of the network. Since the number of $s - t$ paths in Network 2 is exponential in comparison to that of Network 1, we would expect a significant increment in the number of steps performed by the algorithm. Figures 2 and 3 however show that this is not the case. Instead, the number of steps required for Network 2 are at most 5 times the number of steps required for Network 1.

Summarizing our results, we conclude that (i) a shortest path initial allocation is a few greedy selfish steps far from a PNE, amplifying Conjecture 1, while (ii) an arbitrary initial allocation does not assure a similarly fast convergence to a PNE, however Conjecture 1 seems to be valid for this case as well, (iii) the size of the network does not affect significantly the time complexity of the algorithm, and (iv) the worst-case input for an arbitrary initial allocation occurs when all users' weights are distinct and some of them are exponential.

## References

1. Fabrikant, A., Papadimitriou, C., Talwar, K.: The Complexity of Pure Nash Equilibria. Proc. of the 36th ACM Symp. on Theory of Computing (STOC 04), 2004.
2. Fotakis, D., Kontogiannis, S., Spirakis, P.: Selfish Unsplittable Flows. 31st International Colloquium on Automata, Languages and Programming (ICALP'04), pp. 593–605, 2004.
3. Fotakis, D., Kontogiannis, S., Koutsoupias, E., Mavronicolas, M., Spirakis, P.: The Structure and Complexity of Nash Equilibria for a Selfish Routing Game. Proc. of the 29th International Colloquium on Automata, Languages and Programming (ICALP 02), Springer-Verlag, 2002, pp. 123–134.
4. Mehlhorn, K., Näher, S.: LEDA – A Platform for Combinatorial and Geometric Computing. Cambridge University Press, 1999.

5. Milchtaich, I.: Congestion Games with Player-Specific Payoff Functions. Games and Economic Behavior 13 (1996), 111–124.
6. Monderer, D., Shapley, L.: Potential Games. Games and Economic Behavior, 14:124–143, 1996.
7. Nash, J. F.: Equilibrium Points in $N$-person Games. Proc. of National Academy of Sciences, Vol. 36, pp. 48–49, 1950.
8. Rosenthal, R. W.: A Class of Games Poseessing Pure-Strategy Nash Equilibria. International Journal of Game Theory 2, pp. 65–67, 1973.

# New Upper Bound Heuristics for Treewidth*

Emgad H. Bachoore and Hans L. Bodlaender

Institute of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

**Abstract.** In this paper, we introduce and evaluate some heuristics to find an upper bound on the treewidth of a given graph. Each of the heuristics selects the vertices of the graph one by one, building an elimination list. The heuristics differ in the criteria used for selecting vertices. These criteria depend on the fill-in of a vertex and the related new notion of the fill-in-excluding-one-neighbor. In several cases, the new heuristics improve the bounds obtained by existing heuristics.

## 1 Introduction

For several applications involving graphs, it is of great interest to have good algorithms that compute or approximate the treewidth of a graph and its corresponding tree decomposition. The interest in these notions, and some other related notions such as branchwidth, branch decomposition, pathwidth, path decomposition, and minimum fill-in arose because of their theoretical significance in (algorithmic) graph theory, and because a tree decomposition of small width of a graph enables us to solve many graph problems in linear or polynomial time. Among these problems are many well-known combinatorial optimization problems such as: graph coloring, maximum independent set, and the Hamiltonian cycle problem. Nowadays, there are several 'real world' applications that use the notion of tree decomposition or branch decomposition to find solutions for the problems at hand. These come from many different fields, such as expert systems [15], probabilistic networks [4], frequency assignment problems [12, 13], telecommunication networks design, VLSI-design, natural language processing [10], and the traveling salesman problem [9].

The problem of computing the treewidth of a graph is *NP-hard* [2]. Therefore, for computing the treewidth of a graph, we have to use an exact but slow method like branch and bound, algorithms that work only for specific classes of graphs, or resort to algorithms that only approximate the treewidth. In the past years, several heuristics for treewidth have been designed. We can divide those heuristics into two categories: those that find upper bounds for the treewidth, and those that find lower bounds for the treewidth. This paper concentrates on upper bound heuristics. An overview of results on computing the treewidth can be found in [3].

Some of the heuristics for finding an upper bound for the treewidth are based on algorithms that test whether a given graph is triangulated. These are Maximum Cardinality Search, Lexicographic Breadth First search, Minimum Degree and Minimum

* This work has been supported by the Netherlands Organization for Scientific Research NWO (project TACO: 'Treewidth And Combinatorial Optimization').

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 216–227, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Fill-in algorithms. Other heuristics are based on other ideas, e.g. the Minimum Separating Vertex Set algorithm (cf. [1, 7, 11]).

In this paper, we present a number of heuristic methods to find upper bounds for the treewidth of the graph. Each of our heuristics is based on constructing a triangulation of the graph from an elimination ordering of the vertices. These elimination orderings are constructed by repeatedly selecting a vertex and then adding the so-called fill-in edges between its neighbors. The various heuristics differ in the criteria of the selection of the vertices. These criteria, basically, depend on two concepts: the number of edges that must be added between the neighbor vertices of a vertex to form a clique, and the number of edges that must be added between all neighbor vertices of a vertex except one to form a clique between them.

We have implemented the algorithms proposed in this paper, and tested them on a set of 32 graphs, taken from instances of probabilistic networks, frequency assignment and vertex coloring. We compared the results of these algorithms with those of other heuristic methods. We observed that in many cases our algorithms perform well.

## 2 Definitions and Preliminary Results

In this section, we give the definitions of the most frequently used concepts and notions in this paper. Let $G = (V, E)$ be an undirected graph with vertex set $V$ and edge set $E$. A graph $H$ is a ***minor*** of graph $G$, if $H$ can be obtained from $G$ by zero or more vertex deletions, edge deletions, and edge contractions. ***Edge contraction*** is the operation that replaces two adjacent vertices $v$ and $w$ by a single vertex that is connected to all neighbors of $v$ and $w$.

We denote the set of neighbors of vertex $v$ by $N(v) = \{w \in V | \{v, w\} \in E\}$, and the set of neighbors of $v$ plus $v$ itself by $N[v] = N(v) \cup \{v\}$. In the same manner we define $N^0[v] = \{v\}$, $N^{i+1}[v] = N[N^i[v]]$, $N^{i+1}(v) = N^{i+1}[v] \setminus N^i[v]$. We can extend the above definition to a set of vertices instead of one vertex. Suppose that $S$ is a set of vertices, then $N^0[S] = S,\ N^{i+1}[S] = N[N^i[S]],\ N^{i+1}(S) = N^{i+1}[S] \setminus N^i[S],\ N[S] = \bigcup_{v \in S} N[v],\ N(S) = N[S] \setminus S,\ i \in \mathcal{N}$. Let $degree(v) = |N(v)|$ be the degree of vertex $v$. Given a subset $A \subseteq V$ of the vertices, we define the ***subgraph induced by A*** to be $G_A = (A, E_A)$, where $E_A = \{\{x, y\} \in E | x \in A \ and \ y \in A\}$. A subset $A \subseteq V$ of $r$ vertices is an ***r-almost-clique*** if there is a $v \in A$ such that $A - \{v\}$ forms a clique. A vertex $v$ in $G$ is called ***simplicial***, if its set of neighbors $N(v)$ forms a clique in $G$. A vertex $v$ in $G$ is called ***almost simplicial***, if its neighbors except one form a clique in $G$, i.e., if $v$ has a neighbor $w$ such that $N(v) - \{w\}$ is a clique. A graph $G$ is called ***triangulated*** (or: chordal) if every cycle of length four of more possesses a chord. A ***chord*** is an edge between two nonconsecutive vertices of the cycle. A graph $G = (V, E)$ is a subgraph of graph $H = (W, F)$ if $V \subseteq W$ and $E \subseteq F$. A graph $H = (V, F)$ is a ***triangulation of graph*** $G = (V, E)$, if $G$ is a subgraph of $H$ and $H$ is a triangulated graph.

A ***linear ordering*** of a graph $G = (V, E)$ is a bijection $f : V \rightarrow \{1, 2, \cdots, |V|\}$. A linear ordering of the vertices of a graph $G$, $\sigma = [v_1, \cdots, v_n]$ is called a ***perfect elimination order (p.e.o.)*** of $G$, if for every $1 \leq i \leq n$, $v_i$ is a simplicial vertex in $G[v_1, \cdots, v_n]$, i.e., the higher numbered neighbors of $v_i$ form a clique. It has been

shown in [8] that a graph $G$ is triangulated, if and only if $G$ has a $p.e.o.$ ***Eliminating a vertex*** $v$ from a graph $G = (V, E)$ is the operation that first adds an edge between every pair of non-adjacent neighbors of $v$, and then removes $v$ and its incident edges.

A ***tree decomposition*** of the graph $G = (V, E)$ is a pair $(X, T)$ in which $T = (I, F)$ a tree, and $X = \{X_i | i \in I\}$ a collection of subsets of $V$, one for each node of $T$, such that $\bigcup_{i \in I} X_i = V$, for all $(u, v) \in E$, there exists an $i \in I$ with $u, v \in X_i$, and for all $i, j, k \in I$: if $j$ is on path from $i$ to $k$ in $T$, then $X_i \cap X_k \subseteq X_j$. The ***width*** of the tree decomposition $((I, F), \{X_i | i \in I\})$ is $max_{i \in I} |X_i - 1|$. The ***treewidth*** of a graph $G$ is the minimum width over all tree decompositions of $G$.

**Lemma 1.** *(See Bodlaender [5].)*

1. *For every triangulated graph $G = (V, E)$, there exists a tree decomposition $(X = \{X_i | i \in I\}, T = (I, F))$ of $G$, such that every set $X_i$ forms a clique in $G$, and for every maximal clique $W \subseteq V$, there exists an $i \in I$ with $W = X_i$.*
2. *Let $(X = \{X_i | i \in I\}, T = (I, F))$ be a tree decomposition of $G$ of width at most $k$. The graph $H = (V, E \cup E')$, with $E' = \{\{v, w\} | \exists i \in I : v, w \in X_i\}$, obtained by making every set $X_i$ a clique, is triangulated, and has maximum clique size at most $k + 1$.*
3. *Let $(X = \{X_i | i \in I\},\ T = (I, F))$ be a tree decomposition of $G$, and let $W \subseteq V$ form a clique in $G$. Then there exist an $i \in I$ with $W \subseteq X_i$.*

**Lemma 2.** *(See Shoikhet and Geiger [17].)*
*For triangulated graphs, tree decompositions exist where the nodes are exactly the maximal cliques of the graph. Such tree decompositions are called clique trees. The width of a triangulated graph $T$ is $max_{k \in K(T)}(|K| - 1)$, where $K(T)$ is the set of maximal cliques of $T$.*

## 3 Upper Bound Heuristics for Treewidth

In this section we present some heuristic methods to find upper bounds for the treewidth of a given graph, and the corresponding tree decompositions. Basically, these methods depend on two concepts: The first one is the number of edges that must be added between the neighbors of a vertex $x$ to make it simplicial, i.e., the neighborhood of that vertex turn into a clique. We call this the *fill-in* of $x$.

$$\textit{fill-in(x)} = |\{\{v, w\} | v, w \in N(x), \{v, w\} \notin E\}|$$

The second concept is very similar to the first one, but here we find the minimum number of edges which when added between pairs of neighbors of a vertex $x$, turn $x$ into an almost simplicial vertex, i.e., by adding these edges to the graph, the neighborhood of that vertex will turn into an almost clique. We call this parameter the fill-in of x excluding one neighbor *fill-in-excl-one(x)*

$$\textit{fill-in-excl-one(x)} = min_{z \in N(X)} |\{\{v, w\} | v, w \in N(x) - \{z\}, \{v, w\} \notin E\}|$$

A graph with $|V|$ vertices has $|V|!$ (permutations of $|V|$) linear ordering. For each linear ordering $\sigma$ of $G$, we can build a triangulation $H_\sigma$ of $G$, such that $\sigma$ is the *p.e.o.* of $H_\sigma$,

in the following way. For $i = 1, \cdots, |V|$, in that order, we add an edge between every pair of non-adjacent neighbors of $v_i$ that are after $v_i$ in the ordering, $v_i$ is the $i^{th}$ vertex in $\sigma$. One can observe that $\sigma$ is a *p.e.o.* of the resulting graph $H_\sigma$. As $H_\sigma$ is triangulated, its treewidth is one smaller than its maximum clique size, which equals the maximum number of neighbors of $v$ over all vertices $v$ that are after $v$ in the linear ordering $\sigma$. One can construct from $H_\sigma$ a tree decomposition of $H_\sigma$ and of $G$ with width exactly this maximum clique size minus one. It is also known that there is at least one linear ordering of $G$ where we obtain the exact treewidth of $G$ in this way [5]. This suggests the following general scheme for heuristics for treewidth.

**set** $G' \leftarrow G$; $i \leftarrow 1$; $\sigma \leftarrow ()$;
**while** $G'$ is not the empty graph
    **select** according to some criteria a vertex $v$ from $G'$;
    **eliminate** $v$; /* remove $v$ and turn its neighbors into a clique */
    **add** $v$ to position $i$ in the ordering $\sigma$;
    **set** $i \leftarrow i + 1$;
{Now $\sigma$ is a linear ordering of $V$.}
**construct** triangulation $H_\sigma$ of $G$ and the corresponding tree decomposition.

Instead of constructing $H_\sigma$ after $\sigma$ is constructed, we also can construct $H_\sigma$ and the corresponding tree decomposition while $\sigma$ is constructed. The width of the tree decomposition thus obtained is the maximum over all vertices $v$ of the number of neighbors of $v$ in $G'$. We call a graph $G'$ encountered during the algorithm a temporary graph. A linear ordering of $G$, used in this way, is called often an *elimination scheme*. Several heuristics are of this type. Most known are the Minimum Fill-in heuristic, and the Minimum Degree heuristic. In these, we repeatedly select the vertex $v$ with minimum fill-in in $G'$, or minimum degree in $G'$ respectively. These two heuristics appear to be successful heuristics for treewidth; they often give good bounds and are fast to compute. The success of these heuristics encouraged us to develop other heuristics based on similar principles. Our heuristics are inspired by results on preprocessing graphs for treewidth. In [4], reduction rules are given that are safe for treewidth. Here, such rules rewrite a graph $G$ to a smaller graph $G'$, and possibly update a variable *low* that gives a lower bound on the treewidth of the original graph. In [4], the notion of ***safe rule*** was introduced. The safe rule rewrites a graph to a smaller one, and maintains a lower bound variable $low$, such that the maximum of $low$ and the treewidth of the graph at hand stays invariant, i.e., rule $R$ is safe, if for all graphs $G,\ G'$, and all integers $low,\ low'$, we have

$$(G, low) \rightarrow_R (G', low') \Rightarrow max(treewidth(G), low) = max(treewidth(G'), low').$$

Thus, the treewidth of the original graph is known when we know the treewidth of the reduced graph and low. Amongst others, the following two rules were shown to be safe for the treewidth in [4].

**The Simplicial Reduction Rule (SRR):**
***let*** $v$ be a simplicial vertex of $degree(v) \geq 0$.
***remove*** $v$.
***set*** $low$ to $max(low, degree(v))$.

**The Almost Simplicial Reduction Rule (ASRR):**
***let*** $v$ be an almost simplicial vertex of $degree(v) \geq 2$.
***if*** $low \geq degree(v)$ then ***eliminate*** $v$.

The safeness of the simplicial reduction rule tells us that if a vertex $v$ has $fill - in$ zero, then selecting that vertex as the next one in the elimination ordering will not cause the remaining graph to have a treewidth that is larger than necessary for this elimination ordering. It can be seen as a motivation for the Minimum Fill-in Heuristic, where we select vertices with minimum fill-in. Similarly, the almost simplicial reduction rule can be seen as motivation to look at the $fill - in - excl - one$. If a vertex has $fill - in - excl - one$ of zero, then selecting that vertex as the next vertex in the elimination ordering will in many cases not cause the treewidth caused by the formed elimination ordering to be larger than necessary, unless the degree of the almost simplicial vertex is more than the treewidth of the original graph. With a small twist to the terminology, we say that eliminating $v$ is safe (in a graph $G$), if the choice of $v$ in the heuristic scheme presented above can lead to a tree decomposition whose width equals the treewidth of $G$. Motivated by these observations, we designed new heuristics for treewidth that are given below.

### 3.1 Enhanced Minimum Fill-in (EMF):

The motivation for the Enhanced Minimum Fill-in algorithm is based on the safeness of eliminating simplicial vertices and almost simplicial vertices of degree at most the treewidth. Note that when we have vertices $x$ and $y$ with *fill-in-excl-one(y)* $= 0$, $degree(y) = low$ (for some lower bound $low$ on the treewidth of the input graph), *fill-in(x)* $> 1$, and *fill-in(y)* $>$ *fill-in(x)*, then y appears to be the best choice for elimination (as this is safe); the Minimum Fill-in heuristic, however, selects $x$.

For faster implementation of the algorithms, we observe that in many cases we do not have to recompute the values of *fill-in* and *fill-in-excl-one* of every vertex in the temporary graph after we eliminate a vertex from it.

**Lemma 3.** *Let $v$ be a simplicial vertex in graph $G$, $G' = G[V - \{v\}]$ be the graph obtained by eliminating $v$, and fill-in$_G(v)$ be the fill-in of vertex $v$ in graph $G$. For all $w \notin N^1[v]$, we have fill-in$_G(w)$= fill-in$_{G'}(w)$, fill-in-excl-one$_G(w)$= fill-in-excl-one$_{G'}(w)$.*

Therefore, when we eliminate a simplicial vertex $v$ from a graph, and we want to find the next vertex in the graph with minimum fill-in or minimum fill-in-excl-one, we need only to recompute fill-in and fill-in-excl-one for neighbors of $v$, $(N^1(v))$. For instance, if we eliminate vertex 1 from the graph shown in Figure 1(a) and we want to find the next vertex with minimum fill-in or minimum fill-in-excl-one in the graph, then we need only to recompute the fill-in and fill-in-excl-one for vertices 2, 3 and 4.

Similarly, Lemma 4 shows that if we eliminate a non simplicial vertex $v$ from a graph, and we want to find the next vertex in the graph with minimum fill-in or minimum fill-in-excl-one, then only the fill-in and fill-in-excl-one of the vertices in $N^2(v)$ can change. For instance, if we eliminate vertex 1 from the graph in Figure 1 (b) and

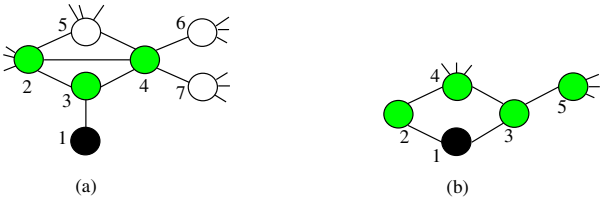



**Fig. 1**

we want to find the next vertex with minimum fill-in or minimum fill-in-excl-one, then we need to recompute the fill -in and fill-in- excl-one for vertices 2, 3, 4 and 5 only.

**Lemma 4.** *Let $v$ be a vertex in graph $G$ that is not simplicial, $G' = G[V - \{v\}]$ be the graph obtained by eliminating $v$, and fill-in$_G(v)$ be the fill-in of $v$ in $G$. For all $w \notin N^2[v]$, we have fill-in$_G(w)$= fill-in$_{G'}(w)$, fill-in-excl-one$_G(w)$= fill-in-excl-one$_{G'}(w)$.*

**Observation.** Many of the heuristics have slightly different implementations that can give different results on the same graph. For instance, consider the Minimum Fill-in heuristic. If there is more than one vertex that has minimum fill-in, the method does not specify which of these has to be selected and placed in the elimination ordering. For instance, there could be some arbitrary numbering of the vertices, and the specific implementation could choose the lowest or the highest numbered vertex with minimum fill-in. It seems better to use criteria that guide towards a better upper bound for the treewidth for such a selection. Using other criteria apart from minimum fill-in can be seen to give better bounds for several inputs. Still, in each case, there are graphs for which we do not find the optimal treewidth with such heuristics; given the NP-hardness of the problem, we also cannot expect to do so.

We would like to remark here that if we want to fully describe an upper bound algorithm, we must specify details like the method of representing the graph, specifically, because of matters like when vertices have the same fill-in, in which manner such a tie is broken. If we do not give such specifications, there is possibility to get different results from the same algorithm. Moreover, it becomes more difficult to compare the results of different methods.

In the proposed Enhanced Minimum Fill-in algorithm, we first select vertices whose elimination is safe, i.e., we select vertices that are simplicial (have fill-in 0), or almost simplicial (have fill-in-excl-one 0) and their degrees are at most the lower bound ($low$) on the treewidth of the graph. Only if no safe vertex is available, we select the vertex of minimum fill-in. Furthermore, we incorporated Lemma 3 and Lemma 4 in the algorithm.

### 3.2 Minimum Fill-in Excluding One Neighbor (MFEO1)

Using the ideas behind the EMF heuristic and additional techniques, we developed a more advanced heuristic. The idea is as follows: first, we test whether the graph contains

simplicial vertices, or almost simplicial vertices with degree at most the lower bound for the treewidth of the graph. If we find such vertices, we process them as in EMF. If the graph does not include such simplicial or almost simplicial vertices anymore, then we do the following test:

**let** $H(W, F)$ be the temporary graph of a given graph $G(V, E)$,
  $p$ be a vertex with *minimum fill-in* in $H$,
  *min-fill-in* $\leftarrow$ *fill-in(p)*,
  *local-min-fill-in* $\leftarrow$ *MaxInt*,
  *local-min-fill-in-excl-one* $\leftarrow$ *MaxInt*;
**for all** $q \in W and q \neq p$
  **if**((*fill-in-excl-one[q]* $<$ *min-fill-in*) $and$ $(degree[q] = low)$)
  **then**
    **if**((*fill-in-excl-one[q]* $<$ *local-min-fill-in-excl-one*) or
      ((*fill-in-excl-one[q]* $=$ *local-min-fill-in-excl-one*) and
      (*fill-in[q]* $<$ *local-min-fill-in*)))
    **then**
      $p \leftarrow q$;
      *local-min-fill-in* $\leftarrow$ *fill-in[q]*;
      *local-min-fill-in-excl-one* $\leftarrow$ *fill-in-excl-one[q]*;

After this test, if there is a vertex $q$ amongst $W$ that full fills these conditions, then that vertex becomes the next eliminating vertex, otherwise, vertex $p$ will be eliminated. However, if the graph contains more than one vertex $q$ with such properties then the vertex with minimum fill-in-excl-one amongst these is eliminated first. But, if still there is more than one vertex that satisfies the last condition, then the vertex with minimum fill-in amongst these should be selected first.

### 3.3 The Minimum Fill-in Excluding One Neighbor Vertex (MFEO2)

The MFEO2 heuristic is a modification of MFEO1 heuristic, where ties are broken using the fill-in and fill-in-excl-one in the other order. If more than one vertex $q$ satisfies the two conditions, namely, *fill-in-excl-one(q)* $<$ *fill-in(p)* and $degree(q) < low$, then the vertex with minimum fill-in amongst these is eliminated first. But, if still there is more than one vertex that satisfies the last condition, then the vertex with minimum fill-in-excl-one amongst these should be processed first.

### 3.4 The Ratio Heuristic, Version 1 (Ratio1)

In the two versions of the Ratio heuristic, we use different rules for when vertices of small fill-in-excl-one can be selected for elimination before vertices of minimum fill-in. Again, we first select simplicial vertices, or almost simplicial vertices whose degree is at least the lower bound for the treewidth. If there are no such vertices in the temporary graph, we proceed now as follows: In the Ratio heuristic, version 1, a vertex $v$ can be selected when its fill-in-excl-one is smaller than minimum fill-in, its degree is at most the lower bound for the treewidth, and it satisfies the following condition. Let $H(W, F)$

be a temporary graph of graph $G(V, E)$. Select a vertex $p$ of minimum fill-in. Compute $r_1(w) =$ *fill-in(w)/fill-in(p)*, and $r_2(w) = degree(w)/degree(p)$. We now require that $r_1(w) < r_2(w)$ for $w \neq p$ to be a candidate for selection at this point. If we have more than one such candidate, we select from these a vertex with minimum difference between $r_1$ and $r_2$, $(r_1 - r_2)$.

The motivation for the Ratio heuristic, version 1, is that we want to select vertices for which elimination creates a large clique while only few fill-in edges are added. Let us illustrate the method with the following examples: Let $p$ be a vertex of minimum fill-in and $w$ be a vertex whose fill-in-excl-one is less than minimum fill-in and its degree is less than the value of the lower bound for the treewidth.

### 3.5 The Ratio Heuristic, Version 2 (Ratio 2)

The second variant of the Ratio heuristic is similar to the first one, with the following difference. For each vertex $w \in W$, we set $r(w) =$ *fill-in(w)*$/degree(w)$, $(degree(w) > 1$, otherwise $w$ is simplicial). When there are no simplicial vertices and no almost simplicial vertices with degree at most the treewidth lower bound, we select the vertex $w$ whose ratio $r(w)$ is smallest.

## 4 Experimental Results

The algorithms described in the previous section and some other algorithms described in [7, 11] have been implemented using Microsoft Visual C++ 6.0 on a Window 2000 PC with a Pentium III 800 MHz processor. The algorithms were tested on two sets of graphs. The first one includes 18 graphs from real-life probabilistic networks and frequency assignment problems [11]. The second set includes 14 graphs from a DIMACS coloring benchmark [7]. The selected graphs have different number of vertices and edges. Also, the differences between the known upper bounds and lower bounds for the treewidth of many of those graphs are noticeable. Thus, it is possible to obtain different results for the upper bound of the same graph by using different heuristics.

In order to be able to achieve good conclusions from this analysis, we analyze our algorithms in different ways. First, we compare the results of the implementations of different methods that have been introduced in this paper. The second part of the evaluations compares the heuristics introduced in this paper with known heuristics. The tables use the following terminology. The columns with the upper bounds on the treewidth are labeled as ub. The column for processing times in seconds are labeled as t. The processing times are rounded of to the nearest integer. The values in column low are not necessarily the best known lower bounds on the treewidth, but by using these values with our heuristics we obtained the best upper bounds.

### 4.1 Comparison Between the Heuristics Introduced in This Paper

In Section 3, we have introduced five methods for finding upper bounds for the treewidth of the graph; in addition we implemented the Minimum Fill-in heuristic (MF) and the

Minimum Degree Fill-in (MDFI). The results of implementing these algorithms on different graphs are given in the following tables. Table 1 and Table 2 (columns 6-10) show a comparison between different methods illustrated in this paper from the point view of their best upper bound values and the processing time used to find these upper bounds. Table 1 consists of instances of probabilistic networks and frequency assignment problems. Table 2 gives graphs from the DIMACS coloring benchmark. The results for the D-LB heuristic in Table 2 were obtained from [7]. D-LB is shown in this table because [7] gives experimentel results for this set of graphs and not for those used in Table 3.
We notice clearly from the results in these tables that in general the upper bounds obtained by MFEO1 and RATIO2 are better than those obtained by the other methods. The main differences between the results obtained by MFEO1 and those obtained by RATIO2 are as follows:

- The upper bounds obtained by using the MFEO1 heuristic on graphs of probabilistic networks and frequency assignment instances in Table 1 are better than or equal to those produced by any other heuristic in that table. However, this is not the case for some of the instances from the DIMACS coloring benchmark, see Table 2. We notice when we consider Table 1 that the upper bounds for some graphs are better when using RATIO2 than those obtained when using MFEO1.
- The upper bounds obtained by using the MFEO1 are more stable, namely, always better than or equal to that produced by any other heuristic, except for RATIO2. RATIO2 does not have such a "stable behavior".

**Table 1**

| Graphname | Size | | low | Heuristic | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | EMF | | MFEO1 | | MFEO2 | | RATIO1 | | RATIO2 | |
| | $\|V\|$ | $\|E\|$ | | ub | t | ub | t | ub | t | ub | t | ub | t |
| alarm | 37 | 65 | 2 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| barley | 48 | 126 | 3 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 |
| boblo | 221 | 326 | 2 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 8 | 3 | 0 |
| celar06_pp | 100 | 350 | 11 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 |
| celar07_pp | 200 | 817 | 16 | 16 | 1 | 16 | 1 | 16 | 1 | 16 | 1 | 16 | 1 |
| celar09_pp | 340 | 1130 | 7 | 16 | 4 | 16 | 4 | 16 | 3 | 16 | 49 | 16 | 4 |
| graph05_pp | 100 | 416 | 11 | 26 | 0 | 25 | 1 | 25 | 1 | 25 | 0 | 26 | 1 |
| midew | 35 | 80 | 2 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| munin1 | 189 | 366 | 3 | 11 | 0 | 11 | 1 | 11 | 1 | 11 | 5 | 11 | 1 |
| oesoca_hugin | 67 | 208 | 9 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 |
| oow_bas | 27 | 54 | 2 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| oow_solo | 40 | 87 | 4 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 |
| oow_trad | 33 | 72 | 4 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 |
| pigs | 441 | 806 | 3 | 10 | 6 | 10 | 6 | 10 | 6 | 10 | 123 | 10 | 6 |
| ship-ship | 50 | 114 | 4 | 8 | 0 | 8 | 0 | 9 | 0 | 8 | 0 | 9 | 0 |
| vsd-hugin | 38 | 62 | 2 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| water | 32 | 123 | 9 | 10 | 0 | 9 | 0 | 10 | 0 | 9 | 0 | 10 | 0 |
| wilson-hugin | 21 | 27 | 2 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 |

**Table 2**

| Graphname | Size | | low | Heuristic | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | EMF | | MFEO1 | | MFEO2 | | RATIO1 | | RATIO2 | | DL_B | |
| | $\|V\|$ | $\|E\|$ | | ub | t | ub | t | ub | t | ub | t | ub | t | ub | t |
| anna | 138 | 986 | 11 | 12 | 0 | 12 | 0 | 12 | 0 | 12 | 0 | 12 | 0 | 12 | 1 |
| david | 87 | 812 | 11 | 13 | 0 | 13 | 0 | 13 | 0 | 13 | 0 | 13 | 0 | 13 | 0 |
| dsjc125_1 | 125 | 736 | 17 | 65 | 6 | 64 | 6 | 64 | 6 | 64 | 6 | 66 | 6 | 67 | 3 |
| dsjc125_5 | 125 | 3891 | 55 | 111 | 35 | 110 | 36 | 110 | 37 | 110 | 36 | 109 | 38 | 110 | 4 |
| dsjc250_1 | 250 | 3218 | 3 | 177 | 451 | 177 | 451 | 177 | 451 | 177 | 453 | 177 | 441 | 176 | 33 |
| games120 | 120 | 1276 | 10 | 39 | 2 | 39 | 2 | 42 | 1 | 41 | 2 | 38 | 2 | 41 | 2 |
| LE450_5A | 450 | 5714 | 3 | 315 | 7630 | 310 | 13694 | 315 | 13437 | 315 | 15308 | 304 | 13301 | 323 | 274 |
| mulsol_i_4 | 175 | 3946 | 32 | 32 | 23 | 32 | 27 | 32 | 26 | 32 | 26 | 32 | 28 | 32 | 14 |
| myciel4 | 23 | 71 | 10 | 11 | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 11 | 0 |
| myciel5 | 47 | 236 | 8 | 21 | 0 | 20 | 1 | 20 | 0 | 20 | 0 | 20 | 0 | 20 | 0 |
| myciel6 | 95 | 755 | 20 | 35 | 2 | 35 | 1 | 35 | 2 | 35 | 1 | 35 | 2 | 35 | 2 |
| myciel7 | 191 | 2360 | 31 | 66 | 31 | 66 | 31 | 66 | 32 | 66 | 32 | 66 | 28 | 70 | 29 |
| queen5_5 | 25 | 320 | 12 | 18 | 0 | 18 | 0 | 18 | 0 | 18 | 0 | 19 | 0 | 18 | 0 |
| school1 | 385 | 19095 | 80 | 225 | 5791 | 225 | 5742 | 225 | 5738 | 225 | 6390 | 209 | 3877 | 242 | 274 |

### 4.2 Comparison Between Heuristics Introduced in This Paper with Known Heuristics

Tables 2, 3 and 4 make a second type of comparison. Here, we compare the heuristics introduced in this paper with a number of existing heuristics from the scientific literature. The data for the existing heuristics were taken from [7, 11]. Table 2 gives the upper bounds found by using D-LB [7] and those found by using the heuristics introduced in this paper, with the processing time used by each heuristic. Table 3 compares between the MFEO1 heuristic and the heuristics that have been described in[11]. The first five are based on building elimination order lists, and respectively use two variants of Lexicographic Breadth First Search (LEX-P and LEX-M), the Maximum Cardinality Search (MCS), the Minimum Fill-in heuristic (MF) and the Minimum Degree heuristic (MD: the vertex of minimum degree in the temporary graph is chosen). The last one is the Minimum Separating Vertex Sets heuristic (MSVS) from Koster [14], where a trivial tree decomposition is stepwise refined with help of minimum vertex separators.

We conclude from the results of Tables 2, 3 and 4 that usually the best upper bounds were achieved by the MFEO1 heuristic. Table 4 shows how often each of the heuristics MFEO1, LEX-P, LEX-M, MF, MCS, and D-LB gives the best result of all seven on the 18 graphs from probabilistic network and frequency assignment instances, and 14 graphs from DIMACS vertex-coloring instances. We can see that out 18 graphs of the first set of graphs in Table 3, the upper bound of 2 graphs became better by using MFEO1 than that produced by the heuristics introduced in [11] together, the upper bounds of 14 graphs remain equal to the best upper bound of them, and no graph from this set of graphs its upper bound became worse by MFEO1 than that produced by those heuristics. As well, by applying the MFEO1 on 14 graphs of the second set of graphs (DIMACS vertex-coloring instances), the upper bounds of 6 graphs became better than that produced by D-LB (introduced in [7]), upper bounds of 7 graphs remain equal

**Table 3**

| Graphname | Size | | Heuristic Name | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | MFEO1 | | LEX-P | | LEX-M | | MF | | MDFI | | MCS | | MSVS | |
| | $\|V\|$ | $\|E\|$ | ub | t | ub | t | ub | t | ub | t | ub | t | ub | t | ub | t |
| alarm | 37 | 65 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| barley | 48 | 126 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 | 7 | 0 |
| boblo | 221 | 326 | 3 | 0 | 4 | 1 | 4 | 10 | 3 | 13 | 3 | 0 | 3 | 1 | 3 | 0 |
| celar06_pp | 100 | 350 | 11 | 0 | 11 | 0 | 11 | 2 | 11 | 1 | 11 | 0 | 11 | 0 | 11 | 0 |
| celar07_pp | 200 | 817 | 16 | 1 | 19 | 2 | 18 | 16 | 16 | 10 | 18 | 0 | 18 | 2 | 18 | 3 |
| celar09_pp | 340 | 1130 | 16 | 4 | 19 | 0 | 18 | 0 | 16 | 98 | 18 | 0 | 19 | 0 | 18 | 0 |
| graph05_pp | 100 | 416 | 25 | 1 | 29 | 3 | 27 | 11 | 26 | 1 | 28 | 0 | 29 | 3 | 26 | 5 |
| midew | 35 | 80 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 |
| munin1 | 189 | 366 | 11 | 1 | 15 | 2 | 13 | 20 | 11 | 9 | 11 | 0 | 20 | 3 | 11 | 2 |
| oesoca_hugin | 67 | 208 | 11 | 0 | 12 | 0 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 | 11 | 0 |
| oow_bas | 27 | 54 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 5 | 0 | 4 | 0 |
| oow_solo | 40 | 87 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 |
| oow_trad | 33 | 72 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 | 6 | 0 |
| pigs | 441 | 806 | 10 | 6 | 19 | 14 | 18 | 161 | 10 | 190 | 10 | 0 | 15 | 8 | 15 | 9 |
| ship-ship | 50 | 114 | 8 | 0 | 9 | 0 | 9 | 0 | 8 | 0 | 8 | 0 | 9 | 9 | 9 | 0 |
| vsd-hugin | 38 | 62 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 5 | 0 | 4 | 0 |
| water | 32 | 123 | 9 | 0 | 10 | 0 | 10 | 0 | 9 | 0 | 11 | 0 | 10 | 0 | 10 | 0 |
| wilson-hugin | 21 | 27 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 | 3 | 0 |

**Table 4**

| S | Heuristic name | Number of graphs where the upper bound produced by the heuristic is the only best one | | is equal the best one | | is worse than the best one | | Sum | |
|---|---|---|---|---|---|---|---|---|---|
| | | set 1 | set 2 | set 1 | set 2 | set 1 | set 2 | set 1 | set 2 |
| 1 | MFEO1 | 2 | 6 | 16 | 7 | 0 | 1 | 18 | 14 |
| 2 | LEX-P | 0 | | 6 | | 12 | | 18 | |
| 3 | LEX-M | 0 | | 6 | | 12 | | 18 | |
| 4 | MF | 0 | | 3 | | 15 | | 18 | |
| 5 | MD | 0 | | 11 | | 7 | | 18 | |
| 6 | MCS | 0 | | 5 | | 13 | | 18 | |
| 7 | DLB | 0 | 1 | | 7 | 6 | | | 14 |

to the best upper bound, and only for one graph, the upper bound obtained by D-LB method is better than that obtained by MFEO1. Table 4 shows a comparison between MFEO1 and each of these seven heuristics. It gives number of graphs when MFEO1 gives a better, equal, or worse upper bounds than that produced by every heuristics introduced in [7, 11]. The processing time of the MFEO1 heuristic is relatively close to the processing time of other heuristics in spite of the fact that this algorithm uses $O(n^4)$ time in the worst case, while some of other heuristics are of $O(n^2)$ time complexity. Although there are only two cases in Table 1 where the MFEO1 heuristic gives a bound that is better than each of other heuristics (namely, for graph05_pp and water), we can see that it gives in many cases the best known value.

In several cases, this is the exact treewidth of the graph. Our recent (unpublished) work on branch and bound algorithms for treewidth has shown that. In many cases, the heuristic gives considerable improvements compared with individual other heuristics. For some of the graphs we do not know the exact treewidth and cannot determine yet how much the upper bounds differ from the exact values.

## References


1. E. Amir. *Efficient approximation for triangulation of minimum treewidth*. Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence, UAI 2001, pages 7-15, Seattle, Washington, USA, 2001.
2. S. Arnborg, D. G. Corneil, and A. Proskurowski. *Complexity of finding embeddings in a k-tree*. SIAM Journal on Algebraic and Discrete Methods, 8:277-284, 1987.
3. H. L. Bodlaender. *Discovering treewidth*. Proceedings SOFSEM 2005: Theory and Practice of Computer Science, pages 1-16, Liptovsky Jan, Slovak Republic, 2005. Springer Verlag, Lecture Notes in Computer Science, vol. 3381.
4. H. L. Bodlaender, A. M. C. A. Koster, F. van den Eijkhof, and L. C. van der Graag. *Preprocessing for triangulation of probabilistic networks*. In J. Breese and D. Koller, editors, Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence, pages 32-39, San Francisco, 2001. Morgan Kaufmann Publishers.
5. H. L. Bodlaender. *A partial k-arboretum of graphs with bounded treewidth*. Theoretical Computer Science, 209:1-45, 1998.
6. H. L. Bodlaender. *A tourist guide through treewidth. Acta Cybernetica*, 11(1-2): 1-21,1993.
7. F. Clautiaux, J. Carlier, A. Moukrim, and S. Negre. *New lower and upper bounds for graph treewidth*. Proceedings of the Second International Workshop on Experimental and Efficient Algorithms, pages 70-80, Ascona, Switzerland, 2003. Springer Verlag, Lecture Notes in Computer Science, vol. 2647.
8. M. C. Columbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, Inc., 1980.
9. W. Cook and P. D. Seymour. *Tour Merging via branch-decomposition*. Informs J. on Computing, 15:233-248, 2003.
10. A. Kornai and Z. Tuza. *Narrowness, pathwidth, and their application in natural language processing*. Discrete Application Mathematics, 36:87-92, 1992.
11. A. M. C. A. Koster, H. L. Bodlaender, and S. van Hoesel. *Treewidth: Computational experiments*. In Hajo Broersma, Ulrich Faigle, Johann Hurink, and Stefan Pickl (editors), Electronic Notes in Discrete Mathematics, volume 8. Elsevier Science Publishers, 2001.
12. A. M. C. A. Koster, C.P.M. van Hoesel, and A.W.J. Kolen. *Solving frequency assignment problems via tree-decomposition*. Technical report RM 99/011, Maastricht University, 1999. Available at http://www.zib.de/koster/.
13. A. M. C. A. Koster, C.P.M. van Hoesel, and A.W.J. Kolen. *Lower bounds for minimum interference frequency assignment problems*. Ricerca Operativa, 30(94-95): 101-116, 2000.
14. A. M. C. A. Koster. *Frequency Assignment ? Models and Algorithms*. PhD thesis, Maastricht University, Maastricht, the Netherlands, 1999.
15. S. J. Lauritzen and D. J. Spiegelhalter. *Local computations with probabilities on graphical structures and their application to expert systems*. The Journal of the Royal Statistical Society, Series B (Methodological), 50:157-224, 1988.
16. D. J. Rose, R. E. Tarjan, and G. S. Lueker. *Algorithmic aspects of vertex elimination on graphs*. SIAM Journal on Computing, 5:266-283, 1976.
17. K. Shoikhet and D. Geiger. *A practical algorithm for finding optimal triangulations*. Proceeding National conference on Artificial Intelligence (AAAI'97), pages 185-190. Morgan Kaufmann, 1997.

# Accelerating Vickrey Payment Computation in Combinatorial Auctions for an Airline Alliance*

Yvonne Bleischwitz[1] and Georg Kliewer[2]

[1] International Graduate School of Dynamic Intelligent Systems, University of Paderborn, Fürstenallee 11, 33102 Paderborn
[2] University of Paderborn, Fürstenallee 11, 33102 Paderborn
{yvonneb, Georg.Kliewer}@upb.de

**Abstract.** Among all the variations of general combinatorial auctions, the Vickrey auction is essentially the only incentive-compatible auction. Furthermore, it is individual rational and weakly budget-balanced. In many cases these properties are very desirable. However, computing the winners and their payments in a Vickrey auction involves solving several NP-complete problems. While there have been many approaches to solve the winner determination problem via search, this search has not been extended to compute the Vickrey payments. The naive approach is to consecutively solve each problem using the same search algorithm. We present an extension to this procedure to accelerate the computation of Vickrey payments using a simple backtrack algorithm. However, our results can be applied to sophisticated branch-and-bound solvers as well. We test our approach on data evolving from a *Lufthansa* flight schedule. Data of this type might be of interest, since authentic data for combinatorial auctions is rare and much sought after. A remarkable result is that after solving the winner determination problem we can provide bounds for the remaining problems that differ from the optimal solution by only 2.2% on average. We as well manage to obtain a rapid speedup by tolerating small deviations from the optimal solutions. In all cases, the actual deviations are much smaller than the allowed deviations.

## 1 Introduction

Many recent applications of auctions require several non-identical goods to be auctioned off. This setting is quite complicated, since bidders are often interested in certain subsets of items and want to ensure to get exactly these subsets with no missing or additional items. However, since there is an exponential number of possible combinations of items, such auctions are often computationally intractable. In previous work, only a few

* This work was partially supported by the German Science Foundation (DFG) priority programme 1126 *Algorithmics of Large and Complex Netzworks*, project *Integration of network design and fleet assignment in airline optimization* under grant MO 285/15-2, by the Future and Emerging Technologies programme of EU under EU Contract 001907 DELIS, *Dynamically Evolving, Large Scale Information Systems*, and by the Future and Emerging Technologies programme of EU under EU Contract 33116 FLAGS, *Foundational Aspects of Global Computing Systems.*

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 228–239, 2005.
© Springer-Verlag Berlin Heidelberg 2005

number of combinatorial auctions were employed due to this burden. On the other hand, combinatorial auctions have very desirable properties. In a scenario where the items are just auctioned off one by one a bidder cannot be ensured that he gets the whole subset of items he desires. This drawback is abolished allowing combinatorial bids. In an auction that allocates flights between airline alliance partners this is essential because of connecting flights and logistic considerations.

It is NP-complete to determine the optimal allocation for a combinatorial auction [17], but heuristics and tractable subcases have been analyzed [9, 12, 15, 8, 5]. A number of exponential search algorithms have been employed while trying to reduce the search overhead as much as possible [13, 3, 8, 18]. Another approach uses commercial software for solving integer programs [1]. It has been shown that non-optimal allocation algorithms cannot always ensure truthfulness [9]. For an overview of combinatorial auctions we refer to de Vries et al. [17].

The goal of an auction designer is to design the auction in such a way that intended goals are met while bidders act selfishly, i.e. choosing the strategy they think is best for them. Designing auctions or more generally games (mechanisms) is the central question of mechanism design. One desirable characteristic of a mechanism is incentive-compatibility. A mechanism is incentive-compatible, if telling the truth is a dominant strategy for each bidder. In an incentive-compatible auction, the auctioneer might hope for a high revenue, since no bidder underbids. Bidders might favour incentive-compatible auctions because they do not have to carry out strategic considerations. The only combinatorial auction that accomplishes incentive-compatibility is essentially the generalized Vickrey auction (*GVA*). However, the implementation of a *GVA* requires several NP-complete problems to be solved in order to compute the payments of the bidders. These problems only differ from each other by the exclusion of one player from the auction. While mechanism design only asks how one can design systems so that agents' selfish behavior results in desired system-wide goals, algorithmic mechanism design additionally considers computational tractability. Focussing on algorithmic mechanism design, we present ideas to speed up the computation of the so-called Vickrey payments to be used in a branch-and-bound algorithm. The main idea is to use information already gained by previous search to obtain good lower bounds for the still unsolved problems. Though there has been a lot of work on computing exact solutions for the winner determination problem via search [13, 3, 8, 18], to the best of our knowledge there has been no attempt to integrate the computation of Vickrey payments in the search process. The only alternative so far is the plain consecutive execution of one winner determination problem after the other. For another mechanism design problem, the shortest path problem [7, 10], Hershberger et al. [4] show that Vickrey payments for all the agents can be computed in the same asymptotic time complexity as for one agent. For iterative auctions, Parkes [11] proposes an experimental design that implements the outcome of the *GVA* in special cases.

While there has been a large amount of work on algorithms for combinatorial auctions, it has only been tested on mostly artificially generated data so far. There is great need to obtain more realistic data in order to evaluate the algorithms in a more practical view. To the best of our knowledge, there has only been one approach to generate realistic test data before this work so far. Leyton-Brown et al. [6] present a generator that

uses real-world domains like matching or scheduling problems to obtain reasonably realistic data. However, they do not use any data that emanates from the real world. It is assumed, that real-world data might be harder to deal with than artificial data. Due to a cooperation with *Lufthansa Systems* we have access to a real-world flight schedule and are able to transform it in order to describe a combinatorial auction in which flights are auctioned off between airline alliance partners[1]. We implemented our ideas within a simple backtrack algorithm and tested it on this data. We want to emphasize that it is not the aim of this paper to compete against already existing algorithms for winner determination but to investigate how the Vickrey payment computation can be accelerated using our improvements. Replacing the simple backtrack algorithm with a more sophisticated one will retain obtained time savings due to our extensions and supposably achieve an overall speedup. The rest of the paper is structured as follows. Section 2 explains the allocation and payment rules of the *GVA*. After presenting a very simple backtrack algorithm in Sect. 3 we propose extensions in Sect. 4 in order to accelerate the Vickrey payment computation. Section 5 deals with converting the *Lufthansa* data into input for a combinatorial auction. Results are presented and interpreted in Sect. 6. Section 7 concludes.

## 2 The Generalized Vickrey Auction

The *GVA* was initially described by Varian et al. [16]. We describe the allocation and payment rules of the *GVA*. Let $[m] = \{1, \dots m\}$ be the set of objects that are auctioned off and $[n] = \{1, \dots n\}$ be the set of players. Let $S_j \subseteq [m]$ be the possible empty set of objects that are allocated to player $j$. The set of all feasible allocations is given by $K = \{S = (S_1, \dots, S_n) | S_j \subseteq [m], j \in [n] \text{ and } S_j \cap S_i = \emptyset, \ \forall i \neq j, i, j \in [n]\}$. The valuation function $v_j(S_j)$ represents the value that player $j$ assigns to the object set $S_j$ of allocation $S = (S_1, \dots, S_n)$. The set of his valuation functions is denoted by $V_j$. Let the valuation function $v_j(\cdot)$ denote his true valuation and a valuation function $\hat{v}_j(\cdot)$ denote his submitted valuation, which does not necessarily have to be his true valuation. By means of this function the mechanism can compute valuations for all possible subsets of objects. For all bidders $j \in [n]$, let $t_j$ be bidder $j$'s payment to the auctioneer. In a Vickrey auction, $t_j$ is always non-positive. This implies that bidder $j$ has to make a payment of $-t_j$ to the auctioneer. The set of admissible alternatives is

$$\mathcal{X} = \{(S, t_1, \dots, t_n) | S \in K \text{ and } t_j \in \mathbb{R}, \sum_{j \in [n]} t_j \leq 0\}.$$

The quasilinear utility for alternative $x \in \mathcal{X}$ of player $j$ is given by $u_j : \mathcal{X} \times V_j \to \mathbb{R}$ with $u_j(x, v_j(\cdot)) = v_j(S_j) + t_j$. The efficient allocation $S^* \in K$ maximizes the sum of players' values. Let $S^* = (S_1^*, \dots, S_n^*)$ be an optimal allocation and $V^*$ be the value of the optimal allocation (the social welfare):

$$S^* = \arg\max_{S \in K} \sum_{j \in [n]} \hat{v}_j(S_j) \text{ and } V^* = \max_{S \in K} \sum_{j \in [n]} \hat{v}_j(S_j).$$

[1] Available on `http://wwwcs.uni-paderborn.de/cs/ag-monien/PERSONAL/YVONNEB/`

Let $V^*_{-j}$ be the value of the optimal allocation with player $j$ excluded from the auction. Player $j$'s payment $t_j$ is defined by $t_j = -V^*_{-j} + (V^* - \hat{v}_j(S^*_j))$. A player pays the additional amount that the other players collectively add to their social welfare if player $j$ is excluded from the auction.

The following *IP* computes the efficient allocation for all valuation functions. The only adjustment to the auction setting in some cases is to include dummy items to ensure correct allocation [8]. If there is more than one bid on one object set, only the one with highest valuation is kept. The variable $x_S \in \{0,1\}$ denotes if the subset $S$ is allocated to the player that placed the highest bid on $S$. Equation (1) ensures, that no object $i \in [m]$ is allocated more than once. The *LP* relaxation in which $x_S \in [0,1]$ will be used to compute upper bounds in the backtrack algorithm in Sect. 3.

$$\begin{array}{lll} \max & \sum_{S\subseteq[m]} v(S)x_S & \\ \text{s.t.} & \sum_{S\ni i} x_S \leq 1 & \forall i \in [m] \\ & x_S \in \{0,1\} & \forall S \subseteq [m] \end{array} \tag{1}$$

## 3 A Standard Backtracking Algorithm

In this section we introduce a standard backtrack algorithm that is the basis for our investigations referred to as *BACKTRACK* in the following. A very similar representation is used by Sandholm et al. [14]. A data structure called *profile* contains all still unallocated bids and information about which objects they contain. Bids that are already a part of the solution are stored in a data structure *IN*. Let *path_value* be the revenue from bids included in *IN* on the search path so far. Let *best* be the value of the best solution found so far and $IN^*$ the set of winning bids of the best solution found so far. The algorithm is given in Fig. 1. The first four steps deal with the case in which the value of the bids winning on the current path is greater than the value of the best solution found so far. If this is the case, the set of winning bids $IN^*$ and *best* have to be updated. Steps 6 to 9 determine an upper bound of what can be reached on the current search path by computing the solution to the *LP* for the profile. If no better solution than the current best solution can provably be achieved, this branch can be cut off. If the solution to the *LP* is integral, it is the optimal assignment for the bids still unallocated given the allocation in *IN* and no further search beyond this search node is required. Steps 10 to 16 cover this case and update the best solution found so far if necessary. Using a greedy allocation algorithm, steps 17 to 21 determine a lower bound of the solution on the current search path and if necessary update the solution. Step 22 chooses the branching bid. In Steps 23 to 25, the branching bid is included into the current solution and all other bids in the profile colliding with the branching bid are removed from the profile. After calculating the value of the subtree below in step 26, the branching bid is excluded from the current solution and all bids that were removed in step 25 are reinserted. Step 29 determines the value of the subtree below for the case that the branching bid is not part of the current solution. There are several ways to modify and tune this standard backtrack algorithm. Upper bounds can be acquired by any upper bound algorithm. We choose the *LP*-relaxation because it can be solved in a reasonable amount of time and the fractional solutions serve as a sorting criteria for the greedy algorithm. The greedy

```
1.  BACKTRACK (profile, path_value){
2.    IF (path_value > best){
3.      IN* -> IN;
4.      best -> path_value;
5.    }
6.    lp_value = LP(profile);
7.    IF (lp_value + path_value <= best){
8.      return;
9.   }
10.   IF (lp solution is integral){
11.     IF (lp_value + path_value > best){
12.       best = lp_value + path_value;
13.       update IN*;
14.     }
15.   return;
16.   }
17.   greedy_value = GREEDY(profile);
18.   IF (greedy_value + path_value > best){
19.     best = greedy_value + path_value;
20.     update IN*;
21.   }
22.   choose bid b from profile
23.   delete b from profile
24.   IN = IN + {b}
25.   remove all bids b in profile that collide with b
26.   BACKTRACK(profile, path_value + v(b))
27.   IN = IN - {b}
28.   reinsert the bids that were removed in step 25
29.   BACKTRACK(profile, path_value)
30.   return;
31. }
```

**Fig. 1.** *BACKTRACK* algorithm

algorithm can as well be substituted by any lower bound algorithm. We use *CPLEX 9.0* [2] to solve the *LP*. In the greedy algorithm, bids are ordered in descending values of the *LP* solution as suggested by Nisan [8] and inserted into the solution if they do not collide with any bids in the solution already. Other suggestions for ordering the bids are given by Sandholm [13]. We always choose the branching bid to be the bid with the largest *LP* solution value of the still unallocated bids.

## 4 Acceleration of Vickrey Payment Computation

*Preprocessing* The only preprocessing that is applied within *BACKTRACK* is to delete dominated bids. A bid $b_1 = (S_1, v_1)$ on object set $S_1$ with value $v_1$ dominates a bid $b_2 = (S_2, v_2)$ if $S_1 \subseteq S_2$ and $v_1 \geq v_2$. Though this is an accurate technique to solve the winner determination problem only, this procedure might produce incorrect results if the Vickrey payments have to be computed afterwards. Consider the following setting:

| player 1 | | player 2 | | player 3 | |
|---|---|---|---|---|---|
| value | objects | value | objects | value | objects |
| 1 | {2,0} | 3 | {0,2,3} | 2 | {1} |
| 3 | {1} | | | 4 | {1,2,3,4} |

In this setting, bid $(2, \{1\})$ of player 3 is dominated by bid $(3, \{1\})$ of player 1. If we delete this bid the optimal assignment grants bid $(3, \{1\})$ to player 1 and bid $(3, \{0, 2, 3\})$ to player 2. This assignment does not change if we keep the dominated bid. However,

[2] www.cplex.com

the payment of player 1 is $-2$ for the case of deleting and $-1$ for the case of keeping the dominated bid. The bid that dominates bid $(2,\{1\})$ of player 3 belongs to player 1. If player 1 is excluded from the auction, no other bid dominates bid $(2,\{1\})$ of player 3. Consequently, this bid can be deleted for the winner determination, but has to be reinserted for the computation of player 1's payment since he is the only player that submits a dominating bid for it. The naive approach would be to eliminate all dominated bids for the winner determination, compute the efficent allocation, reinsert them again, delete the bids of player $j$, delete all dominated bids from the remaining bids, compute $t_j$, reinsert the bids deleted before, delete the bids of player $k$, and so on. In our implementation, a preprocessing phase eliminates all dominated bids. Each player remembers the bids that were deleted only because of one of his bids. If a player is excluded from the auction to compute his payment, he reinserts those bids into the auction profile. Other preprocessing techniques are proposed by Sandholm [13]. It has to be investigated, how they are to be applied to the computations of Vickrey payments.

*Bounds.* To conduct a *GVA*, one winner determination problem has to be solved for the auction comprising all players. Additionally, for each player one winner determination problem has to be solved in which the player is excluded from the auction. The main motivation for the attempt to speed up the calculation of the Vickrey payments using a branch and bound method is the possibility to use the information gained by the previous runs of the algorithm. Let $l_j$ be a lower bound for the solution value $V^*_{-j}$ of the problem excluding player $j$. Consider the first problem that includes all players. A backtrack search tree is depicted in Fig. 2. During the search for the optimal solution, we come across various feasible solutions. These solutions could already be optimal solutions for the problem with one player excluded or at least provide lower bounds. Let $FEA^{-j}$ be a feasible solution that does not assign any objects to player $j$. In that case, we can update the lower bound $l_j$ for the problem without player $j$ if $l_j < FEA^{-j}$. Let now $v_j(S_j)$ be the value of the object set $S_j \neq \emptyset$ assigned to player $j$ by a feasible allocation $FEA$. The lower bound $l_j$ can be updated, if $l_j < FEA - v_j(S_j)$. If the optimal solution does not grant any of player $j$'s bids, his payment is zero and $V^*_{-j} = V^*$. If player $j$ receives a subset $S^*_j$ in the optimal solution with value $OPT$ it has to be checked, if $l_j < OPT - v_j(S^*_j)$. All these updates can be made during the subsequent runs as well. The bounds have only been updated for the players $j$ for which the value

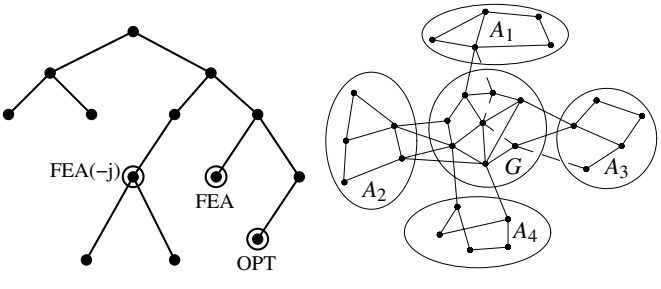



**Fig. 2.** left: *BACKTRACK* search tree. right: partition of flights

$V^*_{-j}$ has not been determined yet. To avoid a large running time overhead we only update the bounds $l_j$ if the current best solution is updated.

*Approximations.* Even if we can provide a good lower bound or even the optimal but not verified solution value for a problem, a branch-and-bound method does not necessariliy have to find the optimal solution faster than if no initial bound is given. This might be the case if a lot of similar bids were placed and various combinations of subsets yield a similar value. In that situation, the *LP* might compute an upper bound dangling just a tiny improvement and the branch cannot be cut off. One way out of this dilemma is to allow small deviations from the optimal solutions. We cut off a branch if the upper bound only differs from the current best solution by a certain percentage. This approach can as well be applied to the winner determination problem on its own. Although approximation in combinatorial auctions compromises the incentive-compatibility of the *GVA*, it is arguable if a deviation by a small percentage can be traced and taken advantage of by the bidders [10].

## 5 Testdata

Most experiments so far have been conducted on artificially generated problems. The most common distributions are random, weighted random, uniform, and decay [14, 17]. Our approach is to use a real flight schedule to be able to conduct experiments on realistic instances. This data is converted into data for a combinatorial auction. The players in this setting are partners of an airline alliance who are competing for rights to fly certain flights. A cooperation with *Lufthansa Systems* makes it possible to acquire an authentic *Lufthansa* flight schedule. Since data of a complete airline alliance is unavailable, we partition the *Lufthansa* flight schedule into several parts to simulate alliance partners. Although these data is artificially generated, it is more influenced by real-world data than other test data used in previous work.

We first want to justify the use of the *GVA* in order to allocate the flights between the alliance partners. The *GVA* is provably hard to compute. The calculation of the optimal allocation as well as the calculation of the payments for each airline are NP-hard problems in the general case. However, the optimal allocation in the *GVA* maximizes the social welfare. Since we consider an alliance, this kind of objective fits perfectly in our setting.

In the following we explain the data generation in detail. Objects are flights. For each flight, each alliance partner has a certain valuation. Flights inside Germany are considered accessible to all alliance partners. Flights outside Germany are exclusive to a single alliance partner. The aim is to allocate the flights inside Germany such that the maximal social welfare for the alliance is achieved. Let $G$ be the set of flights inside Germany and let $A_j$ be the set of flights that are exclusively flown by airline $j$. Figure 2 depicts the partition of the flights for four alliance partners. The nodes of the graph represent cities, each edge represents one leg. One possible passenger travel itinerary is illustrated by the dashed edges. For the investigated test data, the *Lufthansa* flight schedule is partitioned into 20 parts, each for one of 20 alliance partners. The set $G$ consists of 64 flights. For each airline 1000 bids on sets of flights out of $G$ are generated.

Each bid does not contain more than 20 flights. There are about 160000 itineraries that are used to determine the values of the bids. In detail, for each airline $j$ a set $G_j$ consisting of 1000 subsets $G_j(i), i = 1, \ldots, 1000$ with a maximal cardinality of 20 is determined. A subset $G_j(i)$ is called *relevant* for airline $j$, if there are itineraries that contain flights from $G_j(i)$ as well as flights from $A_j$. For each relevant subset $G_j(i) \in G_j$ the valuations of the flights in these itineraries are summed up. This sum represents the bid of airline $j$ for the set of flights $G_j(i)$. Airline $j$ bids on each relevant subset of $G_j$. We generated instances each with $100, 200, \ldots, 900$ bids per player by drawing them randomly from the generated bids. For every amount of bids per player we generated 10 instances.

## 6 Results

First, we show that the simple backtrack algorithm is comparable to other recently published backtrack algorithms. We tested *BACKTRACK* on the most common distributions as described in Sect. 5. These distributions were used by other authors as well [14, 17, 3]. We focus on the results of Sandholm et al. [14] since they claim their algorithm *CABOB* to be the currently fastest optimal algorithm for the winner determination problem. They conducted experiments against *CPLEX 7.0* on various distributions. However, they compare median running times instead of average running times. This still leaves a possibility that nearly half of the running times are arbitrarily larger than the median running time. We compare the average running times and the median running times for *CPLEX 9.0* and *BACKTRACK*. The results are shown in in Fig. 3.

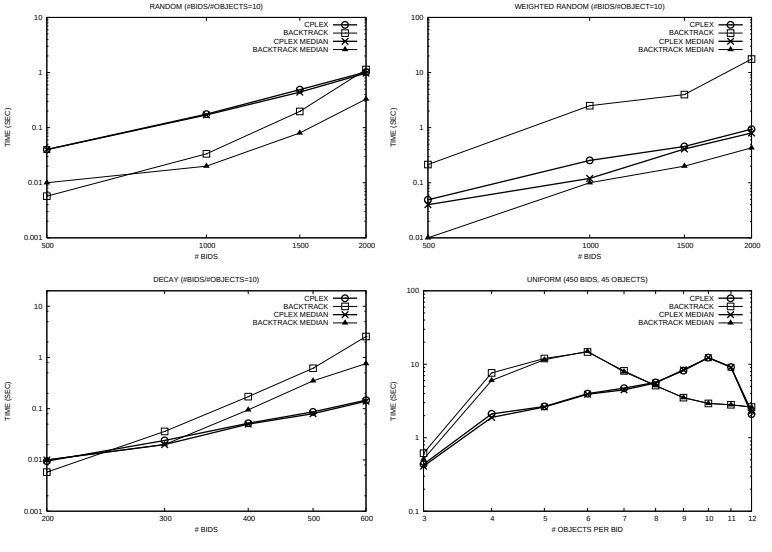



**Fig. 3.** running times on several distributions for *BACKTRACK* and *CPLEX*

There are a lot of interesting phenomenons to discuss which goes beyond the scope of this paper. For further investigation on these problem distributions, see [14, 2]. Since our algorithm is a very simple backtrack algorithm that does not use any sophisticated techniques, the running times are sometimes slower than the running times achieved before. However, they are not dramatically slower and the tendencies if a problem distribution is hard or easy are the same. Again we want to emphasize that it is not our aim to compete with *CPLEX*. We use this simple backtrack algorithm for research on how to speed up the computation of Vickrey payments in general.

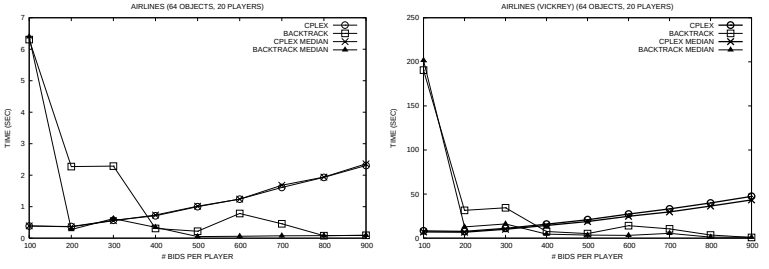



**Fig. 4.** running times on the airline distribution

The rest of our experiments uses the airline alliance test data. To get an impression of the complexity of these problems we show the average and the median running times of *CPLEX 9.0* and *BACKTRACK* in Fig. 4. On the left, only the winner determination problem is solved. On the right, the Vickrey payments are computed as well. The percentage of runs in which the first *LP* relaxation is already integral increases for the left side from 0% for 100 bids per players to 100% for 800 and 900 bids per player. The percentage of the on the spot found integer solutions in the runs computing the Vickrey payments is listed in the second column of the left table of Fig. 7. As one would expect from the results of the pure winner determination, the number rises from 0% to 90.5%. To put the large running times for a small number of bids per player in perspective, we investigate how long it takes until the solution is found and until it is verified. The large average running time for the computation of the winners for 100 bids per player is caused by essentially three instances that yield a much larger running time than the other instances. For these three instances, Fig. 5 depicts the current best solution at each timestep a solution improves during the search. In all three cases, a remarkable amount of time is needed for verification after the best solution has been found. The actual time to find the optimal solution is much smaller than the overall running time. Tuning *BACKTRACK* to increase the number of cuts might decrease the amount of time needed for verification.

Figure 6 compares the average running times and the average recursion calls needed by *BACKTRACK* to compute the Vickrey payments to the several extensions we made to *BACKTRACK* to speed up this computation. The addition *WITH BOUNDS* denotes the permanent update of the lower bounds for the values $V^*_{-j}$ for players $j$ that have not been excluded from the auction yet but have received a set of objects in the efficient

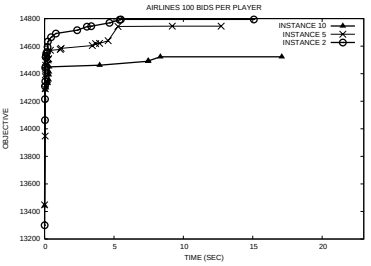



**Fig. 5.** anytime performance of *BACKTRACK* on three airline instances

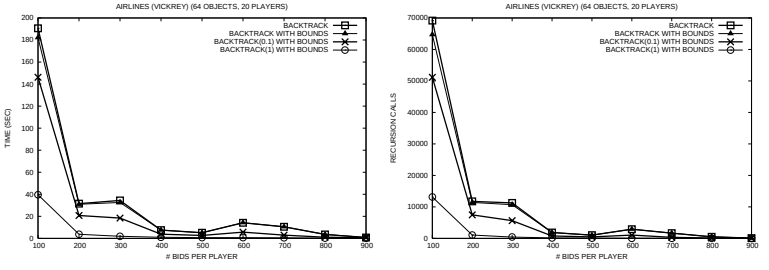



**Fig. 6.** running times and recursion calls for the airline distribution

allocation. Unfortunately, this extension on its own does not significantly reduce the amount of computation. Though this is a drawback in some respects, there are promising results as well. Figure 7 presents some of these results. The third column of the left table presents the percentage of optimal bounds, i.e. bounds that are given as initial solutions to *BACKTRACK* which are already optimal but not verified yet. The fourth column shows the average deviation of the bounds, i.e. the difference between the bounds given as initial solutions and the actual best solution. For the airline instances with 100 bids per player the percentage of 7.3% of the optimal bounds seems to be small compared to the large amount of running time. One could expect, that the more time the algorithm spends on searching, the better the bounds on $V^*_{-j}$. This conjecture is supported by the results on the average deviation. On average, the bounds given as initial

| inst. | int.sol.(%) | opt.bds.(%) | bds.dev.(%) |
|---|---|---|---|
| 100 | 0.0 | 7.3 | 2.2 |
| 200 | 17.0 | 5.3 | 3.4 |
| 300 | 30.0 | 2.8 | 3.3 |
| 400 | 45.2 | 0.0 | 3.7 |
| 500 | 61.9 | 1.8 | 4.6 |
| 600 | 70.5 | 0.0 | 4.2 |
| 700 | 67.6 | 0.0 | 4.2 |
| 800 | 84.3 | 0.0 | 6.5 |
| 900 | 90.5 | 0.0 | 9.4 |

| instance | sol.dev.(%) | | pay.dev.(%) | | opt. pay.(%) | |
|---|---|---|---|---|---|---|
| dev. | 0.1 | 1 | 0.1 | 1 | 0.1 | 1 |
| 100 | 0.03 | 0.23 | 1.56 | 43.04 | 88.52 | 24.40 |
| 200 | 0.03 | 0.20 | 1.69 | 19.86 | 91.38 | 44.98 |
| 300 | 0.00 | 0.19 | 0.16 | 17.13 | 89.00 | 45.93 |
| 400 | 0.00 | 0.15 | 3.55 | 13.94 | 88.04 | 56.94 |
| 500 | 0.00 | 0.07 | 0.43 | 2.41 | 90.43 | 76.08 |
| 600 | 0.00 | 0.05 | 1.39 | 2.86 | 94.26 | 80.38 |
| 700 | 0.00 | 0.06 | 4.28 | 6.07 | 96.17 | 82.78 |
| 800 | 0.00 | 0.02 | 0.01 | 0.58 | 98.56 | 94.26 |
| 900 | 0.00 | 0.00 | 0.00 | 0.04 | 100 | 98.09 |

**Fig. 7.** data for *BACKTRACK WITH BOUNDS* with(out) deviation

solutions only deviate from the optimal solution by 2.2%. With a decreasing percentage of integer solutions the percentage of the optimal bounds decreases and the pecentage of deviation increases. This phenomenon can be explained by less search due to a lot of initial integer *LP* solutions.

Since in a lot of cases either the bounds are very close to the optimal solution or have the same value already, we allow a small deviation from the optimal solution hoping for faster computation of near-optimal solutions or faster verification of optimal solutions in the second case. We carried out experiments allowing a deviation of 0.1% and of 1%. Figure 6 denotes these cases with *BACKTRACK(0.1) WITH BOUNDS* and *BACKTRACK(1) WITH BOUNDS*. For both cases, the running times are improved. The right table in Fig. 7 shows additional information on the experiments. Particulary interesting is the solution deviation, i.e. the actual deviation, not the *allowed* deviation from the optimal solution. Though we allow a deviation of 0.1%, the actual deviation is almost zero in all cases and never larger than 0.03%. If we accept a deviation of 1%, the actual deviation is always less than 0.23%. Despite the negative results on using approximation in truthful mechanisms, there is hope that such a small deviation is intractable by the users. One observable and inevitable effect of computing an approximation no matter how close to optimal is the change of allocation. Objects are very likely given to other players than in the efficient allocation and therefore payments and utilities change enormously. This explains the third and fourth column of the left table in Fig. 7 that show the deviation from the payments and the percentage of correctly computed payments with respect to the optimal allocation and the resulting payments.

## 7 Conclusions and Future Research

To guarantee incentive-compatibility for general combinatorial auctions, a *GVA* has to be implemented. Computing the winners and their payments in a *GVA* involves solving several NP-complete problems. Using a backtrack algorithm on each problem, we on-the-fly computed lower bounds for the remaining problems. We showed that this approach often yields very good bounds if the percentage of initial integral *LP* solutions is small. We observed that it is not sufficent to only provide good bounds in order to speed up the computation. Even if given the optimal solution as a lower bound, the backtrack algorithm might take a long time to verify this solution. Nevertheless, we took advantage of the good bounds by allowing small deviations from the optimal solution and being able to accelerate the computation. Our results support the assumption that authentic data is much harder to cope with than artificially generated data.

For additional investigation, it is a central question to what extend the incentive-compatibility remains uneffected by a small deviation from optimum. In our future research, we will incorporate our extensions into more sophisticated branch-and-bound algorithms to be able to compete with *CPLEX*. Another interesting question is to identify special structures of the airline data and try to exploit it in specialized algorithms. To get a more realistic view, we will conduct experiments that base upon more than one flight plan.

## References


1. A. Andersson, M. Tenhunen, and R. Ygge. Integer Programming for Combinatorial Auction Winner Determination. In *Proceedings of the 4th International Conference on Multi-Agent Systems*, pages 39–46, 2000.
2. Y. Bleischwitz. Kombinatorische Auktionen: Einbettung ins Mechanism Design und Algorithmen zur Erteilung der Zuschlaege, Diploma Thesis, University of Paderborn, 2004.
3. Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the Computational Complexity of Combinatorial Auctions: Optimal and Approximate Approaches. In D. Thomas, editor, *Proceedings of the 16th International Joint Conferences on Artificial Intelligence*, volume 1, pages 548–553. Morgan Kaufmann Publishers, 1999.
4. J. Hershberger and S. Suri. Vickrey Prices and Shortest Paths: What is an edge worth? In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, pages 252–259, 2001.
5. D. Lehmann, L. O'Callaghan, and Y. Shoham. Truth Revelation in Rapid, Approximately Efficient Combinatorial Auctions. *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 96–102, 1999.
6. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a Universal Test Suite for Combinatorial Auction Algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (ACM-EC)*, pages 66–76, 2000.
7. N. Nisan. Algorithms for Selfish Agents. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1999.
8. N. Nisan. Bidding and Allocation in Combinatorial Auctions. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 1–12, 2000.
9. N. Nisan and A. Ronen. Computationally Feasible VCG Mechanisms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 242–252, 2000.
10. N. Nisan and A. Ronen. Algorithmic Mechanism Design. *Games and Economic Behavior*, 35:166–196, 2001.
11. D. Parkes. An Iterative Generalized Vickrey Auction: Strategy-Proofness without Complete Revelation. In *Proceedings of the AAAI Spring Symposium on Game Theoretic and Decision Theoretic Agents*, pages 78–87, 2001.
12. M. Rothkopf, A. Pekec, and R. Harstad. Computationally Manageable Combinatorial Auctions. *Management Science*, 44(8):1131–1147, 1995.
13. T. Sandholm. An Algorithm for Optimal Winner Determination in Combinatorial Auctions. In *Artificial Intelligence*, volume 135, pages 1–54, 2002.
14. T. Sandholm, S. Suri, A. Gilpin, and D. Levine. CABOB: A Fast Optimal Algorithm for Combinatorial Auctions. In *Proceedings of the 16th International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 542–547. Morgan Kaufmann Publishers, 1999.
15. M. Tennenholtz. Some Tractable Combinatorial Auctions. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 98–103, 2000.
16. H. Varian and J. MacKie-Mason. Generalized Vickrey Auctions. Technical Report, University of Michigan, 1995.
17. R. Vohra and S. de Vries. Combinatorial Auctions: A Survey. *INFORMS Journal of Computing*, 15(3), 2003.
18. E. Zurel and N. Nisan. An Efficient Approximate Allocation Algorithm for Combinatorial Auctions. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*, pages 125–136, 2001.

# Algorithm Engineering for Optimal Graph Bipartization

Falk Hüffner*

Institut für Informatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena
hueffner@minet.uni-jena.de

**Abstract.** We examine exact algorithms for the NP-complete Graph Bipartization problem that asks for a minimum set of vertices to delete from a graph to make it bipartite. Based on the "iterative compression" method recently introduced by Reed, Smith, and Vetta, we present new algorithms and experimental results. The worst-case time complexity is improved from $O(3^k \cdot kmn)$ to $O(3^k \cdot mn)$, where $n$ is the number of vertices, $m$ is the number of edges, and $k$ is the number of vertices to delete. Our best algorithm can solve all problems from a testbed from computational biology within minutes, whereas established methods are only able to solve about half of the problems within reasonable time.

## 1 Introduction

There has recently been a much increased interest in exact algorithms for NP-hard problems [23]. All of these exact algorithms have exponential run time, which at first glance seems to make them impractical. This conception has been challenged by the view of *parameterized complexity* [6]. The idea is to accept the seemingly inevitable combinatorial explosion, but to confine it to one aspect of the problem, the *parameter*. If for relevant inputs this parameter remains small, then even large problems can be solved efficiently. Problems for which this "confining" is possible are called *fixed-parameter tractable*.

The problem we focus on here is Graph Bipartization, also known as Maximum Bipartite Subgraph or Odd Cycle Transversal. It is NP-complete [13] and MaxSNP-hard [19]; the best known polynomial-time approximation is by a logarithmic factor [9]. It has numerous applications, for example in VLSI design [1, 12], computational biology [21, 18], and register allocation [24].

In a recent breakthrough paper, solving a more than five years open question [14], Reed, Smith, and Vetta [20] proved that the Graph Bipartization problem on a graph with $n$ vertices and $m$ edges is solvable in $O(4^k \cdot kmn)$ time, where $k$ is the number of vertices to delete. The basic idea is to construct size-$k$ solutions from already known size-$(k+1)$ solutions, the so-called *iterative*

* Supported by the Deutsche Forschungsgemeinschaft, Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 240–252, 2005.
© Springer-Verlag Berlin Heidelberg 2005

*compression*. Their algorithm is of high practical interest for several reasons: the given fixed-parameter complexity promises small run times for small parameter values; no intricate algorithmic concepts with extensive implementation requirements or large hidden runtime costs are used as building blocks; and being able to "optimize" given solutions, it can be combined with known and new heuristics.

In this work we demonstrate by experiments that iterative compression is in fact a worthwhile alternative for solving Graph Bipartization in practice. Thereby, we also shed more light on the potential of iterative compression, which has already found applications in other areas as well [3, 4, 11, 16]. The structure of this work is as follows: In Sect. 3 we give a top-down presentation of the Reed-Smith-Vetta algorithm with the goal of making this novel algorithm technique accessible to a broader audience. Moreover, it prepares the ground for several algorithmic improvements in Sect. 4. In Sect. 5 we present experimental results with real-world data (Sect. 5.1), synthetic application data (Sect. 5.2), and random graphs (Sect. 5.3).

## 2 Preliminaries

By default, we consider only undirected graphs $G = (V, E)$ without self-loops, where $n := |V|$ and $m := |E|$. We use $G[V']$ to denote the subgraph of $G$ induced by the vertices $V' \subseteq V$. For a set of vertices $V' \subseteq V$, we write $G \setminus V'$ for the graph $G[V \setminus V']$. With a *side* of a bipartite graph $G$, we mean one of the two classes of an arbitrary but fixed two-coloring of $G$. A *vertex cut* between two disjoint vertex sets in a graph is a set of vertices whose removal disconnects these two sets in the graph.

**Definition 1 (Graph Bipartization).** *Given an undirected graph $G = (V, E)$ and a nonnegative integer $k$. Does $G$ have an* odd cycle cover *$C$ of size at most $k$, that is, is there a subset $C \subseteq V$ of vertices with $|C| \leq k$ such that each odd cycle in $G$ contains at least one vertex from $C$? Note that the removal of all vertices in $C$ from $G$ results in a bipartite graph.*

We investigate Graph Bipartization in the context of parameterized complexity [6] (see [5, 7, 8, 17] for recent surveys). A parameterized problem is called *fixed-parameter tractable* if it can be solved in $f(k) \cdot n^{O(1)}$ time, where $f$ is a function solely depending on the parameter $k$, not on the input size $n$.

For comparison, we examined two alternative implementations: one by Wernicke based on Branch-and-Bound [22], and one based on the following simple integer linear program (ILP):

$$
\begin{aligned}
& C_1, \ldots, C_n, s_1, \ldots, s_n : \text{binary variables} \\
& \text{minimize} \textstyle\sum_{i=1}^{n} C_i \\
& \text{s. t. } \forall \{v, w\} \in E : s_v + s_w + (C_v + C_w) \geq 1 \\
& \qquad \forall \{v, w\} \in E : s_v + s_w - (C_v + C_w) \leq 1
\end{aligned}
$$

The ILP performs surprisingly well; when solved by GNU GLPK [15], it consistently outperforms the highly problem-specific Branch-and-Bound approach

by Wernicke on our test data, sometimes by several orders of magnitude. Therefore, we use it as the main comparison point for the performance of our algorithms.

## 3 A Top-Down Presentation of the Reed-Smith-Vetta Algorithm

In this section we present in detail the algorithm for Graph Bipartization as described by Reed, Smith, and Vetta [20]. While they focus on the correctness proof and describe the algorithm only implicitly, we give a top-down description of the algorithm while arguing for its correctness, thereby hopefully making the result of Reed et al. more accessible.

The global structure is illustrated by the function Odd-Cycle-Cover. It takes as input an arbitrary graph and returns a minimum odd cycle cover.

```
Odd-Cycle-Cover(G = (V, E))
1  V' ← ∅
2  C ← ∅
3  for each v in V
4      do V' ← V' ∪ {v}
5          C ← Compress-OCC(G[V'], C ∪ {v})
6  return C
```

The routine Compress-OCC takes a graph $G$ and an odd cycle cover $C$ for $G$, and returns a smaller odd cycle cover for $G$ if there is one; otherwise, it returns $C$ unchanged. Therefore, it is a loop invariant that $C$ is a minimum odd cycle cover for $G[V']$, and since eventually $V' = V$, we obtain an optimal solution for $G$.

It remains to implement Compress-OCC. The idea is to use an auxiliary graph $H(G, C)$ constructed from $G = (V, E)$ and $C$ as follows (see Fig. 1 (a) and (b)):

- Remove the vertices in $C$ from $G$ and determine the sides of the remaining bipartite graph (in Fig. 1 (a), one side comprises $\{b, d\}$ and the other $\{e, f, h\}$).
- For each $c \in C$, add a vertex $c_1$ to one side and another vertex $c_2$ to the other side.
- For each edge $\{v, c\} \in E$ with $v \notin C$ and $c \in C$, connect $v$ to that vertex from $c_1$ and $c_2$ that is on the other side (see the bold lines in Fig. 1 (b)).
- For each edge $\{c, d\} \in E$ with both $c, d \in C$, arbitrarily connect either $c_1$ and $d_2$ or $c_2$ and $d_1$ (for example in Fig. 1, we chose $\{g_1, c_2\}$).

The crucial property of the resulting graph $H$ is that every odd cycle in $G$ that contains a vertex $c \in C$ implies a path $(c_1, \ldots, c_2)$ in $H$. This means that all odd cycles in $G$ can be found as such paths in $H$, since the vertices in $C$ touch all odd cycles. For example, the triangle $d, c, h$ in $G$ (Fig. 1 (a)) can be found as path $(c_1, h, d, c_2)$ in $H(G, C)$ (Fig. 1 (b)).

Therefore, if we could find a set $C'$ of vertices whose removal disconnects for each $c \in C$ the two vertices $c_1$ and $c_2$ in $H$, then $C'$ is an odd cycle cover for $G$. Unfortunately, solving this multi-cut problem is still NP-complete. Consider, however, a partition of the vertices $\bigcup_{c \in C}\{c_1, c_2\}$ such that for all $c \in C$ the two copies $c_1$ and $c_2$ are in different classes (called a *valid partition* for $C$). We can find a vertex cut between the two classes of a valid partition in polynomial time by using maximum flow techniques. It is clear that such a cut is also an odd cycle cover for $G$, since in particular it separates $c_1$ and $c_2$ for each $c \in C$. It is not clear, though, that if there is a smaller odd cycle cover for $G$, then we will find it as such a cut. This is provided by the following lemma, which while somewhat technical, does not require advanced proof techniques.

**Lemma 1 ([20]).** *Consider a graph $G$ with an odd cycle cover $C$ with $|C| = k$ containing no redundant vertices, and a smaller odd cycle cover $C'$ with $C' \cap C = \emptyset$ and $|C'| < k$. Let $V_1'$ and $V_2'$ be the two sides of the bipartite graph $G \setminus C'$. Then $C'$ is a vertex cut in $H(G, C)$ between $\{c_1 \mid c \in C \cap V_1'\} \cup \{c_2 \mid c \in C \cap V_2'\}$ and $\{c_2 \mid c \in C \cap V_1'\} \cup \{c_1 \mid c \in C \cap V_2'\}$.*

That is, provided $C' \cap C = \emptyset$, we can in fact find $C'$ as a vertex cut between the two classes of a valid partition, namely the valid partition $(V_1, V_2)$ that can be constructed as follows: for $c \in C$, if $c$ is on the first side of $G \setminus C'$, put $c_1$ into $V_1$ and $c_2$ into $V_2$; otherwise, put $c_2$ into $V_1$ and $c_1$ into $V_2$. For the proof we refer to Reed et al. [20].

To meet the requirement of $C' \cap C = \emptyset$, we simply enumerate all $2^k$ subsets $Y \subseteq C$; the sets $Y$ are odd cycle covers for $G \setminus (C \setminus Y)$. We arrive at the following implementation of Compress-OCC.

```
Compress-OCC(G, C)
1  for each Y ⊆ C
2     do H ← Aux-Graph(G \ (C \ Y), Y)
3        for each  valid partition (Y1, Y2) of Y
4           do if ∃ vertex cut D in H between Y1 and Y2 with |D| < |Y|
5                 then return (C \ Y) ∪ D
6  return C
```

We examine every subset $Y$ of the known odd cycle cover $C$. For each $Y$, we look for smaller odd cycle covers for $G$ that can be constructed by replacing the vertices of $Y$ in $C$ by fewer new vertices from $V \setminus C$ (clearly, for any smaller odd cycle cover, such a $Y$ must exist). Since we thereby decided to retain the vertices in $C \setminus Y$ in our odd cycle cover, we examine the graph $G' := G \setminus (C \setminus Y)$. If we now find an odd cycle cover $D$ for $G'$ with $|D| < |Y|$, we are done, since then $(C \setminus Y) \cup D$ is an odd cycle cover smaller than $C$ for $G$. To find an odd cycle cover for $G'$, we use its auxiliary graph $H$ and Lemma 1.

*Example.* Let us now examine an example for Compress-OCC (see Fig. 1). Given is a graph $G$ and an odd cycle cover $C = \{a, c, g\}$, marked with circles (Fig. 1 (a)). Observe that partitioning the remaining vertices into $\{b, d\}$

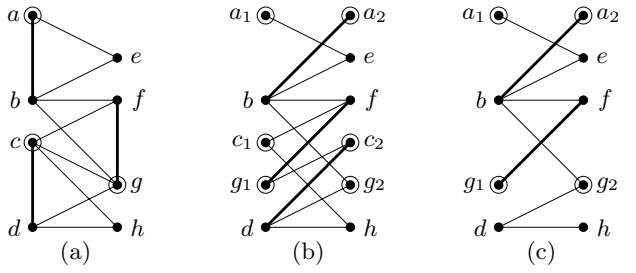



**Fig. 1.** Construction of auxiliary graphs in Compress-OCC. (a) $G$ with $C = \{a, c, g\}$; (b) $H$ for $Y = \{a, c, g\}$; (c) $H$ for $Y = \{a, g\}$

and $\{e, f, h\}$ induces a two-coloring in $G \setminus C$; only the bold edges conflict with this two-coloring in $G$. The function Compress-OCC now tries all subsets $Y$ of $C$; we give two examples, first $Y = C$. We construct the auxiliary graph $H$ (Fig. 1 (b)). Note how by selecting a suitable copy of the duplicated vertices from $Y$ for the bold edges, we can honor the two-coloring (for example, we chose $a_2$ over $a_1$ for the edge $\{a, b\}$). The algorithm will now try to find a vertex cut of size less than 3 for some valid partition. Consider for example the valid partition $\{a_1, c_2, g_1\}$ and $\{a_2, c_1, g_2\}$. With $(a_1, e, b, a_2)$, $(c_2, d, g_2)$, and $(g_1, f, c_1)$, we can find 3 vertex-disjoint paths between the two classes, so there is no vertex cut smaller than 3. In fact, for this choice of $Y$, there is no valid partition with a vertex cut smaller than 3. Next, we examine the case $Y = \{a, g\}$ (Fig. 1 (c)). Here we succeed: for the valid partition $\{a_1, g_1\}$, $\{a_2, g_2\}$, the set $D := \{b\}$ is a vertex cut of size 1. Note this valid partition corresponds to a two-coloring of $G \setminus ((C \setminus Y) \cup D)$. We can now construct a smaller odd cycle cover for $G$ as $(C \setminus Y) \cup D = \{b, c\}$.

Note that although Lemma 1 does not promise it, we might also find a vertex cut that leads to a smaller odd cycle cover for some $Y$ with $Y \cap C' \neq \emptyset$. For example, had we chosen to insert the edge $\{c_1, g_2\}$ instead of $\{c_2, g_1\}$ in Fig. 1 (b), we would have found the cut $\{b, c_1\}$ between $\{a_1, c_1, g_1\}$ and $\{a_2, c_2, g_2\}$, leading to the odd cycle cover $\{b, c\}$. Therefore, in practice one can find a smaller odd cycle cover often much faster than predicted by the worst case estimation.

*Running Time.* Reed et al. [20] state the run time of their algorithm as $O(4^k \cdot kmn)$; a slightly more careful analysis reveals it as $O(3^k \cdot kmn)$. For this, note that in effect the two loops in line 1 and 3 of Compress-OCC iterate over all possible assignments of each $c \in C$ to 3 roles: either $c \in C \setminus Y$, or $c \in Y_1$, or $c \in Y_2$. Therefore, we solve $3^k$ flow problems, and since we can solve one flow problem in $O(km)$ time by the Edmonds-Karp algorithm [2], the run time for one invocation of Compress-OCC is $O(3^k \cdot km)$. As Odd-Cycle-Cover calls Compress-OCC $n$ times, we arrive at an overall run time of $O(3^k \cdot kmn)$.

**Theorem 1.** Graph Bipartization *can be solved in* $O(3^k \cdot kmn)$ *time.*

## 4 Algorithmic Improvements

In this section we present several improvements over the algorithm as described by Reed et al. [20]. We start with two simple improvements that save a constant factor in the run time. In Sect. 4.1 we then show how to save a factor of $k$ in the run time, and in Sect. 4.2 we present the improvement which gave the most pronounced speedups in our experiments presented in Sect. 5.

First, it is easy to see that each valid partition $(Y_1, Y_2)$ is symmetric to $(Y_2, Y_1)$ when looking for a vertex cut, and therefore we can arbitrarily fix the allocation of one vertex to $Y_1$, saving a factor of 2 in the run time.

The next improvement is justified by the following lemma.

**Lemma 2.** *Given a graph $G = (V, E)$, a vertex $v \in V$, and a minimum odd cycle cover $C$ for $G \setminus \{v\}$ with $|C| = k$. Then no odd cycle cover of size $k$ for $G$ contains $v$.*

*Proof.* If $C'$ is an odd cycle cover of size $k$ for $G$, then $C' \setminus \{v\}$ is an odd cycle cover of size $k-1$ for $G[V \setminus \{v\}]$, contradicting that $|C|$ is of minimum size. □

With Lemma 2 it is clear that the vertex $v$ we add to $C$ in line 5 of Odd-Cycle-Cover cannot be part of a smaller odd cycle cover, and we can omit the case $v \notin Y$ in Compress-OCC, saving a third of the cases.

### 4.1 Exploiting Similarity of Flow Subproblems

The idea here is that the flow problems solved in Compress-OCC are "similar" in such a way that we can "recycle" the flow networks for each problem. Recall that each flow problem corresponds to one assignment of the vertices in $C$ to the three roles "$c_1$ source, $c_2$ target" ($c \in Y_1$), "$c_2$ source, $c_1$ target" ($c \in Y_2$), and "not present" ($c \in C \setminus Y$). Using a so-called $(3, k)$-ary Gray code [10], we can enumerate these assignments in such a way that adjacent assignments differ in only one element. For each of these (but the first one), one can solve the flow problem by adapting the previous flow:

- If the affected vertex $c$ was present previously, zero the flow along the paths with end points $c_1$ resp. $c_2$ (note they might be identical).
- If $c$ is present in the updated assignment, find an augmenting path from $c_1$ to $c_2$ resp. from $c_2$ to $c_1$.

Since each of these operations can be done in $O(m)$ time, we can perform the update in $O(m)$ time, as opposed to $O(km)$ time for solving a flow problem from scratch. This improves the overall worst case run time to $O(3^k \cdot mn)$. We call this algorithm OCC-Gray.

**Theorem 2.** Graph Bipartization *can be solved in $O(3^k \cdot mn)$ time.*

### 4.2 Enumeration of Valid Partitions

Lemma 1 tells us that given the correct subset $Y$ of an odd cycle cover $C$, there is a valid partition for $Y$ such that we will find a cut in the auxiliary graph leading to a smaller odd cycle cover $C'$. Therefore, simply trying all valid partitions will be successful. However, Lemma 1 even describes the valid partition that will lead to success: it corresponds to a two-coloring of the vertices in $G \setminus C'$. This allows us to omit some valid partitions from consideration. If for example there is an edge between two vertices $c, d \in Y$, then any two-coloring of $G \setminus C'$ must place $c$ and $d$ on different sides. Therefore, we only need to consider valid partitions that place $c$ and $d$ into different classes. This leads to the following modification of Compress-OCC:

```
Compress-OCC'(G = (V, E), C)
1  for each  bipartite subgraph B of G[C]
2     do for each  two-coloring V1, V2 of B
3           do H ← Aux-Graph(G \ (C \ V(B)), V(B))
4              if ∃ vertex cut D in H between V1 and V2 with |D| < |V(B)|
5                 then return (C \ V(B)) ∪ D
6  return C
```

The correctness of this algorithm follows directly from Lemma 1. The worst case for Compress-OCC' is that $C$ is an independent set in $G$. In this case, every subgraph of $G[C]$ is bipartite and has $2^{|C|}$ two-colorings. This leads to exactly the same number of flow problems solved as for Compress-OCC. In the best case, $C$ is a clique, and $G[C]$ has only $O(|C|^2)$ bipartite subgraphs, each of which admits (up to symmetry) only one two-coloring.

It is easy to construct a graph where any optimal odd cycle cover is independent; therefore the described modification does not lead to an improvement of the worst-case run time. However, at least in a dense graph, it is "unlikely" that the odd cycle covers are completely independent, and already a few edges between vertices of the odd cycle cover can vastly reduce the required computation.

Note that with a simple branching strategy, one can enumerate all bipartite subgraphs of a graph and all their two-colorings with constant cost per two-coloring. This can also be done in such a way that modifications to the flow graph can be done incrementally, as described in Sect. 4.1. The two simple improvements mentioned at the beginning of this section also can still be applied. We call the thus modified algorithm OCC-Enum2Col.

It seems plausible that for dense graphs, an odd cycle cover is "more likely" to be connected, and therefore this heuristic is more profitable. Experiments on random graphs confirm this (see Sect. 5.3). This is of particular interest because other strategies (such as reduction rules [22]) seem to have a harder time with dense graphs than with sparse graphs, making hybrid algorithms appealing.

## 5 Experiments

*Implementation Details.* The program is written in the C programming language and consists of about 1400 lines of code. The source and the test data are available from http://www.minet.uni-jena.de/~hueffner/occ.

*Data Structures.* Over 90% of the time is spent in finding an augmenting path within the flow network; all that this requires from a graph data structure is enumerating the neighbors of a given vertex. The only other frequent operation is "enabling" or "disabling" vertices as determined by the Gray code (see Sect. 4.1). In particular, it is not necessary to quickly add or remove edges, or query whether two vertices are neighbored. Therefore, we chose a very simple data structure, where the graph is represented by an array of neighbor lists, with a null pointer denoting a disabled vertex.

Since the flow simply models a set of vertex-disjoint paths, it is not necessary to store a complete $n \times n$-matrix of flows; it suffices to store the flow predecessor and successor for each node, reducing memory usage to $O(n)$.

*Finding Vertex Cuts.* It has now become clear that in the "inner loop" of the algorithm, we need to find a minimum vertex cut between two sets $Y_1$ and $Y_2$ in a graph $G$, or equivalently, a maximum set of vertex-disjoint paths between two sets. This is a classical application for maximum flow techniques: The well-known max-flow min-cut theorem tells us that the size of a minimum edge cut is equal to the maximum flow. Since we are interested in vertex cuts, we create a new, directed graph $G'$ for our input graph $G = (V, E)$: for each vertex $v \in V$, create two vertices $v_{in}$ and $v_{out}$ and a directed edge $(v_{in}, v_{out})$. For each edge $\{v, w\} \in E$, we add two directed edges $(v_{out}, w_{in})$ and $(w_{out}, v_{in})$. It is not too hard to see that a maximum flow in $G'$ between $Y_1' := \bigcup_{y \in Y_1} y_{in}$ and $Y_2' := \bigcup_{y \in Y_2} y_{out}$ corresponds to a maximum set of vertex disjoint paths between $Y_1$ and $Y_2$. Furthermore, an edge cut $D$ between $Y_1'$ and $Y_2'$ is of the form $\bigcup_{v \in V}(v_{in}, v_{out})$, and $\bigcup_{(v_{in}, v_{out}) \in D} v$ is a vertex cut between $Y_1$ and $Y_2$ in $G$.

Since we know that the cut is relatively small (less than or equal $k$), we employ the Edmonds-Karp algorithm [2]. This algorithm repeatedly finds a shortest augmenting path in the flow network and increases the flow along it, until no further increase is possible.

*Experimental Setup.* We tested our implementation on various inputs. The testing machine is an AMD Athlon 64 3400+ with 2400 MHz, 512 KB cache, and 1 GB main memory, running under the Debian GNU/Linux 3.1 operating system. The source was compiled with the GNU gcc 3.4.3 compiler with options "-O3 -march=k8". Memory requirements are around 3 MB for the iterative compression based algorithms, and up to 500 MB for the ILP.

### 5.1 Minimum Site Removal

The first test set originates from computational biology. The instances were constructed by Wernicke [22] from data of the human genome as a means to

**Table 1.** Run times in seconds for different algorithms for Wernicke's benchmark instances [22]. Runs were cancelled after 2 hours without result. We show only the instance of median size for each value of $|C|$. The column "ILP" gives the run time of the ILP given in Sect. 2 when solved by GNU GLPK [15]. The column "Reed" gives the run time of Reed et al.'s algorithm without any of the algorithmic improvements from Sect. 4 except for omitting symmetric valid partitions. The columns "OCC-Gray" and "OCC-Enum2Col" give the run time for the respective algorithms from Sect. 4.1 and 4.2. The "augmentations" colums give the number of flow augmentations performed

| | $n$ | $m$ | $\|C\|$ | ILP | Reed | | OCC-Gray | | OCC-Enum2Col | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | time [s] | time [s] | augmentations | time [s] | augmentations | time [s] | augmentations |
| Afr. #31 | 30 | 51 | 2 | 0.02 | 0.00 | 7 | 0.00 | 6 | 0.00 | 5 |
| Jap. #19 | 84 | 172 | 3 | 0.12 | 0.00 | 27 | 0.00 | 14 | 0.00 | 10 |
| Jap. #24 | 142 | 387 | 4 | 0.97 | 0.00 | 117 | 0.00 | 46 | 0.00 | 31 |
| Jap. #11 | 51 | 212 | 5 | 0.46 | 0.00 | 412 | 0.00 | 109 | 0.00 | 79 |
| Afr. #10 | 69 | 191 | 6 | 2.50 | 0.00 | 1,558 | 0.00 | 380 | 0.00 | 97 |
| Afr. #36 | 111 | 316 | 7 | 15.97 | 0.01 | 5,109 | 0.00 | 696 | 0.00 | 1,392 |
| Jap. #18 | 71 | 296 | 9 | 47.86 | 0.05 | 59,052 | 0.01 | 7,105 | 0.00 | 568 |
| Jap. #17 | 79 | 322 | 10 | 237.16 | 0.22 | 205,713 | 0.02 | 18,407 | 0.00 | 1,591 |
| Afr. #11 | 102 | 307 | 11 | 6248.12 | 0.79 | 671,088 | 0.14 | 85,851 | 0.00 | 1,945 |
| Afr. #54 | 89 | 233 | 12 | | 6.48 | 5,739,277 | 0.73 | 628,445 | 0.03 | 20,385 |
| Afr. #34 | 133 | 451 | 13 | | 10.13 | 6,909,386 | 1.04 | 554,928 | 0.04 | 16,413 |
| Afr. #52 | 65 | 231 | 14 | | 18.98 | 22,389,052 | 1.83 | 2,037,727 | 0.01 | 11,195 |
| Afr. #22 | 167 | 641 | 16 | | 350.00 | 229,584,280 | 64.88 | 15,809,779 | 0.08 | 22,607 |
| Afr. #48 | 89 | 343 | 17 | | 737.24 | 731,807,698 | 74.20 | 54,162,116 | 0.06 | 41,498 |
| Afr. #50 | 113 | 468 | 18 | | 3072.82 | 2,913,252,849 | 270.60 | 151,516,435 | 0.05 | 26,711 |
| Afr. #19 | 191 | 645 | 19 | | | | 1020.22 | 421,190,990 | 3.70 | 1,803,293 |
| Afr. #45 | 80 | 386 | 20 | | | | 2716.87 | 2,169,669,374 | 0.14 | 99,765 |
| Afr. #29 | 276 | 1058 | 21 | | | | | | 0.23 | 56,095 |
| Afr. #40 | 136 | 620 | 22 | | | | | | 0.80 | 333,793 |
| Afr. #39 | 144 | 692 | 23 | | | | | | 0.65 | 281,403 |
| Afr. #17 | 151 | 633 | 25 | | | | | | 5.68 | 2,342,879 |
| Afr. #38 | 171 | 862 | 26 | | | | | | 1.69 | 631,053 |
| Afr. #28 | 167 | 854 | 27 | | | | | | 1.02 | 464,272 |
| Afr. #42 | 236 | 1110 | 30 | | | | | | 73.55 | 22,588,100 |
| Afr. #41 | 296 | 1620 | 40 | | | | | | 236.26 | 55,758,998 |

solve the so-called Minimum Site Removal problem. The results are shown in Table 1.

As expected, the run time of the iterative compression algorithms mainly depends on the size of the odd cycle cover that is to be found. Interestingly, the ILP also shows this behavior. The observed improvement in the run time from "Reed" to "OCC-Gray" is lower than the factor of $k$ gained in the worst case complexity, but clearly still worthwhile. The heuristic from Sect. 4.2 works exceedingly well and allows to solve even the hardest instances within minutes. For both improvements, the savings in run time closely follow the savings of flow augmentations.

### 5.2 Synthetic Data from Computational Biology

In this section we examine solving the Minimum Fragment Removal [18] problem with Graph Bipartization. We generate synthetic Graph Bipartization instances using a model of Panconesi and Sozi [18], with parameters $n = 100$, $d = 0.2$, $k = 20$, $p = 0.02$, and $c$ varying (see Table 2). We refer to [18] for details on the model and its parameters.

**Table 2.** Run times in seconds for different algorithms for synthetic Minimum Fragment Removal instances [18]. Here, $c$ is a model parameter. Average over 20 instances each

| $c$ | $\|V\|$ | $\|E\|$ | $\|C\|$ | ILP | Reed | OCC-Gray | OCC-Enum2Col |
|---|---|---|---|---|---|---|---|
| 2 | 24 | 22 | 1.4 | 0.02 | 0.00 | 0.00 | 0.00 |
| 3 | 49 | 58 | 3.1 | 1.40 | 0.00 | 0.00 | 0.00 |
| 4 | 75 | 103 | 4.8 | 1538.41 | 0.02 | 0.00 | 0.00 |
| 5 | 111 | 169 | 7.7 | | 4.18 | 0.42 | 0.04 |
| 6 | 146 | 247 | 9.8 | | 5.22 | 0.68 | 0.04 |
| 7 | 181 | 353 | 13.8 | | 3044.25 | 238.80 | 1.89 |
| 8 | 214 | 447 | 14.9 | | | 4547.54 | 8.03 |
| 9 | 246 | 548 | 16.8 | | | | 17.41 |
| 10 | 290 | 697 | 20.1 | | | | 744.19 |

The results are consistent with those of Sect. 5.1. The ILP is outperformed by the iterative compression algorithms; for OCC-Gray, we get a speedup by a factor somewhat below $|C|$ when compared to "Reed". The speedup from employing OCC-Enum2Col is very pronounced, but still far below the speedup observed in Sect. 5.1. A plausible explanation is the lower average vertex degree of the input instances; we examine this further in Sect. 5.3. Note that even with all model parameters constant, run times varied by a factor of up to several orders of magnitude for all algorithms for different random instances.

### 5.3 Random Graphs

The previous experiments have established OCC-Enum2Col as a clear winner. Therefore, we now focus on charting its tractability border. We use the following method to generate random graphs with given number of vertices $n$, edges $m$, and odd cycle cover size at most $k$: Pre-allocate the roles "black" and "white" to $(n-k)/2$ vertices each, and "odd cycle cover" to $k$ vertices; select a random

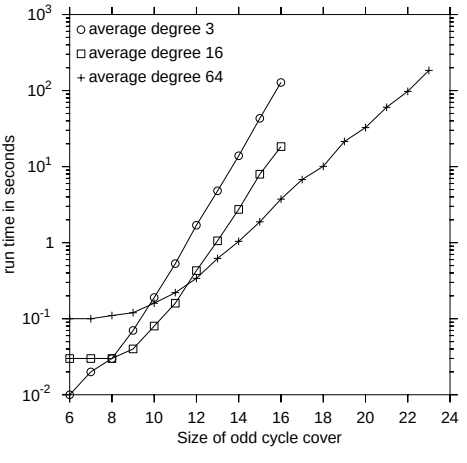



**Fig. 2.** Run time of OCC-Enum2Col (Sect. 4.2) for random graphs of different density ($n = 300$). Each point is the average over at least 40 runs

vertex and add an edge to another random vertex consistent with the roles until $m$ edges have been added.

In Fig. 2, we display the run time of OCC-Enum2Col for different sizes of the odd cycle cover and different graph densities for graphs with 300 vertices. Note that the actual optimal odd cycle cover can be smaller than the one "implanted" by our model; the figure refers to the actual odd cycle cover size $k$.

At an average degree of 3, the growth in the measurements closely matches the one predicted by the worst-case complexity $O(3^k)$. For the average degree 16, the measurements fit a growth of $O(2.5^k)$, and for average degree 64, the growth within the observed range is about $O(1.7^k)$. This clearly demonstrates the effectiveness of OCC-Enum2Col for dense graphs, at least in the range of values of $k$ we examined.

## 6 Conclusions

We evaluated the iterative compression algorithm by Reed et al. [20] for Graph Bipartization and presented several improvements. The implementation performs better than established techniques, and allows to solve instances from computational biology that previously could not be solved exactly. In particular, a heuristic (Sect. 4.2) yielding optimal solutions performs very well on dense graphs. This result makes the practical evaluation of iterative compression for other applications [3, 4, 11, 16] appealing.

*Future Work.*

- Wernicke [22] reports that data reduction rules are most effective for sparse graphs. This makes a combination with OCC-Enum2Col (Sect. 4.2) attractive, since in contrast, this algorithm displays the worst performance for sparse graphs.
- Guo et al. [11] give an $O(2^k \cdot km^2)$ time algorithm for Edge Bipartization, where the task is to remove up to $k$ edges from a graph to make it bipartite. The algorithm is based on iterative compression; it would be interesting to see whether our improvements can be applied here, and do experiments with real world data.
- Iterative compression can also be employed to "compress" a non-optimal solution until an optimal one is found. Initial experiments indicate that OCC-Enum2Col with this mode finds an optimal solution very quickly, even when starting with $C = V$, but then takes a long time to prove the optimality.

**Acknowledgements.** The author is grateful to Jens Gramm (Tübingen) and Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke (Jena) for many helpful suggestions and improvements.

## References


1. H.-A. Choi, K. Nakajima, and C. S. Rim. Graph bipartization and via minimization. *SIAM Journal on Discrete Mathematics*, 2(1):38–47, 1989.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
3. F. Dehne, M. R. Fellows, M. A. Langston, F. A. Rosamond, and K. Stevens. An $\mathcal{O}^*(2^{O(k)})$ FPT algorithm for the undirected feedback vertex set problem. Manuscript, Dec. 2004.
4. F. Dehne, M. R. Fellows, F. A. Rosamond, and P. Shaw. Greedy localization, iterative compression, and modeled crown reductions: New FPT techniques, an improved algorithm for set splitting, and a novel $2k$ kernelization for Vertex Cover. In *Proc. 1st IWPEC*, volume 3162 of *LNCS*, pages 271–280. Springer, 2004.
5. R. G. Downey. Parameterized complexity for the skeptic. In *Proc. 18th IEEE Annual Conference on Computational Complexity*, pages 147–169, 2003.
6. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
7. M. R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in FPT. In *Proc. 29th WG*, volume 2880 of *LNCS*, pages 1–12. Springer, 2003.
8. M. R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized. In *Proc. 8th WADS*, volume 2748 of *LNCS*, pages 505–520. Springer, 2003.
9. N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.
10. D.-J. Guan. Generalized Gray codes with applications. *Proceedings of the National Science Council, Republic of China (A)*, 22(6):841–848, 1998.
11. J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Improved fixed-parameter algorithms for two feedback set problems. Manuscript, Feb. 2005.
12. A. B. Kahng, S. Vaya, and A. Zelikovsky. New graph bipartizations for double-exposure, bright field alternating phase-shift mask layout. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 133–138. ACM, 2001.
13. J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.
14. M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999.
15. A. Makhorin. *GNU Linear Programming Kit Reference Manual Version 4.7*. Dept. Applied Informatics, Moscow Aviation Institute, 2004.
16. D. Marx. Chordal deletion is fixed-parameter tractable. Manuscript, Dept. Computer Science, Budapest University of Technology and Economics, Aug. 2004.
17. R. Niedermeier. Ubiquitous parameterization—invitation to fixed-parameter algorithms. In *Proc. 29th MFCS*, volume 3153 of *LNCS*, pages 84–103. Springer, 2004.
18. A. Panconesi and M. Sozio. Fast hare: A fast heuristic for single individual SNP haplotype reconstruction. In *Proc. 4th WABI*, volume 3240 of *LNCS*, pages 266–277. Springer, 2004.
19. C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.
20. B. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, 2004.

21. R. Rizzi, V. Bafna, S. Istrail, and G. Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *Proc. 2nd WABI*, LNCS, pages 29–43. Springer, 2002.
22. S. Wernicke. On the algorithmic tractability of single nucleotide polymorphism (SNP) analysis and related problems. Diplomarbeit, Univ. Tübingen, Sept. 2003.
23. G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Proc. 5th International Workshop on Combinatorial Optimization*, volume 2570 of *LNCS*, pages 185–208. Springer, 2003.
24. X. Zhuang and S. Pande. Resolving register bank conflicts for a network processor. In *Proc. 12th PACT*, pages 269–278. IEEE Press, 2003.

# Empirical Analysis of the Connectivity Threshold of Mobile Agents on the Grid*

Xavier Pérez

Dept. Llenguatges i Sistemes, Universitat Politècnica de Catalunya
xperez@lsi.upc.edu

**Abstract.** This paper gives empirical evidence about the connectivity properties of moving agents on a grid graph. The theoretical aspects of the problem were studied asymptotically in [1]. Here, it is proven that the asymptotical behaviour is also true for real size cases.

## 1 Introduction

Imagine that we have a set of $w$ agents (for example robots), with *cardinal movement (N/S/W/E)*, sampling several city levels (ambiental noise, carbon monoxide, ozone, humidity, etc.). The robots move around the city. At regular steps of time, they stop, take their samplings, broadcast to the others, and randomly continue or change direction. The agents communicate with radio-frequency, using a simple goship protocol. One of the agents has a secondary station with sufficient power to rely the information of all agents to a base station. The agents are deployed uniformly at random through the intersections of the street. We assume the streets of the city are modelled in a grid like pattern. The first question to be studied is the threshold for connectivity between the agents; i.e. given the size of the grid and, the maximun distance of broadcast (as function of the grid size) we wish to estimate the minimun number of agents we need to insure connectivity. Then we must describe how this connectivity evolves in a dynamic setting.

For very large parameters, this situation was mathematically modelled in [1]. There, the authors study what they call the *walkers model*. They consider the graph $G = (V, E)$, $|V| = N = n^2$, to be a $n \times n$ grid embedded into a torus (to avoid considering boundary effects). A set of walkers $W$ with $|W| = w$ and a "distance" $d$ are given. The walkers are placed uniformly at random (u.a.r.) and independently on the vertices of $G$ (a vertex may contain more than one walker). Two walkers $w_1$ and $w_2$ communicate in one hop if the distance between the positions of the walkers is at most $d$. We say that they communicate if they can reach each other by a sequence of such hops. The walkers move around the graph, each one performing an independent standard random walk. At each step they move from their position to any of the 4 neighboring vertices with

* Supported by the EU 6th. FP under contract 001907 (DELIS).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 253–264, 2005.
© Springer-Verlag Berlin Heidelberg 2005

equal probability. Given $f : W \to V$ a random assignment of walkers into $V$, we define the *graph of walkers*, $G_f[W]$. The vertices of $G_f[W]$ are the vertices of $G$ that contain at least one walker (*occupied vertices*), and two vertices in $G_f[W]$ are joined by an edge iff they are at distance at most $d$ in $G$. Components in $G_f[W]$ are connected components in the usual graph theory sense. A *simple component* is an isolated vertex (i.e. an occupied vertex with no other walkers within distance $d$). In the first part, the paper [1] studies the probability of $G_f[W]$ being connected, the number of components and their sizes, under certain mild asymptotic restrictions on $w$ and $d$. In the second part, the paper studies the dynamic situation, where, from an initial placement of walkers, at each time step, every walker simultaneously moves onto a randomly selected neighbour vertex in $G$. They provide characterisations of the probability of creation and destruction of connected components, and use it to give estimations of the expected lifespan of connected components and the expected time the graph of walkers remains connected (or disconnected).

In this paper we validate empirically the asymptotics results in [1] for grids of reasonable size. In particular for the static case we deal with grids of size $N = 1000 \times 1000$, $N = 3000 \times 3000$ and $N = 10000 \times 10000$. For the dynamic case, the size is $N = 1000 \times 1000$. The experiments show that the behaviour of the model is not far in most cases from the theoretical predictions.

Although several aspects communication of moving agents in networks have been studied, in particular efficient protocols for communication, not that much has been done for studying experimentally connectivity properties of moving agents in the grid. In [4], the authors present algorithms for computing coverage properties of sensors in a random geometric graphs, but the results are of a very different nature than the one presented in this paper. More work has been done on experimentation for protocols and communication for mobility of agents (see for example [3, 2]) or with transmission power performance [5], but none that this author is aware, for estimating the evolution of the properties of the connectivity graph, as agents move simultaneously.

Through the paper, $K$ is the number of connected components in $G_f[W]$ and $X$ the number of simple components. Let $\varrho = w/N$ denote the expected number of walkers at a vertex. Let $h$ be the number of vertices in the $G$ within distance greater than 0 and at most $d$ from any given one. We do the experiment for the Manhatan distance ($\ell^1$ norm) and in this case $h = 2d(d+1)$. Equivalent results, modulo a constant, can be obtained for distances defined from other norms, in particular the euclidian distance.

## 2 Static Properties

In [1], parameters $w$ and $d$ are considered as functions of $N$ and the results are asymptotic for $N$ growing large. They require $w \to \infty$ in order to avoid small-case effects. Furthermore, they assume that $w < N \log N + O(N)$ and that $h = o(N)$ (or $d = o(n)$) since otherwise the graph $G_f[W]$ is asymptotically almost surely (a.a.s.) connected.

Let $\mu$ be the expected number of simple components. Under the mild restrictions above, the following theorems are proved in [1]:

**Theorem 1.** *The expected number $\mu$ of simple components satisfies*

$$\mu \sim N\left(1-e^{-\varrho}\right)\left(1-\frac{h}{N}\right)^{w}$$

*Furthermore, if $\mu$ is bounded then $X$ is asymptotically Poisson with mean $\mu$, whilst if $\mu$ is bounded away from 0 then $\left(1-\frac{h}{N}\right)^{w} \sim e^{-h\varrho}$ and we have $\mu \sim N\left(1-e^{-\varrho}\right)e^{-h\varrho}$.*

**Theorem 2.**
- *For $\mu \to \infty$, $G_f[W]$ is disconnected a.a.s.*
- *For $\mu = \Theta(1)$, then $K = 1 + X$ a.a.s., and $X$ is asymptotically Poisson.*
- *For $\mu \to 0$, $G_f[W]$ is connected a.a.s.*

These results, provide a sharp characterisation of the connectivity in the static case and show the existence of a *phase transition* when $\mu = \Theta(1)$. At this point, just a finite number of simple components exist (following a Poisson distribution) and the remaining walkers belong to one single component which is called the *giant component*.

From Theorem 1, the relationship between $w$ and $h$ (or $d$) at the threshold can be easily computed. One observes that a larger amount of walkers implies a smaller $h$ (or $d$) and viceversa. For instance, some usual situations can be summarized in the following

**Proposition 1.** *In the case $\mu \sim N(1-e^{-\varrho})e^{-h\varrho} = \Theta(1)$, then*

1. *$h = \Theta(1)$ iff $w = \Theta(N \log N)$,*
2. *$h = \Theta(\log N)$ iff $w = \Theta(N)$,*
3. *$h = \Theta(N^c)$ iff $w = \Theta(N^{1-c} \log N)$, for $0 < c \leq 1$,*
4. *$h = \Theta(\frac{N}{\log N})$ iff $w = \Theta(\log N \log\log N)$.*

*Proof.* If we apply logarithms to the asymptotic expression of $\mu$, we obtain that $\log N(1-e^{-\varrho}) = h\varrho + \Theta(1)$. Then, by taking into account the initial restrictions imposed to $w$ and $h$, the proof is immediate. □

In this paper, we test experimentally these results for some interesting values which may arise in real life. We deal with grids of sizes $N = 1000^2$, $3000^2$ and $10000^2$. We study each case for $d$'s of several shapes ranging from a constant to a function growing large slightly slower than $n$. For each pair $N$, $d$, we choose the amount of walkers $w$ that makes $N(1-e^{-\varrho})e^{-h\varrho} = \log 2$. (Since $w$ must be an integer, we choose the closest one.) A summary of these parameters can be found in Table 1.

Note that we are demanding $\mu = \log 2$ because the condition $\mu = \Theta(1)$ is purely asymptotic and makes no sense for fixed values of $N$. The reason for choosing $\log 2$ in particular is that then, according to the theoretical results, the number of simple components should be roughly Poisson with expectation

**Table 1.** Parameters at the phase transition ($\mu = \log 2$)

| | $N = 1000 \times 1000$ | $N = 3000 \times 3000$ | $N = 10000 \times 10000$ |
|---|---|---|---|
| $d$ constant | $d = 3$<br>$w = 555377$ | $d = 3$<br>$w = 5866110$ | $d = 3$<br>$w = 75639720$ |
| $d = \log n$ | $d = 7$<br>$w = 106128$ | $d = 8$<br>$w = 875018$ | $d = 9$<br>$w = 9079434$ |
| $d = n^{1/3}$ | $d = 10$<br>$w = 50804$ | $d = 14$<br>$w = 275985$ | $d = 22$<br>$w = 1436466$ |
| $d = n^{1/2}$ | $d = 32$<br>$w = 4113$ | $d = 55$<br>$w = 14538$ | $d = 100$<br>$w = 55931$ |
| $d = n^{2/3}$ | $d = 100$<br>$w = 301$ | $d = 208$<br>$w = 719$ | $d = 464$<br>$w = 1825$ |
| $d = n/\log n$ | $d = 145$<br>$w = 122$ | $d = 375$<br>$w = 177$ | $d = 1086$<br>$w = 249$ |

$\log 2$. This makes the probability of $G_f[W]$ being connected (or disconnected) be around $1/2$.

For each triple of parameters $N$, $w$ and $d$ previously described, we experimentally place uniformly at random (u.a.r.) $w$ walkers on a grid of size $N$, check whether $G_f[W]$ is connected or not, and count the number of occupied vertices, the number of components, the size of the biggest component and the average size of the remaining ones. We repeat this experiment independently 100 times and take averages of the observed magnitudes. Then we compare the obtained data with what we would expect according to the theoretical results. In fact, for large $N$, these magnitudes approach the expressions in Table 2.

**Table 2.** Asymptotic expected values for $N$ growing large

| | |
|---|---|
| Occupied vertices | $N(1 - e^{-\varrho})$ |
| Probability that $G_f[W]$ is connected | $e^{-\mu}$ |
| Number of components | $1 + \mu$ |
| Size of the biggest component | $N(1 - e^{-\varrho}) - \mu$ |
| Average size of the other components | 1 |

For each particular run of our experiments, our algorithm must assign at random grid coordinates $(i, j)$ to each walker. It is convenient to store this data in a Hashing table of size $w$ instead of using a $n \times n$ table in order to optimize space resources. By doing this we don't loose much time efficiency, since the cost of checking wether a given vertex is occupied remains constant in expectation. We use then a Depth-First-Search to find all components. The whole algorithm takes expected time $\Theta(wh)$ (since for each walker we examine all the grid positions

within distance $d$) and requires space $\Theta(w)$. Moreover, as we are testing it in situations where $wh \sim N \log w$, the time is roughly proportional to $N$ besides logarithmic factors.

Tables 3, 4 and 5 contrast the averages of the experimental results with the asymptotic expected values (see Table 2) for the selected parameters.

**Table 3.** Contrasted results at the phase transition for $N = 1000 \times 1000$

| $N = 1000 \times 1000$ | | Experimental average | Theoretical value |
|---|---|---|---|
| $d = 3$ | Occupied vertices | 426140.57 | 426144.12 |
| | Probability of connectivity | 0.54 | 0.50 |
| | Number of components | 1.68 | 1.69 |
| $w = 555377$ | Size of the biggest component | 426139.80 | 426143.43 |
| | Average size of other components | 1.14 | 1 |
| $d = 7$ | Occupied vertices | 100674.83 | 100690.47 |
| | Probability of connectivity | 0.40 | 0.50 |
| | Number of components | 1.89 | 1.69 |
| $w = 106128$ | Size of the biggest component | 100673.72 | 100689.78 |
| | Average size of other components | 1.23 | 1 |
| $d = 10$ | Occupied vertices | 49533.84 | 49535.06 |
| | Probability of connectivity | 0.39 | 0.50 |
| | Number of components | 1.95 | 1.69 |
| $w = 50804$ | Size of the biggest component | 49532.60 | 49534.36 |
| | Average size of other components | 1.31 | 1 |
| $d = 32$ | Occupied vertices | 4104.37 | 4104.55 |
| | Probability of connectivity | 0.37 | 0.50 |
| | Number of components | 1.97 | 1.69 |
| $w = 4113$ | Size of the biggest component | 4102.96 | 4103.86 |
| | Average size of other components | 1.53 | 1 |
| $d = 100$ | Occupied vertices | 301.00 | 300.95 |
| | Probability of connectivity | 0.36 | 0.50 |
| | Number of components | 2.16 | 1.69 |
| $w = 301$ | Size of the biggest component | 298.52 | 300.27 |
| | Average size of other components | 2.02 | 1 |
| $d = 145$ | Occupied vertices | 122.00 | 121.99 |
| | Probability of connectivity | 0.19 | 0.50 |
| | Number of components | 2.38 | 1.69 |
| $w = 122$ | Size of the biggest component | 118.05 | 121.30 |
| | Average size of other components | 2.69 | 1 |

What we described so far accounts for the situation at the *phase transition*. However, we also want to verify experimentally that there is indeed a phase transition. We consider only the case $N = 3000 \times 3000$ and deal with the same types of $d$ as before (i.e. $d =$constant, $d = \log n$, $d = n^{1/3}$, $d = n^{1/2}$, $d = n^{2/3}$ and $d = n/\log n$). For each $d$, we consider 10 different values for $w$, ranging from $w_0/5$ to $2w_0$ and equidistant, where $w_0$ is the amount of walkers needed to have $\mu = \log 2$ (see Table 1). (As before, all these quantities are rounded to the nearest integer.) For each triple of parameters $N$, $w$ and $d$, we sample at random again 100 independent instances of $G_f[W]$ and check whether they are connected. The probability of connectivity can be estimated from the ratio between *connected* outputs and the total number of trials.

Since we are just concerned with connectivity, we can slighly modify our previous algorithm to improve time performance. Given a random arrangement of walkers in the grid $G$ stored as before in a Hashing table, we first examine the

**Table 4.** Contrasted results at the phase transition for $N = 3000 \times 3000$

| $N = 3000 \times 3000$ | | Experimental average | Theoretical value |
|---|---|---|---|
| $d = 3$ | Occupied vertices | 4309968.88 | 4309990.64 |
| | Probability of connectivity | 0.49 | 0.50 |
| | Number of components | 1.67 | 1.69 |
| $w = 5866110$ | Size of the biggest component | 4309968.18 | 4309989.95 |
| | Average size of other components | 1.05 | 1 |
| $d = 8$ | Occupied vertices | 833825.19 | 833827.19 |
| | Probability of connectivity | 0.37 | 0.50 |
| | Number of components | 1.87 | 1.69 |
| $w = 875018$ | Size of the biggest component | 833824.16 | 833826.49 |
| | Average size of other components | 1.22 | 1 |
| $d = 14$ | Occupied vertices | 271795.67 | 271796.38 |
| | Probability of connectivity | 0.39 | 0.5 |
| | Number of components | 1.86 | 1.69 |
| $w = 275985$ | Size of the biggest component | 271794.6 | 271795.69 |
| | Average size of other components | 1.25 | 1 |
| $d = 55$ | Occupied vertices | 14525.60 | 14526.26 |
| | Probability of connectivity | 0.41 | 0.5 |
| | Number of components | 1.86 | 1.69 |
| $w = 14538$ | Size of the biggest component | 14524.48 | 14525.57 |
| | Average size of other components | 1.34 | 1 |
| $d = 208$ | Occupied vertices | 718.97 | 718.97 |
| | Probability of connectivity | 0.29 | 0.50 |
| | Number of components | 2.10 | 1.69 |
| $w = 719$ | Size of the biggest component | 717.16 | 718.28 |
| | Average size of other components | 1.58 | 1 |
| $d = 375$ | Occupied vertices | 176.99 | 177.00 |
| | Probability of connectivity | 0.28 | 0.50 |
| | Number of components | 2.29 | 1.69 |
| $w = 177$ | Size of the biggest component | 174.20 | 176.31 |
| | Average size of other components | 1.98 | 1 |

existence of simple components. We run along the table and, for each unmarked walker, we look for another walker within distance $d$ and mark both as "not in a simple component". If we detect one isolated walker (a simple component), we stop and output *disconnected*. Otherwise, we perform as before a Depth-First-Search to find all components. In the worst case, the algorithm has the same complexity as the previous one, but if $G_f[W]$ has some simple components, we may be lucky and have a quick output. This proves quite useful for our particular kind of graphs since simple components are very common.

The plots in Figs.1, 2 and 3 show for each grid of size $N$ and distance $d$, the evolution of the probability that $G_f[W]$ is connected as we increase the amount of walkers. The dots correspond to the experimental values we obtained. In contrast, the curves show the theoretical value of this probability according to [1]. This is asymptotically $e^{-\mu}$, where the expression of $\mu$ is given in Theorem 1.

We were using for the tests, the joint effort of 10 computers with the following power:

- Processor: AMD K6(tm) 3D processor (450 MHz)
- Main memory: 256 Mb

**Table 5.** Contrasted results at the phase transition for $N = 10000 \times 10000$

| $N = 10000 \times 10000$ | | Experimental average | Theoretical value |
|---|---|---|---|
| $d = 9$ | Occupied vertices | 8679487.90 | 8679449.86 |
| | Probability of connectivity | 0.52 | 0.5 |
| | Number of components | 1.71 | 1.69 |
| $w = 9079434$ | Size of the biggest component | 8679487.05 | 8679449.16 |
| | Average size of other components | 1.23 | 1 |
| $d = 22$ | Occupied vertices | 1426204.88 | 1426198.05 |
| | Probability of connectivity | 0.40 | 0.50 |
| | Number of components | 1.89 | 1.69 |
| $w = 1436466$ | Size of the biggest component | 1426203.82 | 1426197.36 |
| | Average size of other components | 1.19 | 1 |
| $d = 100$ | Occupied vertices | 55915.73 | 55915.36 |
| | Probability of connectivity | 0.38 | 0.50 |
| | Number of components | 1.97 | 1.69 |
| $w = 55931$ | Size of the biggest component | 55914.51 | 55914.67 |
| | Average size of other components | 1.3 | 1 |
| $d = 464$ | Occupied vertices | 1824.97 | 1824.98 |
| | Probability of connectivity | 0.37 | 0.50 |
| | Number of components | 2.04 | 1.69 |
| $w = 1825$ | Size of the biggest component | 1823.39 | 1824.29 |
| | Average size of other components | 1.48 | 1 |
| $d = 1086$ | Occupied vertices | 249.00 | 249.00 |
| | Probability of connectivity | 0.33 | 0.50 |
| | Number of components | 2.23 | 1.70 |
| $w = 249$ | Size of the biggest component | 246.25 | 248.30 |
| | Average size of other components | 2.32 | 1 |

### 2.1 Conclusions for Static Case

Our experimental results show that the qualitative behaviour of the walkers model sticks reasonably well to the theoretical predictions. In fact, we observe a clear threshold phenomenon on the connectivity property even though in some cases the observed critical point is slightly displaced from its theoretical location. Furthermore, at the phase transition, we observed that there is indeed one giant component consisting of the vast majority of walkers, and a few small components (not far from being simple in most cases).

From a quantitative point of view, the accuracy of our predictions is dramatically better for small $d$'s. This is probably due to the fact that, at the phase transition, a smaller distance $d$ requires a bigger amount of walkers $w$, and we recall that the asymptotic results in [1] require $w \to \infty$ as a regularity condition. For instance, in our last case where $d = n/logn$, the corresponding $w$ is essentially logarithmic on $N$. Then, we may need to consider exponentially huge grid sizes in order to have a big amount of walkers and get reliable predictions.

Strangely enough, for the cases we considered, the accuracy of the predictions does not seem to improve significantly as we increase the grid size $N$ from $10^6$ to $10^8$. Possibly the improvement is too small to be detected within the precision of our experiments. We could always perform more trials for each test, or we could even consider much bigger values of $N$, but this last is beyond our current computational means.

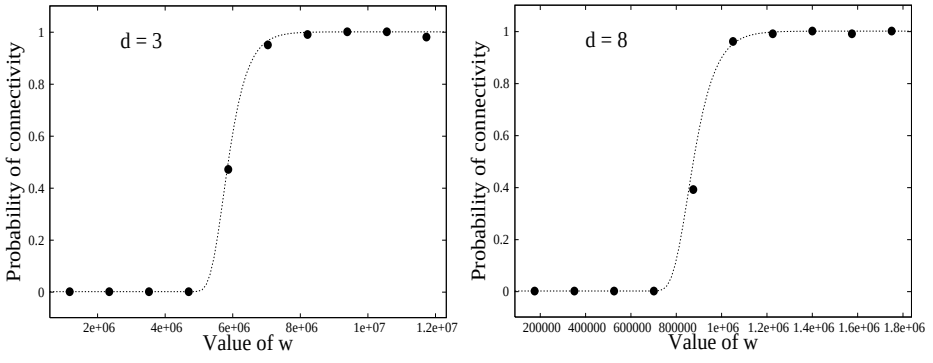



**Fig. 1.** Threshold of connectivity

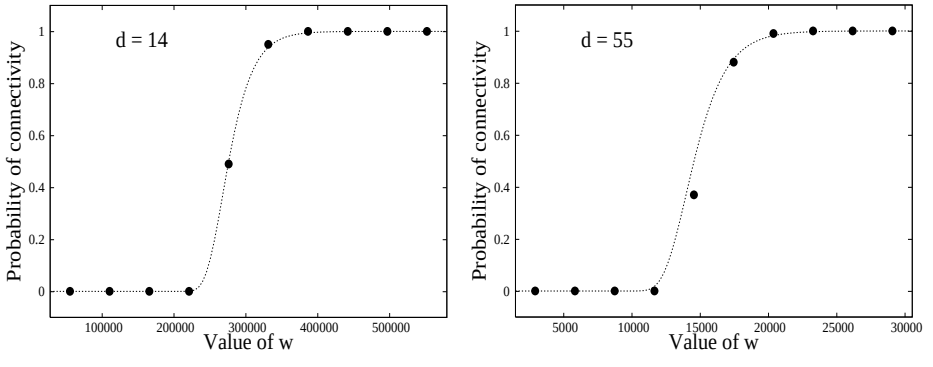



**Fig. 2.** Threshold of connectivity

## 3 Dynamic Properties

Assume that from an initial random placement $f$ of the walkers, at each time step, every walker moves from its current position to one of its 4 neighbours, with probability 1/4. This is a standard random walk on the grid for each walker.

A *configuration* is an arrangement of the $w$ walkers on the vertices of $G$. Consider the graph of configurations, where the vertices are the $N^w$ different configurations. The dynamic process can be regarded as a random walk on the graph of configurations and, in particular, as a Markov chain.

Let $G_{f_t}[W]$ denote the graph of walkers at time $t$. Note that the the walkers are u.a.r. arranged at the initial configuration and that this property stays invariant throughout the process. Hence, for any fixed $t$, we can regard $G_{f_t}[W]$ as $G_f[W]$ in the static case. Thus, if $\mu \to 0$ (or $\mu \to \infty$) then, for $t$ in any fixed bounded time interval, $G_{f_t}[W]$ is a.a.s. connected (or a.a.s. disconnected). So, for the remaining of the section, we restrict our atention to the phase transition ($\mu = \Theta(1)$) since we wish to study only the nontrivial dynamic situations.

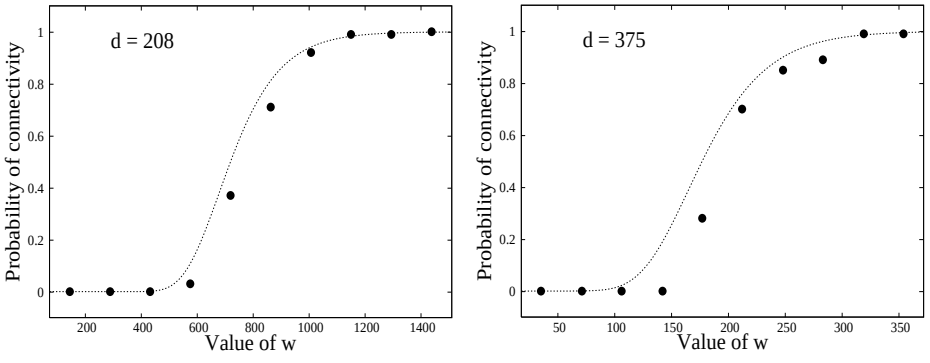



**Fig. 3.** Threshold of connectivity

In [1], is provided a full characterisation of the dynamic evolution of the system at the phase transition. As a consequence of the results obtained in the static section, it suffices to study the behaviour of the simple components.

We say that a simple component dies at vertex $v$ between times $t$ and $t+1$ if at time $t$ there is a simple component on $v$, but at time $t+1$ none of the neighbors of $v$ is a simple component (i.e. either the walkers in $v$ jump towards different directions and the component grows larger or they join another component nearby). Similarly, we say that a simple component is born at vertex $v$ between times $t$ and $t+1$ if at time $t$ none of the neighbors of $v$ is a simple component, but at time $t+1$ there is one on $v$. And finally, we say that a simple component survives at vertex $v$ between times $t$ and $t+1$ if at time $t$ there is a simple component on $v$, and all walkers there jump onto the same neighbor and stay incommunicated from the other ones.

Let $B(t)$, $D(t)$ and $S(t)$ denote the number of births deaths and survivals between times $t$ and $t+1$. They are asymptotically determined by the following theorem, provided in [1]:

**Theorem 3.** *For $t$ in any fixed bounded time interval, the random variables $S(t)$, $B(t)$ and $D(t)$ are asymptotically jointly independent Poisson, with the expectations*

$$\mathbf{E}\left[S(t)\right] \sim \begin{cases} \mu & d\varrho \to 0, \\ \mu - \lambda & d\varrho \to c, \\ 4\frac{1-e^{-\varrho/4}}{1-e^{-\varrho}} e^{-(2d+\frac{5}{4})\varrho}\mu & d\varrho \to \infty, \end{cases} \quad \mathbf{E}\left[B(t)\right] = \mathbf{E}\left[D(t)\right] \sim \begin{cases} 2d\varrho\mu & d\varrho \to 0, \\ \lambda & d\varrho \to c, \\ \mu & d\varrho \to \infty, \end{cases}$$

*where $\lambda = \left(1 - e^{-2d\varrho}\right)\mu$. Here $0 < \lambda < \mu$ for $d\varrho \to c$.*

This result has many important consequences and is the main tool for characterising the dynamic behaviour of $G_{f_t}[W]$.

For one particular history of the dynamic process, let us define $T_C$ as the average number of steps that $G_{f_t}[W]$ remains connected from each time it becomes so, after a disconnected period (i.e. the average length of the connected periods). Similarly, let $T_D$ be the average length of the disconnected periods. Also in[1] is

the following theorem, which gives the expectation of these two random variables averaged over all possible histories of the process. It is an important non-trivial consequence of Theorem 3.

**Theorem 4.** *Let $\lambda$ be defined as in Theorem 3. Then,* $\mathbf{E}[T_C] \sim \frac{1}{1-e^{-\lambda}}$ *and* $\mathbf{E}[T_D] \sim \frac{e^{\mu}-1}{1-e^{-\lambda}}$.

Our aim in this section is to validate experimentally the asympotic result in Theorem 4. We deal with grids of size $N = 1000 \times 1000$ and we consider the parameters $d$ and $w$ listed in Table 1.

For each triple $N$, $w$ and $d$, we experimentally place the walkers on the grid u.a.r. as in the static situation, but then we perform $T$ dynamic steps, for some big enough chosen $T$. We examine the connectivity of $G_{f_t}[W]$ at each time step, measure the length of the different periods we encounter in which $G_{f_t}[W]$ is (dis)connected, and then take the average. This accounts for the average time $G_{f_t}[W]$ remains (dis)connected for this particular history between times 0 and $T$. We repeat this experiment independently 50 times and take the average of the averages. We choose $T$ to be about 500 times the final value we expect to get. In our cases, this ranges from 1000 to 20000 depending on the parameters $N$, $w$ and $d$ (see last column in Table 6).

Our algorithm is an easy extension of the one used for the static situation. For each walker we choose its grid coordinates at random and store them in a Hashing table of size $w$. To perform a dynamic transition, we just need to run along the walkers and move each one to any of the 4 neighboring grid positions with equal probability. This has expected complexity $\Theta(w)$ since the expected time for search, insertion and removal in the table is constant. Besides, we must look at the connectivity at each time step, and the same observations we made in §2 apply here. (It is of great help checking the existence of simple components first, since it is usually much faster, and it is a sufficient condition for non-connectivity.) Hence, the algorithm requires space $\Theta(w)$ and takes time $O(Twh)$, but it usually runs much faster at the steps when $G_{f_t}[W]$ is disconnected.

We used the same machines and system of computation as for the static case, and the results are summarised in Table 6.

### 3.1 Conclusions for Dynamic Case

The experimental values obtained for $\mathbf{E}[T_C]$ and $\mathbf{E}[T_D]$ are in all cases of the same order of magnitud as the values predicted by the theoretical model. However, the level of accuracy is much higher for the smaller values of $d$ and gets poorer for the largest $d$'s, exactly as in the static situation. Again the reason may be that in these last cases, the considered amount of walkers $w$ is quite small, while in [1] $w$ is required to grow to infinity.

We observe as well that the average length of the disconnected periods is larger than that of the connected periods and it is much closer to the predicted value. Here is a plausible explanation to this: We were studying situations where ideally in the limit there should be one giant component and an average of $\mu = \log 2$ small (indeed simple) components. In this case the probability of

**Table 6.** Contrasted results for the dynamic proccess ($N = 1000 \times 1000$)

| $N = 1000 \times 1000$ | | Experimental average | Theoretical expectation |
|---|---|---|---|
| $d = 3$ | Time $G_{f_t}[W]$ stays connected | 1.93 | 2.05 |
| $w = 555377$ | Time $G_{f_t}[W]$ stays disconnected | 2.14 | 2.05 |
| $d = 7$ | Time $G_{f_t}[W]$ stays connected | 2.05 | 2.41 |
| $w = 106128$ | Time $G_{f_t}[W]$ stays disconnected | 2.70 | 2.41 |
| $d = 10$ | Time $G_{f_t}[W]$ stays connected | 2.28 | 2.79 |
| $w = 50804$ | Time $G_{f_t}[W]$ stays disconnected | 3.17 | 2.79 |
| $d = 32$ | Time $G_{f_t}[W]$ stays connected | 4.89 | 6.75 |
| $w = 4113$ | Time $G_{f_t}[W]$ stays disconnected | 7.56 | 6.75 |
| $d = 100$ | Time $G_{f_t}[W]$ stays connected | 14.14 | 25.36 |
| $w = 301$ | Time $G_{f_t}[W]$ stays disconnected | 27.86 | 25.13 |
| $d = 145$ | Time $G_{f_t}[W]$ stays connected | 18.97 | 41.80 |
| $w = 122$ | Time $G_{f_t}[W]$ stays disconnected | 55.20 | 42.09 |

**Table 7.** New predictions, using the observed average number of non-giant components instead of $\mu$

| $N = 1000 \times 1000$ | | Experimental average | Modified prediction |
|---|---|---|---|
| $d = 3$ | Time $G_{f_t}[W]$ stays connected | 1.93 | 2.08 |
| $w = 555377$ | Time $G_{f_t}[W]$ stays disconnected | 2.14 | 2.02 |
| $d = 7$ | Time $G_{f_t}[W]$ stays connected | 2.05 | 2.01 |
| $w = 106128$ | Time $G_{f_t}[W]$ stays disconnected | 2.70 | 2.88 |
| $d = 10$ | Time $G_{f_t}[W]$ stays connected | 2.28 | 2.20 |
| $w = 50804$ | Time $G_{f_t}[W]$ stays disconnected | 3.17 | 2.49 |
| $d = 32$ | Time $G_{f_t}[W]$ stays connected | 4.89 | 4.97 |
| $w = 4113$ | Time $G_{f_t}[W]$ stays disconnected | 7.56 | 8.15 |
| $d = 100$ | Time $G_{f_t}[W]$ stays connected | 14.14 | 15.26 |
| $w = 301$ | Time $G_{f_t}[W]$ stays disconnected | 27.86 | 33.42 |
| $d = 145$ | Time $G_{f_t}[W]$ stays connected | 18.97 | 21.35 |
| $w = 122$ | Time $G_{f_t}[W]$ stays disconnected | 55.20 | 63.51 |

connectivity would be $\mathbf{P}(K = 1) = e^{-\mu} = 1/2$, and moreover we would have $\mathbf{E}[T_C] = \frac{1}{1-e^{-\lambda}} = \frac{e^{\mu}-1}{1-e^{-\lambda}} = \mathbf{E}[T_D]$. But as shown in Figs.1, 2 and 3 the real observed probability of connectivity is mostly below the theoretical predictions at the limit, and this fact is stressed for the largest values of $d$. This is the same as saying that the phase transition occurs slightly afterwards for the observed cases than in the theoretical limit, or equivalently that the observed amount of non-giant components is slightly bigger than what we would asymptotically expect. This explains why in our experiments $\mathbf{E}[T_C] < \mathbf{E}[T_D]$.

We note that this deviation between the observed number of non-giant components and $\mu$, gets amplified in the expressions of $\mathbf{E}[T_C]$ and $\mathbf{E}[T_D]$ since $\mu$ appears there exponentially. So let's try the following: let us use the average number of non-giant components we observed (see Table 3) as the value of $\mu$ in the expressions in Theorem 4. Then, in Table 7 we compare the obtained values with our observations. The new predictions turn out to be much closer to the experimental quantities.

This gives reasonable evidence for the validity of Theorem 4, but also restricts its applicabilaty to the cases where the number of non-giant components is close to the expected number $\mu$ of simple components in the limit.

## 4 Further Problems

Similar theoretical work has already been done for agents moving on the cycle and the hypercube. Currently, asymptotic results are being studied on the behaviour of a dynamically evolving random geometric graph, which models a generic type of Mobile Ad hoc Network. Since, from a practical point of view, the cycle has no interest for the empirical study of real life size networks, we intend to continue the research for the hypercube and the random geometric graphs.

## References

1. J. Díaz, X. Pérez, M. Serna and N. C. Wormald, The walkers problem on the cycle and the grid. *Proc. of 22nd Annual Symposium on Theoretical Aspects of Computer Science*, 353–364, 2005
2. A. Jardosh, E. Belding-Royer, K. Almeroth and S. Suri. Towards Realisitic Mobility Models for Mobile Ad hoc Networks. *ACM Mobicom*, San Diego, 2003.
3. E. Jennings and C. Okino, On the diameter of sensor networks. *Preceedings of the IEEE Aerospace Conference*, Montana, 2002.
4. S. Meguerdichian, F. Coushanfar, M. Potkonjak and M. B. Srivastava, Coverage problems in wireless ad-hoc sensor networks. *Proc. of INFOCOM*, 1380–1387, 2001.
5. E. M. Royer, P.M. Melliar-Smith and L. E. Moser, An anlysis of the optimum node density for ad hoc mobile networks *Proc. of IEEE International Conference on Communications*, 857–861, 2001.

# Multiple-Winners Randomized Tournaments with Consensus for Optimization Problems in Generic Metric Spaces

Domenico Cantone, Alfredo Ferro, Rosalba Giugno,
Giuseppe Lo Presti, and Alfredo Pulvirenti

Dipartimento di Matematica e Informatica,
University of Catania, Viale A. Doria,
6, 95125 Catania, Italy
{cantone, ferro, giugno, apulvirenti}@dmi.unict.it
http://www.dmi.unict.it/~ctnyu/

**Abstract.** Extensions of the randomized tournaments techniques introduced in [6, 7] to approximate solutions of 1-median and diameter computation of finite subsets of general metric spaces are proposed. In the linear algorithms proposed in [6] (resp. [7]) randomized tournaments are played among the elements of an input subset $S$ of a metric space. At each turn the residual set of winners is randomly partitioned in nonempty disjoint subsets of fixed size. The 1-median (resp. diameter) of each subset goes to the next turn whereas the residual elements are discarded. The algorithm proceeds recursively until a residual set of cardinality less than a given threshold is generated. The 1-median (resp. diameter) of such residual set is the approximate 1-median (resp. diameter) of the input set $S$. The $\mathcal{O}(n \log n)$ extensions proposed in this paper replace local single-winner tournaments by multiple-winners ones. Moreover consensus is introduced as multiple runs of the same tournament. Experiments on both synthetic and real data show that these new proposed versions give significantly better approximations of the exact solutions of the corresponding optimization problems.

## 1 Introduction

Solutions of optimization problems in generic metric spaces are crucial in the development of efficient algorithms and data structures for searching and data mining [13, 17]. Examples of such optimization problems are clustering [3], 1-median [6], and diameter computation [7].

The 1-median of a set of points $S$ in a metric space is the element of $S$ whose average distance from all points of $S$ is minimal. This problem is known to be $\Omega(n^2)$ [16]. In [16], Indyk proposed a provably correct linear $(1 + \delta)$-approximation algorithm for the 1-median problem, with an $\mathcal{O}(n/\delta^5)$ running-time. However, Indyk's solution, based on large sampling, turns out not to be practical. In [6], Cantone *et al.* proposed a practical linear randomized approximate algorithm –here referred to as the Single-Winner 1-median algorithm –

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 265–276, 2005.
© Springer-Verlag Berlin Heidelberg 2005

which outperforms Indyk's solution in practice. The Single-Winner 1-median algorithm elects as approximate 1-median of a given input set $S$ the winner of the following randomized tournament played among the elements of $S$. At each turn, the elements which passed the preceding turn are randomly partitioned into disjoint subsets, say $X_1, \ldots, X_k$. Then, each subset $X_i$ is locally processed by a procedure which computes its exact 1-median $x_i$. The resulting elements $x_1, \ldots, x_k$ move to the next turn. The tournament terminates when the number of residual winners falls under a given threshold. The exact local 1-median of the residual set is taken to approximate the exact 1-median of $S$.

The diameter of a set $S$ of points in a metric space is the maximum distance among all pairs of elements in $S$. As observed in [16], one can construct a metric space where all distances among points are set to 1 except for one (randomly chosen) distance, which is set to 2. Thus, any algorithm which computes the diameter with an approximation factor strictly greater than $\frac{1}{2}$ must necessarily examine all pairs. A randomized algorithm – here referred to as the Single-Winner diameter algorithm – has been proposed in [7]. The Single-Winner diameter algorithm, based on the computation of local diameters, plays the following randomized tournament among the elements of the input set $S$. As before, at each turn the winners of the previous turn are randomly partitioned into disjoint subsets, say $X_1, \ldots, X_k$. The endpoints of the diameter of each round $X_i$ are the winners of the current turn. The tournament terminates when the number of residual elements falls under a given threshold. The farthest pair of the residual set is the *Antipole pair* and its distance is taken to approximate the diameter of the initial set $S$.

The 1-median and diameter problems for generic metric spaces find important applications in such areas as molecular biology [15], network management [1, 5, 10], and information retrieval [12]. Among their most relevant applications here we cite:

- *approximate queries with a given threshold* on very large databases of objects belonging to clustered metric spaces. In such a problem, one seeks clusters whose representatives have distance from the query bounded by the threshold. It turns out that if the 1-medians are selected as representatives of the clusters and the clusters diameters are comparable with the threshold, then the average error during the search is minimized with very high probability [8, 9];
- *k-clustering of metric spaces*, in which iterative computations of 1-medians and diametrical pairs are required [11, 14];
- *multiple sequence alignment*, in which the goal is to find a common alignment of a set of genetic sequences [15] (this is a basic problem in biological data engineering).

In this paper we propose some improvements on the randomized tournament techniques for the 1-median and diameter computations.

The first idea is aimed at avoiding early elimination of good candidates in the initial phases of the tournament. This is achieved by enlarging the number of local winners. We call the resulting algorithms Multiple-Winners variants of the original Single-Winner 1-median and diameter algorithms. The second idea is to also use consensus [18, 19], which consists in extracting the best solution from several runs of the algorithm. We call the resulting algorithms Multiple-Winners-With-Consensus variants of the original ones. Of course, the Multiple-Winners-With-Consensus variants require more computational resources. In particular, $\mathcal{O}(n \log n)$ distance computations are needed instead of the $\mathcal{O}(n)$ distance computations required by the Single-Winner versions. However, it turns out from the experimental results that improvements in precision very much justify the slightly higher computational costs. More precisely, a thorough comparison has been carried out between the original Single-Winner randomized tournament algorithms with the newly proposed Multiple-Winners-With-Consensus variants, using both synthetic and real data sets. Synthetic data consisted of sets of randomly chosen points from highly dimensional Euclidean spaces with different distributions. Real data included strings from the Linux dictionary and image histogram databases extracted from the Corel Image database [2]. All experimental results showed that our newly proposed algorithms perform considerably better than the old versions.

## 2 Approximate Single-Winner 1-Median and Diameter Computation

In this section, we review the randomized algorithm for the approximate 1-median (resp. diameter) computation given in [6] (resp. [7]).

Let $(M, dist)$ be a metric space and let $S$ be a finite set of points in $M$. The *1-median* of $S$ is an element of $S$ whose average distance from all points of $S$ is minimal. The *farthest pair* of $S$ is a pair of points $A, B$ of $S$ such that $dist(A, B) \geq dist(x, y)$, for $x, y \in S$. The distance of the farthest pair $A, B$ is the *diameter* of $S$.

The algorithm proposed in [6] (resp. [7]) is based on a randomized tournament played among the elements of an input set $S$ taken from a given metric space $(M, dist)$. At each turn, the elements which passed the previous turn are randomly partitioned into subsets (rounds), say $X_1, \ldots, X_k$, having the same size $t$, with the possible exception of only one subset, whose size must lie between $(t + 1)$ and $(2t - 1)$. Then, the exact 1-median $x_i$ (resp. farthest pair $(a_i,b_i)$) of each subset $X_i$ is computed, for $i = 1, \ldots, k$. The elements $x_1, \ldots, x_k$ (resp. the points $(a_1,b_1)$, $\ldots$,$(a_k,b_k)$) win the round and go to the next turn. When the number of residual elements falls below a given *threshold*, the exact local 1-median (resp. farthest pair) of the residual set is taken as the approximate 1-median (resp. farthest pair) of $S$. Plainly, the requirement that no round is played with less than $t$ elements is useful to ensure statistical significance of the tournament.

In [6] (resp. [7]), it is shown that such algorithm has a worst-case complexity of $\frac{t}{2}n+o(n)$ (resp. $\frac{t(t-1)}{2(t-2)}n+o(n)$) in the input size $n$, provided that the threshold is $o(\sqrt{n})$.

## 3 Multiple-Winners Extensions

In this section, we propose improvements of the algorithms described above, starting from the following observation. If only few good candidates participate to the same round, they might be discarded early with a high probability. Thus, to avoid such loss of information, more elements must be promoted at each round.

### 3.1 Approximate Multiple-Winners 1-Median Computation

In the Multiple-Winners version of the 1-median computation, each round $r$ of the tournament is played among $2t$ elements. Let us define the *weight distance* of an element $x$ as the sum of the distances between $x$ and all other elements playing in the same round. We sort the elements in $X_r$ with respect to the weight distance in increasing order and we define $T_r$ as the sequence of the first $t$ elements in such ordering. Let $H_i$ be the set of $i$-th components of the sequences $T_r$. Intuitively, elements belonging to $H_i$ with lower index $i$ have higher probability to approximate the exact 1-median. Elements in all $H_i$ are promoted to the next turn. The tournament terminates when the number of residual winners falls under a given threshold. The exact local 1-median of the residual set is taken to approximate the exact 1-median of $S$. We assume that the parameter *threshold*, which bounds the number of residual winners, has been set to $\min\{4t-1, \lfloor\sqrt{n}\rfloor\}$, where $n$ is the size of the input set $S$.

A high level description of our proposed algorithm is the following.

1. Given a set $S$ of elements and a tournament size $t$, partition $S$ into subsets $X_r$ of size $2t$, with the possible exception of only one subset whose size must lie between $(2t+1)$ and $(4t-1)$;
2. For each subset $X_r$:
   (a) Compute the weight distance $w(x, X) = \sum_{y \in X} dist(y, x)$, for $x \in X$.
   (b) Sort the elements in $X_r$ with respect to the weight distance in increasing order and let $T_r$ be the sequence of the first $t$ elements in such ordering.
   (c) $H_i = H_i \bigcup \{(T_r)_i\}$ for $i = 1, \ldots, t$.
3. If $|H_1| \geq threshold/t$ then partition $\bigcup_{i=1}^{t} H_i$ into subsets $X_r$ of size $2t$ each, taking two random elements from each bucket $H_j$. All the subsets $X_r$ will have $2t$ elements with the possible exception of only one subset of size $3t$. Go to step 2.
4. If $|H_1| < threshold/t$ then return the exact 1-median of $\bigcup_{i=1}^{t} H_i$.

Next we analyze the computational complexity of our proposed algorithm.

**Complexity Analysis.** Let $G(n, t, \mathit{threshold})$ be the number of rounds computed at step 2 of Multiple-Winners 1-median algorithm, with an input of size $n$, and using the parameters $t \geq 2$ and $\mathit{threshold} \geq 1$. An upper bound for $G(n, t, \mathit{threshold})$, denoted with $G_1(n)$, satisfies the following recurrence relation:

$$G_1(n) = \begin{cases} 0 & \text{if } 0 \leq n \leq 1 \\ 1 & \text{if } 2 \leq n < 4t \\ \lfloor 2n/t \rfloor + G_1(\lfloor n/2t \rfloor\, t) & \text{if } n \geq 4t\,. \end{cases}$$

By induction on $n$, we can show that $G_1(n) \leq \lfloor n/t \rfloor$. For $n < 4t$, our estimate is trivially true. Thus, let $n \geq 4t$. Then by inductive hypothesis, we have

$$G_1(n) = \left\lfloor \frac{n}{2t} \right\rfloor + G_1\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq \frac{n}{2t} + \left\lceil \frac{n}{2} \times \frac{1}{t} \right\rceil = \left\lfloor \left\lceil \frac{n}{2t} \right\rceil + \frac{n}{2t} \right\rfloor \leq \left\lceil \frac{n}{t} \right\rceil .$$

The number of distance computations made at each round is equal to $\sum_{i=1}^{|X_r|}(i-1) = \frac{|X_r|(|X_r|-1)}{2}$. Each round of the first turn has size $2t$, with the possible exception of the last round, which can have size between $(2t+1)$ and $(4t-1)$. In the successive turns we may have a round of size $3t$. Since there are $\lceil \log_2 n \rceil$ rounds, it follows that the total number of distances computed by an execution of our algorithm, with $|S| = n$, a constant tournament size $2t$, and a threshold $\vartheta$, is upper bounded by the expression

$$G(n, t, \vartheta) \cdot t(2t-1) + \frac{(4t-1)(4t-2)}{2} + \lceil \log_2 n - 1 \rceil \cdot \left[ \frac{3t(3t-1)}{2} - 2(t-1) \right]$$

$$+ \frac{(\vartheta - 1)(\vartheta - 2)}{2} = n(2t-1) + \mathcal{O}(\log n + \vartheta^2).$$

### 3.2 Approximate Multiple-Winners Diameter (Antipole) Computation

In the Multiple-Winners version of the diameter computation, we replace local farthest pair computation with the generation (in the style of [14]) of the farthest sequence $T$ of $t$ points starting with the two diameter endpoints. More precisely, each round $r$ of the tournament is played among $2t$ elements. If $p_1$, $p_2$ are the two diameter endpoints of $X_r$, $p_{i+1}$ is the element of $X_r$ which maximizes the minimum distance from $p_1, \ldots, p_i$, for each $i = 2, \ldots, t-1$. Let $T_r$ be the ordered sequence $p_1, \ldots, p_t$. Let $H_i$ be the set of $i$-th components of the sequences $T_r$. We require that elements in all $H_i$ are promoted to the next turn. The tournament terminates when the number of residual elements falls under a given threshold. The farthest pair computed in the tournament is taken to approximate the diameter endpoints of the initial set $S$. We assume that the parameter *threshold*, which bounds the number of residual winners, has been set to $\min\{4t-1, \lfloor \sqrt{n} \rfloor\}$, where $n$ is the size of the input set $S$.

A high level description of our proposed algorithm is the following.

1. Set a variable $d_{max} = 0$ and $p_{max} = \emptyset$.
2. Given a set $S$ of elements in a metric space, randomly partition the input set $S$ into subsets $X_r$ of size $2t$, with the possible exception of only one subset whose size lies between $(2t + 1)$ and $(4t - 1)$.
3. For each subset $X_r$:
   (a) let $a$, $b$ the diameter endpoints of $X_r$ and let $d = dist(a, b)$;
   (b) if $d > d_{max}$ then $d_{max} = d$; $p_{max} = \{a, b\}$;
   (c) set $T$=$[a, b]$; $H_1 = H_1 \bigcup \{a\}$; $H_2 = H_2 \bigcup \{b\}$;
   (d) for $i = 2$ to $t - 1$:
      i. let $c$ be the point in $X_r \setminus T$ which maximize the minimum distance from elements of $T$;
      ii. add $c$ to the end of $T$; $H_i = H_i \cup \{c\}$.
4. If $|H_1| \geq threshold/t$ then partition $\bigcup_{i=1}^{t} H_i$ in subsets $X_r$ of size $2t$ each taking two random elements from each bucket $H_j$. All the subsets $X_r$ will have $2t$ elements with the possible exception of only one subset of size $3t$. Go to step 2.
5. If $|H_1| < threshold/t$ then let $\{a, b\}$ be the farthest pair in $\bigcup_{i=1}^{t} H_i$.
6. If $d_{max} > dist(a, b)$ then return $p_{max}$ else return $\{a, b\}$.

The proposed algorithm computes the same number of distances of the algorithm in Section 2. Therefore it has complexity $n(2t - 1) + \mathcal{O}(\log n + \vartheta^2)$, where $n$ is the size of the input, $\theta$ is the threshold, and $2t$ is the size of rounds.

## 4 Experimental Results

Experiments have shown that Multiple-Winners variants of the algorithms perform better than the respective Single-Winner versions in terms of precision. In order to further improve the precision, we applied consensus techniques [18, 19].

In our context, consensus consists of executing several independent runs of a given randomized algorithm, which aims at approximating the solution of an optimization problem. Then the best output of all runs is selected. Here we have chosen to perform $\log n$ runs of both Single-Winner and Multiple-Winners verions of 1-median and diameter computations.

Our implementations have been done in standard C (GNU-gcc compiler v.2.96) and all the experiments have been carried out on a PC Pentium IV 2.8GHz with the Linux operating system (RedHat distribution v. 8.1). Each experiment refers to $5,000$ independent executions of the algorithm on a *fixed* input set.

Due to space limitations, here we report only the experiments relative to the 1-median algorithms. The corresponding group of experiments on the farthest pair computation have similar results and can be found at the following address (http://alpha.dmi.unict.it/~ct-nyu/diam.htm).

**Asymptotic behavior of the 1-median algorithms.** Experiments in Fig. 1 report the relative frequencies histograms showing the empirical distribution of

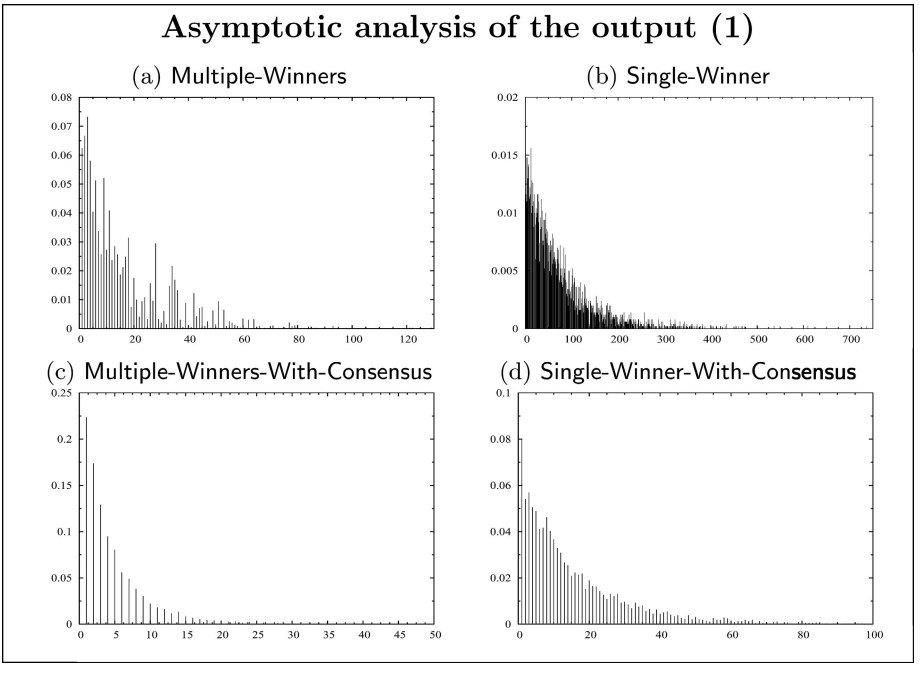



**Fig. 1.** Relative frequencies histograms of the outputs. The abscissae refer to the elements in the input set with the smallest weight, in increasing order. The leftmost element is the exact 1-median of $S$. The ordinates show the corresponding winning frequencies

the algorithms outputs, obtained with round size $2t = 4$ for the Multiple-Winners version and $t = 3$ for the Single-Winner version. The input set has size 10000, it is drawn from $[0, 1]^2$ with uniform distribution and it is equipped with the Euclidean metric $L_2$. In Fig. 2, the input sets are taken from: (i) uniformly distributed $[0, 1]^{50}$ with the Euclidean metric $L_2$, (ii) clustered points in $\mathbb{R}^2$ with the Euclidean metric $L_2$, and (iii) randomly sampled Linux words with editing distance.

In all groups of histograms, we notice that the proposed Multiple-Winners algorithm (in both versions with and without consensus) has higher winning frequencies of elements with lower weights than the the corresponding Single-Winner version.

Histograms give the output precision of our algorithms in terms of the relative position w.r.t. the exact 1-median. Nevertheless, in many applications, it is more convenient to define the output quality in terms of the weight function $w()$, by introducing the following values relatively to a generic input set $S$:

- $m_S = \min_{x \in S} w(x)$, the minimum weight in $S$, i.e., the weight of the exact *1-median*;
- $M_S = \max_{x \in S} w(x)$, the maximum weight in $S$;

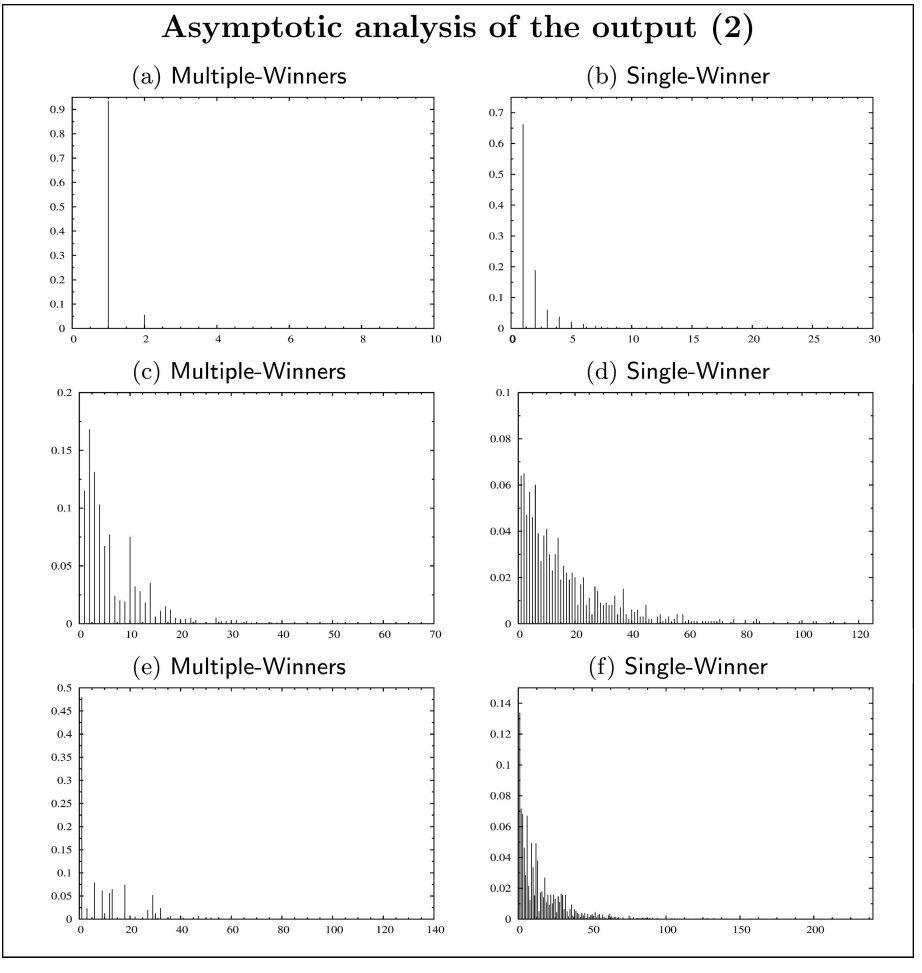



**Fig. 2.** Frequencies histograms of the algorithms outputs with consensus. The input sets are taken from: (in (a) and (b)) uniformly distributed $[0:1]^{50}$, (in (c) and (d)) clustered points in $\mathbb{R}^2$, (in (e) and (f)) sampled Linux words

- $\mu_S = E[w(x)]$ and $\sigma_S = \sigma[w(x)]$ (with $x \in S$), i.e., the average and the standard deviation of weights in $S$;
- $w_{out} = w(Output)$, the weight of the *Output* element returned by our algorithm on input $S$.

Next we introduce the relative error, relatively to a single test on a random input set $S$:

- $\epsilon_{out} = \frac{w_{out} - m_S}{M_S - m_S} \cdot 100$, the percentage error distance defined w.r.t. the largest range of values $w(x)$, with $x \in S$; the extreme values assumed by $\epsilon_{out}$ are 0% and 100%, when the minimum- and maximum-weight element in $S$ are returned by the algorithm, respectively.

Figs. 3-(a) and (b) report the average percentage error distance $E[\epsilon_{out}]$ and its standard deviation $\sigma[\epsilon_{out}]$ of experiments performed with and without consensus in $[0,1]^2$ on a fixed set $S$.

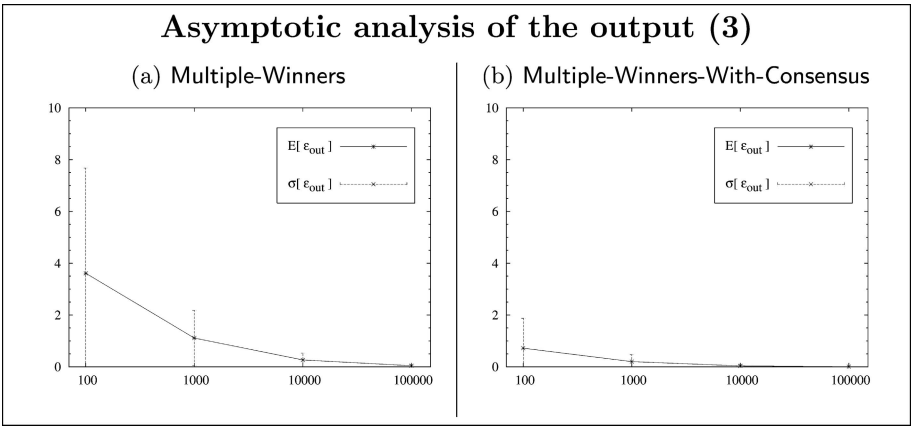



**Fig. 3.** Average percentage error $E[\epsilon_{out}]$ and standard deviation $\sigma[\epsilon_{out}]$, w.r.t. the input size, on fixed input set (a) Multiple-Winners(b) Multiple-Winners-With-Consensus

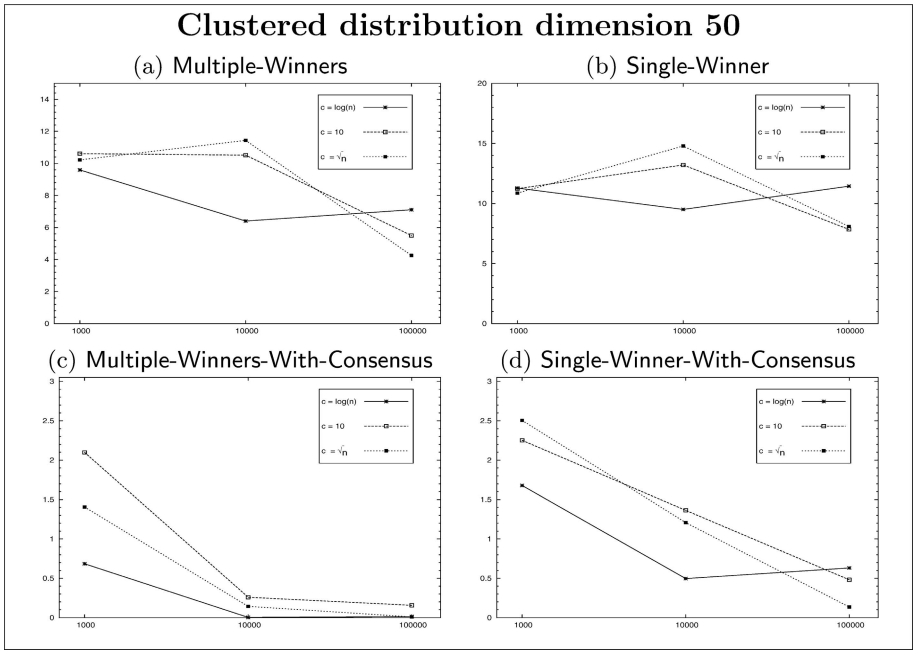



**Fig. 4.** Average percentage error $E[\epsilon_{out}]$ for different types of clustered distribution, w.r.t. the number of clusters

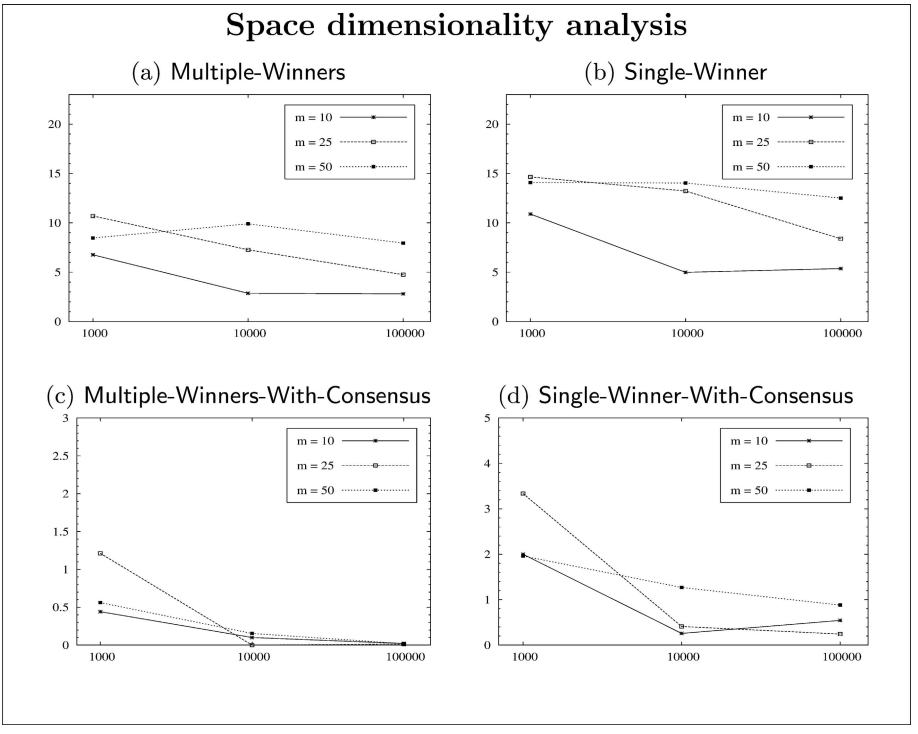



**Fig. 5.** Average percentage error $E[\epsilon_{out}]$, w.r.t. the input size, for different space dimensions

**Distribution and Space Dimensionality Analysis.** Next we analyzed the uniform distributions of $c$ clustered input sets containing $n = 10^i/c$ points each, with $i = 3, 4, 5$, and where $c = \lfloor \log_{10} n \rfloor$, 10, $\lfloor \sqrt{n} \rfloor$. We generate $c$ random clusters in $[0, 1]^{50}$, with uniform distribution in each cluster.

Clusters are characterized by a parameter $0 < \rho < \frac{1}{2}$ which determines the wideness of clusters. Such clusters are generated using the same procedures implemented for the experimental session reported in [4]. We use the Euclidean metric $L_2$, with a round size $2t = 2$ for the Multiple-Winners version and $t = 3$ for the Single-Winner version. The average percentage errors $E[\epsilon_{out}]$ are shown in Fig. 4, with wideness factor $\rho = 0.2$.

Results in Fig. 5 allow one to evaluate the performance of our algorithm in the case of a $[0, 1]^m$ metric space equipped with the metric $L_2$, for $m = 10, 25, 50$, with uniformly generated data.

**Real world data-sets analysis.** Finally we computed the average percentage error $E[\epsilon_{out}]$ of our algorithm on input data-sets extrapolated from real world databases. In Fig. 6-(a) the input set of $n = 10^i$, with $i = 2, 3, 4$, is randomly

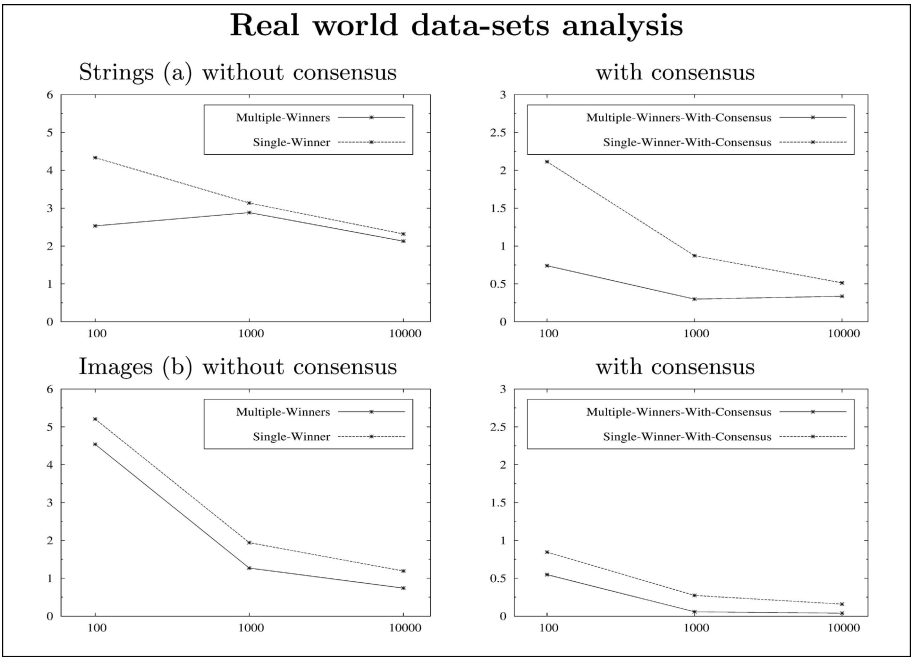



**Fig. 6.** Average percentage error $E[\epsilon_{out}]$ for (a) the strings metric space with the minimum edit distance, (b) the images metric space with Euclidean distance

chosen from the *Linux Dictionary*,[1] using the round size $2t = 4$ for Multiple-Winners variant and $t = 3$ for Single-Winner version. In Fig. 6-(b) the input set of $n = 10^i$, with $i = 2, 3, 4$, is randomly chosen from the *Corel* images database [2]. Each image has been characterized by its colors histograms, represented in the Euclidean metric space $\mathbb{R}^{32}$.

## 5 Conclusion

$O(n \log n)$ extensions of linear randomized tournaments techniques to better approximate solutions of optimization problems in metric spaces have been presented. The proposed extensions replace Single-Winner rounds with Multiple-Winners ones. Using a logarithmic consensus strategy further improves precision. Applications to 1-median and diameter computation have been considered. Experiments on both real and synthetic data showed that such newly proposed versions significantly outperform the Single-Winner strategy.

[1] The dictionary is contained in the text file /usr/share/dict/linux.words, under the Linux Mandrake v.8.1.

## References


1. V. Auletta, D. Parente, and G. Persiano. Dynamic and static algorithms for optimal placement of resources in a tree. *Theoretical Computer Science*, 165:441–461, 1996.
2. M. Ortega Binderberger. Corel images database. *UCI Knowledge Discovery in Databases Archive, URL http://kdd.ics.uci.edu/*, 1999.
3. A. Borodin, R. Ostrovsky, and Y. Rabani. Subquadratic approximation algorithms for clustering problems in high dimensional spaces. *Ann. ACM Symp. Theory of Computing*, pages 435–444, 1999.
4. T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transaction on Database Systems*, 24(3):361–404, 1999.
5. R. E. Burkard and J. Krarup. A linear algorithm for the pos/neg-weighted 1-median problem on a cactus. *Computing*, 60(3):193–216, 1998.
6. D. Cantone, G. Cincotti, A. Ferro, and A. Pulvirenti. An efficient approximate algorithm for the 1-median problem in metric spaces. *SIAM Journal on Optimization*, 2005. To appear.
7. D. Cantone, A. Ferro, A. Pulvirenti, D. Reforgiato, and D. Shasha. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *IEEE Transaction on knowledge and Data Engineering*, 17(4), 2005.
8. C. Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers Group, The Netherlands, 1996.
9. W.B. Frakes and R. Baeza-Yates. *Information Retrieval - Data Structures and Algorithms*. Prentice Hall, New Jersey, 1992.
10. Greg N. Frederickson. Parametric search and locating supply centers in trees. *F. Dehne and J.-R. Sack and N. Santoro, Editors, Algorithms and Data Structures, 2Nd Workshop WADS '91, Volume 519 of Lecture Notes in Computer Science*, pages 299–319, 1991.
11. K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, New York, 1990.
12. V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French. Clustering large datasets in arbitrary metric spaces. *Proceedings of the IEEE 15th International Conference on Data Engineering*, pages 502–511, 1999.
13. A. Goel, P. Indyk, and K.Varadarajan. Reductions among high dimensional proximity problems. *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 769–778, 2001.
14. T.F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.
15. E. Gusfield. Efficient methods for multiple sequence alignments with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993.
16. P. Indyk. Sublinear time algorithms for metric space problems. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 428–434, 1999.
17. P. Indyk. Dimensionality reduction techniques for proximity problems. *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 371–378, 2000.
18. Motwani R. and Raghavan P. *Randomized Algorithms*. Cambridge University Press, 2000.
19. Swift S., Tucker A., Vinciotti V., Martin N., Orengo C., Liu X., and Kellam P. Consensus clustering and functional interpretation of gene-expression data. *Genome Biology*, 5(11), 2004.

# On Symbolic Scheduling Independent Tasks with Restricted Execution Times

Daniel Sawitzki*

University of Dortmund, Computer Science 2
D-44221 Dortmund, Germany
daniel.sawitzki@cs.uni-dortmund.de

**Abstract.** Ordered Binary Decision Diagrams (OBDDs) are a data structure for Boolean functions which supports many useful operations. It finds applications in CAD, model checking, and symbolic graph algorithms. We present an application of OBDDs to the problem of scheduling $N$ independent tasks with $k$ different execution times on $m$ identical parallel machines while minimizing the over-all finishing time. In fact, we consider the decision problem if there is a schedule with makespan $D$. Leung's dynamic programming algorithm solves this problem in time $\mathcal{O}\left(\log m \cdot N^{2(k-1)}\right)$. In this paper, a symbolic version of Leung's algorithm is presented which uses OBDDs to represent the dynamic programming table $T$. This heuristical approach solves the scheduling problem by executing $\mathcal{O}(k \log m \log(mD))$ operations on OBDDs and is expected to use less time and space than Leung's algorithm if $T$ is large but well-structured. The only known upper bound of $\mathcal{O}\left((m \cdot D)^{3k+2}\right)$ on its resource usage is trivial. Therefore, we report on experimental studies in which the symbolic method was applied to random scheduling problem instances.

## 1 Introduction

The problem of nonpreemptively scheduling $N$ independent tasks with integral execution times on $m$ identical and parallel machines while minimizing the over-all finishing time (the *makespan*) is one of the most fundamental and well-studied problems of deterministic scheduling theory. It is known to be NP-hard in the strong sense [5]. In this paper, we consider the restricted case that the tasks have only a constant number $k$ of different execution times. Moreover, we are interested in the decision problem if there is a schedule with makespan not larger than $D$. This restricted problem is simply referred to as *scheduling problem* throughout this paper.

**Definition 1 (Scheduling Problem).** *A* scheduling problem $\mathcal{P}$ *consists of* $k$ execution times $t_1, \ldots, t_k \in \mathbb{N}$, *corresponding* demands $N_1, \ldots, N_k \in \mathbb{N}$, *a*

* Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research Cluster "Algorithms on Large and Complex Networks" (1126).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 277–289, 2005.
© Springer-Verlag Berlin Heidelberg 2005

*number $m$ of machines, and a* makespan bound $D$. *The* over-all demand *of $\mathcal{P}$ is defined by $N := \sum_{1 \le i \le k} N_i$.*

*A* schedule $\mathcal{S}\colon \{1,\dots,k\} \times \{1,\dots,m\} \to \mathbb{N}$ *for a scheduling problem $\mathcal{P}$ is called* valid *if $\sum_{j=1}^{m} \mathcal{S}(i,j) \ge N_i$ for every $i \in \{1,\dots,k\}$ and $\sum_{i=1}^{k} t_i \cdot \mathcal{S}(i,j) \le D$ for every $j \in \{1,\dots,m\}$.*

*A* scheduling algorithm *has to decide if there is a valid schedule $\mathcal{S}$ for $\mathcal{P}$.*

Leung [13] presents a scheduling algorithm with time $\mathcal{O}\big(\log m \cdot N^{2(k-1)}\big)$ and space $\mathcal{O}\big(\log m \cdot N^{(k-1)}\big)$. Following a dynamic programming approach, it computes a table $T$ of $\mathcal{O}\big(\log m \cdot N^{k-1}\big)$ partial solution values. The author considers the algorithm as polynomial for constant $k$ because the input size of general scheduling problems is $\Omega(N)$. However, due to the restriction to $k$ different execution times, the input can be represented by $2k+2$ numbers of length $\log(mD)$.

The idea behind the symbolic scheduling algorithm presented in this paper is to use *Ordered Binary Decision Diagrams (OBDDs)* [3, 4, 22] to represent the dynamic programming table $T$. OBDDs are a data structure for Boolean functions offering efficient functional operations, which is well-established in many areas like CAD, model checking [8, 14], and symbolic graph algorithms [9, 18, 17, 20, 23]. It is known to be a compact representation for structured and regular functions and allows to compute many table entries in parallel by few operations applied to the corresponding OBDDs. On the one hand, we expect this approach to require essentially less space than Leung's method; on the other hand, this implies also less runtime, because the efficiency of OBDD operations depends on the size of their operands.

In order to analyze the behavior of symbolic OBDD-based heuristics, we have to analyze the OBDD size of all Boolean functions occurring during their execution. This is known to be a difficult task in general and has been done only in a few pioneer works so far [18, 20, 21, 23]. So in most papers the usability of symbolic algorithms is just proved by experiments on benchmark inputs from special application areas [9, 10, 12, 15, 24]. In other works considering more general graph problems, mostly the number of OBDD operations (often referred to as "symbolic steps") is bounded as a hint on the actual runtime [2, 6, 7, 16].

To evaluate the usefulness of the presented scheduling method, it has been implemented and applied to random input instances for $k = 3$ due to three popular distributions of execution times. On these instances, the symbolic algorithm was observed to beat Leung's scheduling algorithm w. r. t. time and space if the product $P := \Pi_{j=1}^{k-1} N_j$ of task quantities is sufficiently large.

The paper is organized as follows: Section 2 introduces OBDDs and the operations offered by them. Then, Sect. 3 gives some preliminaries on symbolic algorithms and their notation. After a brief description of Leung's algorithm in Sect. 4, we present the symbolic scheduling method in Sect. 5. The experiments' setting and results are documented in Sects. 6 and 7. Finally, Sect. 8 gives conclusions on the work.

## 2 Ordered Binary Decision Diagrams (OBDDs)

We denote the class of Boolean functions $f\colon \{0,1\}^n \to \{0,1\}$ by $B_n$. The $i$th character of a binary number $x \in \{0,1\}^n$ is denoted by $x_i$ and $|x| := \sum_{i=0}^{n-1} x_i 2^i$ identifies its value.

A Boolean function $f \in B_n$ defined on variables $x_0, \ldots, x_{n-1}$ can be represented by an *Ordered Binary Decision Diagram (OBDD)* [3, 4]. An OBDD $\mathcal{G}$ is a directed acyclic graph consisting of *internal nodes* and *sink nodes*. Each internal node is labeled with a Boolean variable $x_i$, while each sink node is labeled with a Boolean constant. Each internal node is left by two edges one labeled by 0 and the other by 1. A *function pointer* $p$ marks a special node that represents $f$. Moreover, a permutation $\pi \in \Sigma_n$ called *variable order* must be respected by the internal nodes' labels on every path from $p$ to a sink. For a given variable assignment $a \in \{0,1\}^n$, we compute the function value $f(a)$ by traversing $\mathcal{G}$ from $p$ to a sink labeled with $f(a)$ while leaving a node $x_i$ via its $a_i$-edge.

An OBDD with variable order $\pi$ is called $\pi$-OBDD. The minimal-size $\pi$-OBDD $\mathcal{G}$ for a function $f \in B_n$ is known to be canonical. Its size size($\mathcal{G}$) is measured by the number of its nodes and will be denoted by $\pi\mathcal{G}[f]$. We adopt the usual assumption that all OBDDs occurring in symbolic algorithms have minimal size, since all essential OBDD operations produce minimized diagrams. On the other hand, finding an optimal variable order leading to the minimum size OBDD for a given function is known to be NP-hard. There is an upper bound of $\big(2 + o(1)\big)2^n/n$ for the OBDD size of every $f \in B_n$.

The satisfiability of $f$ can be decided in time $\mathcal{O}(1)$. The negation $\overline{f}$ as well as the replacement of a function variable $x_i$ by a constant $a_i$ (i. e., $f_{|x_i=a_i}$) is obtained in time $\mathcal{O}\big(\text{size}(\pi\mathcal{G}[f])\big)$ without enlarging the OBDD. Whether two functions $f$ and $g$ are equivalent (i. e., $f = g$) can be decided in time $\mathcal{O}\big(\text{size}(\pi\mathcal{G}[f]) + \text{size}(\pi\mathcal{G}[g])\big)$. These operations are called *cheap*. Further essential operations are the *binary synthesis* $f \otimes g$ for $f, g \in B_n$, $\otimes \in B_2$ (e. g., "$\wedge$" or "$\vee$"), and the *quantification* $(\mathcal{Q}x_i)f$ for a quantifier $\mathcal{Q} \in \{\exists, \forall\}$. In general, the result $\pi\mathcal{G}[f \otimes g]$ has size $\mathcal{O}\big(\text{size}(\pi\mathcal{G}[f]) \cdot \text{size}(\pi\mathcal{G}[g])\big)$, which is also the general runtime of this operation. The computation of $\pi\mathcal{G}\big[(\mathcal{Q}x_i)f\big]$ can be realized by two cheap operations and one binary synthesis in time and space $\mathcal{O}\big(\text{size}^2(\pi\mathcal{G}[f])\big)$.

The book of Wegener [22] gives a comprehensive survey on different types of binary decision diagrams.

## 3 Preliminaries on Symbolic Algorithms

The functions used for symbolic representations are typically defined on a number of $m$ subsets of Boolean variables, each having a certain interpretation within the algorithm. We assume w. l. o. g. that all arguments consist of the same number of $n$ Boolean variables. If there is no confusion, both a function $f \in B_{mn}$ defined on $x^{(1)}, \ldots, x^{(m)} \in \{0,1\}^n$ as well as its OBDD representation $\pi\mathcal{G}[f]$

will be denoted by $f$ in this paper. Quantifications $\left(\mathcal{Q}x_0^{(i)}, \ldots, x_{n-1}^{(i)}\right)$ over all $n$ variables of argument $i$ will be denoted by $\left(\mathcal{Q}x^{(i)}\right)$.

**Argument reordering.** Assume that each of the $m$ function arguments $x^{(1)}, \ldots, x^{(m)} \in \{0,1\}^n$ has its own variable order $\tau_i \in \Sigma_n$. The global order $\pi \in \Sigma_{mn}$ is called *$m$-interleaved* if it respects each $\tau_i$ while reading variables $x_j^{(i)}$ with same bit index $j$ en bloc, that is, $\pi := \left(x_{\tau_1(0)}^{(1)}, x_{\tau_2(0)}^{(2)}, \ldots, x_{\tau_m(0)}^{(m)}, x_{\tau_1(1)}^{(1)}, \ldots, x_{\tau_m(n-1)}^{(m)}\right)$.

Let $\rho \in \Sigma_m$ and $f \in B_{mn}$ be defined on variables $x^{(1)}, \ldots, x^{(m)} \in \{0,1\}^n$. A function $g \in B_{mn}$ is called the *argument reordering* of $f$ w. r. t. $\rho$ if $g\left(x^{(1)}, \ldots, x^{(m)}\right) = f\left(x^{(\rho(1))}, \ldots, x^{(\rho(m))}\right)$. Computing argument reorderings is an important operation of symbolic algorithms and is possible in linear time and space $\mathcal{O}(n)$ if an $m$-interleaved variable order is used and $m$ is constant (see [19]).

**Multivariate threshold and modulo functions.** The symbolic scheduling algorithm contains comparisons of weighted sums with threshold values like $f(x,y,z) := (a \cdot |x| + b \cdot |y| \geq T)$, $a, b, T \in \mathbb{Z}$, which can be realized by *multivariate threshold functions.*

**Definition 2 (Woelfel [23]).** *Let $f \in B_{mn}$ be defined on variables $x^{(1)}, \ldots, x^{(m)} \in \{0,1\}^n$. Then, $f$ is called* $m$-variate threshold function *iff there are $W \in \mathbb{N}$, $T \in \mathbb{Z}$, and $w_1, \ldots, w_m \in \{-W, \ldots, W\}$ such that*

$$f\left(x^{(1)}, \ldots, x^{(m)}\right) = \left(\sum_{i=1}^{m} w_i \cdot \left|x^{(i)}\right| \geq T\right) \quad .$$

Clearly, the relations $>$, $\leq$, $<$, and $=$ can be composed of multivariate threshold functions. For constant $W$ and $m$, such comparisons have $\pi$-OBDDs of size $\mathcal{O}(n)$ for an $m$-interleaved variable order $\pi$ with increasing bit significance (i. e., $\tau_i = \mathrm{id}$) [23]. These OBDDs can be computed efficiently in linear time.

Moreover, the symbolic scheduling algorithm makes use of *multivariate modulo functions.*

**Definition 3 (Woelfel [23]).** *Let $f \in B_{mn}$ be defined on variables $x^{(1)}, \ldots, x^{(m)} \in \{0,1\}^n$. Then, $f$ is called* $m$-variate modulo function *iff there are $M \in \mathbb{N}$, $T \in \mathbb{Z}$, and $w_1, \ldots, w_m \in \mathbb{Z}$ such that*

$$f\left(x^{(1)}, \ldots, x^{(m)}\right) = \left(\sum_{i=1}^{m} w_i \cdot \left|x^{(i)}\right| \bmod M = T\right) \quad .$$

For constant $M$ and $m$, $m$-variate modulo functions have $\pi$-OBDDs of size $\mathcal{O}(n)$ using an $m$-interleaved variable order $\pi$ with increasing bit significance (i. e., $\tau_i = \mathrm{id}$) [23]. These OBDDs can be computed efficiently in linear time.

We conclude that all essential functional operations are realized efficiently by OBDDs w. r. t. the corresponding OBDD size if an interleaved variable order is

used. Therefore, this property is assumed in the following. This is also crucial for threshold and modulo functions to have compact OBDDs, which will be the building blocks of all Boolean functions computed by the symbolic scheduling algorithm.

## 4 The Scheduling Algorithm of Leung

Let $\mathcal{P}$ be a scheduling problem according to Def. 1. and assume that $\log_2(m) \in \mathbb{N}$. Leung's algorithm [13] computes a $k$-dimensional table $T$ with entries $T(\ell, i_1, \ldots, i_{k-1})$ for $\ell = 0, \ldots, \log_2 m$, $i_j = 0, \ldots, N_j$, and $j = 1, \ldots, k-1$. Such an entry contains the maximum number $I$ of tasks of type $k$ that can be scheduled onto $2^\ell$ machines together with $i_j$ tasks of type $j$ for all types $j = 1, \ldots, k-1$.

We define upper bounds $B_j := \min\{N_j, \lfloor D/t_j \rfloor\}$, $j = 1, \ldots, k$, for the maximum number of tasks of type $j$ that can be scheduled onto one machine. Let us consider the case $\ell = 0$: If $0 \le i_j \le B_j$ for $j = 1, \ldots, k-1$ and $D \ge \sum_{j=1}^{k-1} t_j \cdot i_j$, it is $I = \left\lfloor \left( D - \sum_{j=1}^{k-1} t_j \cdot i_j \right) / t_k \right\rfloor$; else, we define $I := -1$.

Having computed all $\Pi_{j=1}^{k-1} (N_j + 1)$ entries $T(\ell, i_1, \ldots, i_{k-1})$ for some machine count $2^\ell$, the entries for $2^{\ell+1}$ machines are obtained by

$$\begin{aligned} T(\ell+1, i_1, \ldots, i_{k-1}) := \max \big\{ -1,\ & T(\ell, i'_1, \ldots, i'_{k-1}) + T(\ell, i''_1, \ldots, i''_{k-1}) \\ & \mid \forall j \in \{1, \ldots, k-1\} \colon i'_j, i''_j \in \{0, \ldots, N_j\},\ i_j = i'_j + i''_j, \\ & T(\ell, i'_1, \ldots, i'_{k-1}) \ne -1 \ne T(\ell, i''_1, \ldots, i''_{k-1}) \big\} \quad . \end{aligned} \tag{1}$$

This procedure can be easily modified to cope with values $m$ that are not powers of 2. Finally, there is a valid schedule for $\mathcal{P}$ if and only if $T(\log_2 m, N_1, \ldots, N_{k-1}) \ge N_k$. Altogether, $\mathcal{O}\big(\log m \cdot \Pi_{j=1}^{k-1} (N_j + 1)\big) = \mathcal{O}\big(\log m \cdot N^{k-1}\big)$ table entries are computed, each one as maximum over $\mathcal{O}\big(\Pi_{j=1}^{k-1} (N_j + 1)\big) = \mathcal{O}\big(N^{k-1}\big)$ vectors $\big(i'_1, \ldots, i'_{k-1}\big)$ implying the runtime complexity $\mathcal{O}\big(\log m \cdot N^{2(k-1)}\big)$.

The minimal makespan can be found by a binary search using $\mathcal{O}(\log \max\{t_1, \ldots, t_k\})$ executions of Leung's algorithm (see [13]). By storing the optimal partition vector $\big(i'_1, \ldots, i'_{k-1}\big)$ for each table entry, the algorithm can easily be extended to compute an optimal schedule if one exists.

## 5 The Symbolic Scheduling Algorithm

We again assume that $\log_2(m) \in \mathbb{N}$. Moreover, it is reasonable to require $t_i \le D$ and $N_i \le mD$ for $i = 1, \ldots, k$. Then, $\lceil \log_2(mD + 1) \rceil =: n$ Boolean variables suffice to represent the number arguments of all Boolean functions occurring during the algorithm.

The symbolic scheduling algorithm works with *characteristic* Boolean functions $\chi_{T,\ell} \in B_{kn}$ of Leung's dynamic programming table $T$ defined by

$$\chi_{T,\ell}\Big(x^{(1)},\ldots,x^{(k)}\Big) = 1 :\Leftrightarrow T\Big(\ell, \Big|x^{(1)}\Big|,\ldots,\Big|x^{(k-1)}\Big|\Big) = \Big|x^{(k)}\Big|$$

for $\ell = 0,\ldots,\log_2 m$ and vectors $x^{(j)} \in \{0,1\}^n$, $j = 1,\ldots,k$. The binary value of $x^{(j)}$ corresponds to the number $i_j$ of tasks in Sect. 4. Because $\big|x^{(k)}\big|$ is non-negative, $\chi_{T,\ell}$ is false for table entries $-1$.

In order to compute the initial function $\chi_{T,0}$, we express the conditions for $\ell = 0$ stated in Sect. 4 in terms of Boolean equations using multivariate threshold and modulo functions as building blocks. At first, we state a function $g$ for the condition $\big|x^{(k)}\big| = \Big\lfloor\Big(D - \sum_{j=1}^{k-1} t_j \cdot \big|x^{(j)}\big|\Big)/t_k\Big\rfloor$, which is equivalent to

$$D - \sum_{j=1}^{k-1} t_j \cdot \Big|x^{(j)}\Big| = \Big|x^{(k)}\Big| \cdot t_k + \left(D - \sum_{j=1}^{k-1} t_j \cdot \Big|x^{(j)}\Big|\right) \bmod t_k \ .$$

This leads to the following symbolic formulation for $g$ which can be computed by subsequent applications of OBDD operations starting with multivariate threshold and modulo functions. It uses two vectors $y, z \in \{0,1\}^n$ of intermediate helping variables.

$$g\Big(x^{(1)},\ldots,x^{(k)}\Big) := (\exists y, z)\left[\left(|y| = D - \sum_{j=1}^{k-1} t_j \cdot \Big|x^{(j)}\Big|\right)\right.$$
$$\left.\wedge\,(|z| < t_k) \wedge (|y| - |z| \bmod t_k = 0) \wedge \Big(|y| = \Big|x^{(k)}\Big| \cdot t_k + |z|\Big)\right]$$

Altogether, the initial function $\chi_{T,0}$ is obtained by

$$\chi_{T,0}\Big(x^{(1)},\ldots,x^{(k)}\Big) := \bigwedge_{j=1}^{k-1}\Big(\Big|x^{(j)}\Big| \le B_j\Big)$$
$$\wedge\left(D \ge \sum_{j=1}^{k-1} t_j \cdot \Big|x^{(j)}\Big|\right) \wedge g\Big(x^{(1)},\ldots,x^{(k)}\Big) \ .$$

At next, the iterative step (1) is realized in terms of OBDD operations using $2k+1$ vectors $y, u^{(1)}, v^{(1)}, \ldots, u^{(k)}, v^{(k)} \in \{0,1\}^n$ of intermediate helping variables. Assume that $\chi_{T,\ell}$ has already been computed for some $\ell \in \{0,\ldots,\log_2 m - 1\}$. We define $h_{\ell+1} \in B_{k,n}$ as

$$h_{\ell+1}\Big(x^{(1)},\ldots,x^{(k)}\Big) := \Big(\exists u^{(1)},v^{(1)},\ldots,u^{(k-1)},v^{(k-1)}\Big)$$
$$\left[\bigwedge_{j=1}^{k-1}\Big(\Big|x^{(j)}\Big| = \Big|u^{(j)}\Big| + \Big|v^{(j)}\Big|\Big) \wedge \chi_{T,\ell}\Big(u^{(1)},\ldots,u^{(k)}\Big) \wedge \chi_{T,\ell}\Big(v^{(1)},\ldots,v^{(k)}\Big)\right] .$$

That is, $h_{\ell+1}$ represents load vectors $\big(\big|x^{(1)}\big|,\ldots,\big|x^{(k)}\big|\big)$ that can be partitioned into loads $\big(\big|u^{(1)}\big|,\ldots,\big|u^{(k)}\big|\big)$ and $\big(\big|v^{(1)}\big|,\ldots,\big|v^{(k)}\big|\big)$ each fitting onto $2^\ell$ machines while respecting makespan bound $D$.

Finally, we have to guarantee the maximality of the number of type-$k$-tasks:

$$\chi_{T,\ell+1}\Big(x^{(1)},\ldots,x^{(k)}\Big) := h_{\ell+1}\Big(x^{(1)},\ldots,x^{(k)}\Big) \wedge \overline{(\exists y)\left[\big(|y| > \big|x^{(k)}\big|\big) \wedge h_{\ell+1}\big(x^{(1)},\ldots,x^{(k-1)},y\big)\right]} .$$

That is, there is no number $|y|$ of type-$k$-tasks greater than $\big|x^{(k)}\big|$ that can also be distributed onto $2^{\ell+1}$ machines according to $h_{\ell+1}$.

Having computed $\chi_{T,\log_2 m}$ this way, we replace each variable vector $x^{(j)}$ by the binary representation of $N_j$ for $j = 1,\ldots,k-1$. Then, the unique satisfying assignment of the remaining variables $x^{(k)}$ correspond to $T(\log_2 m, N_1,\ldots,N_{k-1})$ which is compared to $N_k$. The correctness follows from the correctness of Leung's algorithm. We have solved scheduling problem $\mathcal{P}$ following Leung's approach by using a symbolic OBDD representation for the dynamic programming table $T$.

Similar to Leung's algorithm, the symbolic scheduling methods can be easily modified to handle arbitrary numbers $m$ of machines as well as to compute concrete schedule $\mathcal{S}$.

**Theorem 1.** *The symbolic scheduling algorithm solves a scheduling problem $\mathcal{P}$ with task execution times $t_1,\ldots,t_k$, task demands $N_1,\ldots,N_k$, machine count $m$, and makespan bound $D$ by executing $\mathcal{O}(k\log m\log(mD))$ operations on OBDDs defined on $(3k+2)n$ Boolean variables with $n := \lceil\log_2(mD+1)\rceil$.*

*Proof.* We compute $\log_2 m + 1$ Boolean functions $\chi_{T,\ell}$ with $\ell = 0,\ldots,\log_2 m$. Each function occurring during the algorithm is defined on no more than $(3k+2)n$ variables and computed by a constant number of binary syntheses and quantifications over variable vectors of length $n$. Altogether, each $\chi_{T,\ell}$ takes $\mathcal{O}(k\log(mD))$ OBDD operations. □

The upper bound of $\big(2+o(1)\big)2^n/n$ for the OBDD size of every $f \in B_n$ implies a maximum OBDD size of $\mathcal{O}\big((mD)^{3k+2}\big)$. Hence, each OBDD operation takes time $\mathcal{O}\big((mD)^{6k+4}\big)$ in the worst case. After having computed $\chi_{T,\ell+1}$ we may discard $\chi_{T,\ell}$. Therefore, $\mathcal{O}\big((mD)^{3k+2}\big)$ is also an upper bound on the overall space usage. Of course, these theoretical bounds cannot compete with the complexity of Leung's algorithm.

Nevertheless, we hope that heuristical methods like the symbolic scheduling algorithm perform much better than in the worst case when applied on practical

problem instances or in the average case. Then, they are expected to beat known algorithms with better worst-case behavior. Hence, we applied the presented algorithm to randomly generated instances hoping that structures and regularities of table $T$ lead to compact OBDDs for the functions $\chi_{T,\ell}$ and, therefore, to an efficient over-all time and space usage.

## 6 Experimental Setting

The symbolic scheduling algorithm was implemented[1] in C++ using the gcc 2.95.3 and the OBDD package CUDD 2.3.1 by Fabio Somenzi.[2] Initially, an interleaved variable order with increasing bit significance is used for the Boolean variables of each function argument. After quantification operations, the actual variable order $\pi$ is heuristically optimized by permuting three adjacent variables while keeping $\pi$ interleaved. This is iterated until a local optimum is reached (see [11]).

The scheduling problem instances $\mathcal{P}$ generated for the experiments have a load sum $L := \sum_{j=1}^{k} t_j \cdot N_j$ with mean $E[L] = M := (mD)/1.2$. That is, the effective capacity $mD$ is 20% larger than the expected load. This is achieved by first drawing a uniformly distributed fraction $F_j$ of $M$ such that $\sum_{j=1}^{k} F_j = M$ for each task type $j = 1, \ldots, k$. Then, the number $N_j$ of tasks of each type $j$ is drawn uniformly due to a mean parameter $E[N_j]$. Finally, the execution times $t_j$ are drawn due to the uniform, exponential, or Erlang distribution (shape parameter 2) with mean $E[t_j] := F_j/N_j$, which are common distributions in modeling synthetic scheduling instances (see, e. g. [1]).

$$E[L] = \sum_{j=1}^{k} E[t_j \cdot N_j] = \sum_{j=1}^{k} E[t_j] \cdot N_j = \sum_{j=1}^{k} F_j = M$$

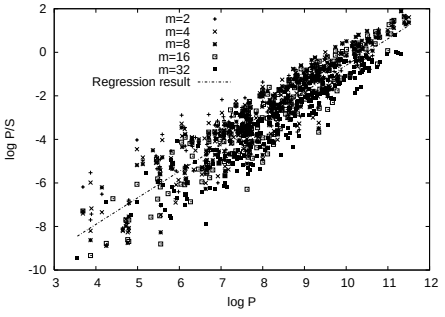
This random procedure has parameters $m$, $D$, and $E[N_j]$ for $j = 1, \ldots, k$. The values $t_j$ and $N_j$ are rounded randomly to integers. Moreover, only those instances are accepted that fulfill the conditions $t_j \leq D$, $N_j \leq mD$, and $t_j \neq t_{j'}$ for $1 \leq j < j' \leq k$.

The experiments consist of three series with $mD = 800, 1600, 3200$ and $k = 3$. Within each series, $m$ was chosen to be 2, 4, 8, 16, and 32. For each setting, 20 instances have been generated due to the three execution time distribution mentioned above. Moreover, 10 different values of $E[N_j]$ between $(mD)/6$ to $(mD)/3$ have been used per series ($E[N_1] = \cdots = E[N_k]$).

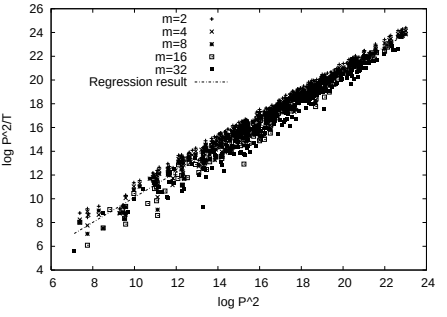
The experiments took place on a PC with Pentium 4 3GHz processor and 1 GB of main memory running Linux 2.4.21. The runtime has been measured by seconds of process time, while the space usage is given as the maximum number

[1] Implementation and experimental data are available at `http://thefigaro.sourceforge.net/`.

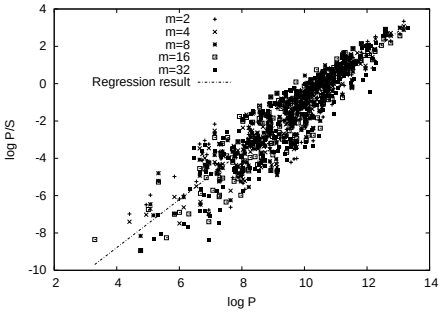
[2] CUDD is available at `http://vlsi.colorado.edu/`.



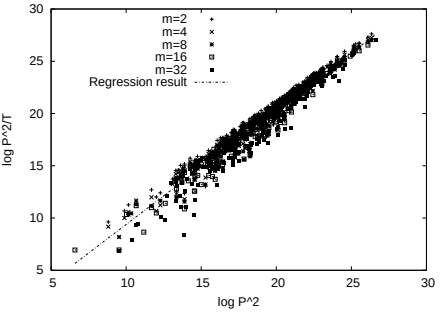

(a) Space comparison for $mD = 800$



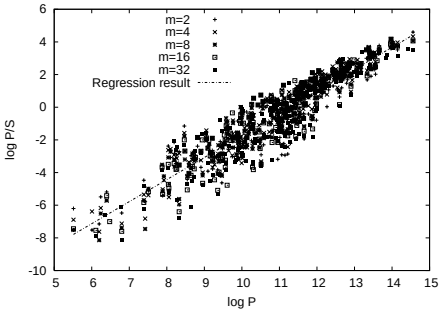

(b) Time comparison for $mD = 800$



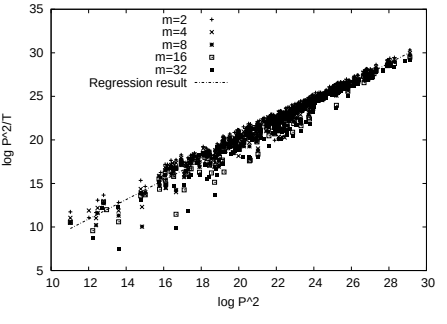

(c) Space comparison for $mD = 1600$




(d) Time comparison for $mD = 1600$




(e) Space comparison for $mD = 3200$




(f) Time comparison for $mD = 3200$

**Fig. 1.** Experimental results on random instances with exponentially distributed execution times. $P$ denotes $\Pi_{j=1}^{k-1} N_j$, $S$ denotes the symbolic algorithm's space, $T$ denotes the symbolic algorithm's time

of OBDD nodes present at any time during an algorithm execution. The latter is of same magnitude as the over-all space usage and independent of the used computer system.

## 7 Experimental Results

In order to compare Leung's algorithm to the symbolic approach, we have to take a closer look at the resources used by the former one. Because we exclusively consider values $m$ with $\log_2(m) \in \mathbb{N}$, each subtable $T(\ell, \dots)$ due to some machine count can be discarded after having computed the subtable for $\ell + 1$ machines. Moreover, we assume that $N_k = \min\{N_1, \dots, N_k\}$. Then, Leung's algorithm needs space $\Omega\big(\Pi_{j=1}^{k-1} N_j\big)$ and time $\Omega\big(\Pi_{j=1}^{k-1} N_j^2\big)$.

We consider the experimental results for exponentially distributed execution times: Figures 1(a), 1(c), and 1(e) show plots of the ratios $P/S$ for $P := \Pi_{j=1}^{k-1} N_j$ and the symbolic algorithm's space usage $S$ for growing $P$ using logarithmic scales. Analogue, Figs. 1(b), 1(d), and 1(f) show plots of the ratios $P^2/T$ and the symbolic algorithm's time usage $T$ for growing $P^2$. We observe the symbolic method to beat Leung's algorithm w. r. t. both time and space for inputs with high task demand products $P$.

Concretely, the presented plots for exponentially distributed execution times as well as the omitted plots for the other two distributions hint to a linear dependence of $\log(P/S)$ of $\log P$ resp. $\log\big(P^2/T\big)$ of $\log P^2$. Therefore, a least squares method has been used to fit parameters $a_1$ and $b_1$ for $\log(P/S) = a_1 \cdot \log P + b_1$ resp. $a_2$ and $b_2$ for $\log(P^2/T) = a_2 \cdot \log P^2 + b_2$. Table 1 shows the fitting results for all three values of $mD$ and the three considered distributions. The gradients' $a_1$ and $a_2$ asymptotic standard errors never exceed 1.4%.

**Table 1.** Fits of $a_1$ and $b_1$ for $P/S$ (Tab. 2(a)) resp. $a_2$ and $b_2$ for $P^2/T$ (Tab. 2(b))

| Distribution / $mD$ | 800 | 1600 | 3200 |
|---|---|---|---|
| Uniform | 1.28270 / -13.21930 | 1.32832 / -14.21380 | 1.36477 / -15.22760 |
| Exponential | 1.21732 / -12.77160 | 1.28701 / -13.93220 | 1.34987 / -15.22830 |
| Erlang | 1.26583 / -13.11690 | 1.37385 / -14.63660 | 1.44698 / -16.14440 |

(a) Fits for the symbolic algorithm's space usage

| Distribution / $mD$ | 800 | 1600 | 3200 |
|---|---|---|---|
| Uniform | 1.06093 / -0.47254 | 1.08898 / -1.35841 | 1.12353 / -2.51579 |
| Exponential | 1.05560 / -0.43123 | 1.09418 / -1.55419 | 1.11825 / -2.50760 |
| Erlang | 1.06234 / -0.52043 | 1.11002 / -1.82228 | 1.15972 / -3.31591 |

(b) Fits for the symbolic algorithms time usage

The Erlang distribution seems to result in slightly higher absolute values $a$ and $|b|$.

The hypothesed linear dependencies imply $P/S = P^{a_1} \cdot 2^{b_1} \Leftrightarrow S = P^{1-a_1} \cdot 2^{-b_1}$ resp. $T = P^{2(1-a_2)} \cdot 2^{-b_2}$. Due to $1 < a_1, a_2 < 1.5$ (see Tab. 1), we conclude the symbolic scheduling algorithm to have space usage $2^{-b_1}/\sqrt[c_1]{P}$ for $c_1 = 1/(a_1 - 1) > 2$ resp. time usage $2^{-b_2}/\sqrt[c_2]{P}$ for $c_2 = 1/(2a_2 - 2) > 1$.

That is, the $a$- and $b$-parameters seem to depend only on $mD$ and $k$ while being independent of $m$. For fixed $mD$ and $k$, the OBDD sizes shrink proportional to $\sqrt[c_1]{P}$ leading to essentially smaller time and space than Leung's algorithm. Although only experiments with $k = 3$ are documented, these results have been also observed for higher values $k$.

## 8 Conclusions

We presented a symbolic algorithm for the decision problem of scheduling independent tasks with restricted execution times. It solves the problem by performing $\mathcal{O}(k \log m \log(mD))$ OBDD operations, while its final runtime and space usage depends on the size of the OBDDs it generates. Therefore, it was applied to random scheduling instances whose execution times were generated due to the uniform, exponential, and Erlang distribution. On these instances, the symbolic algorithm was observed to beat Leung's scheduling algorithm w. r. t. time and space if the product $P := \Pi_{j=1}^{k-1} N_j$ is sufficiently large. For fixed $mD$ and $k$, the symbolic time and space usage is observed to grow as $\Theta\left(1/\sqrt[c]{P}\right)$ for some constant $c > 1$ depending on $mD$ and the measured quantity.

Hence, we consider the application of OBDDs to Leung's scheduling method as a useful way to compress its dynamic programming table, which succeeds in savings of runtime and space on inputs with large demand $N$. Future research could address experiments on real world instances as well as several heuristics like different strategies for OBDD variable reordering.

**Acknowledgments.** Thanks to Detlef Sieling, Ingo Wegener, and Berthold Vöcking for fruitful discussions.

## References

[1] S. Albers and B. Schröder. An experimental study of online scheduling algorithms. *Journal of Experimental Algorithms*, 7:3, 2002.

[2] R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer-Aided Design 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2000.

[3] R.E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In *Design Automation Conference 1985*, pages 688–694. ACM Press, 1985.

[4] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
[5] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
[6] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *Symposium on Discrete Algorithms 2003*, pages 573–582. ACM Press, 2003.
[7] R. Gentilini and A. Policriti. Biconnectivity on symbolically represented graphs: A linear solution. In *International Symposium on Algorithms and Computation 2003*, volume 2906 of *Lecture Notes in Computer Science*, pages 554–564. Springer, 2003.
[8] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.
[9] G.D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0–1 networks. *Formal Methods in System Design*, 10:207–219, 1997.
[10] R. Hojati, H. Touati, R.P. Kurshan, and R.K. Brayton. Efficient $\omega$-regular language containment. In *Computer-Aided Verification 1993*, volume 663 of *Lecture Notes in Computer Science*, pages 396–409. Springer, 1993.
[11] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *International Conference on Computer Aided Design 1991*, pages 472–475. IEEE Press, 1991.
[12] H. Jin, A. Kuehlmann, and F. Somenzi. Fine-grain conjunction scheduling for symbolic reachability analysis. In *Tools and Algorithms for the Construction and Analysis of Systems 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2002.
[13] J. Y.-T. Leung. On scheduling independent tasks with restricted execution times. *Operations Research*, 30(1):163–171, 1982.
[14] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1994.
[15] I. Moon, J.H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Design Automation Conference 2000*, pages 23–28. ACM Press, 2000.
[16] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Formal Methods in Computer-Aided Design 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2000.
[17] D. Sawitzki. Experimental studies of symbolic shortest-path algorithms. In *Experimental and Efficient Algorithms 2004*, volume 3059 of *Lecture Notes in Computer Science*, pages 482–497. Springer, 2004.
[18] D. Sawitzki. Implicit flow maximization by iterative squaring. In *SOFSEM 2004: Theory and Practice of Computer Science*, volume 2932 of *Lecture Notes in Computer Science*, pages 301–313. Springer, 2004.
[19] D. Sawitzki. On graphs with characteristic bounded-width functions. Technical report, Universität Dortmund, 2004.
[20] D. Sawitzki. A symbolic approach to the all-pairs shortest-paths problem. In *Graph-Theoretic Concepts in Computer Science 2004*, volume 3353 of *Lecture Notes in Computer Science*, pages 154–167. Springer, 2004.
[21] D. Sawitzki. Lower bounds on the OBDD size of graphs of some popular functions. In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2005.

[22] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, Philadelphia, 2000.

[23] P. Woelfel. Symbolic topological sorting with OBDDs. In *Mathematical Foundations of Computer Science 2003*, volume 2747 of *Lecture Notes in Computer Science*, pages 671–680. Springer, 2003.

[24] A. Xie and P.A. Beerel. Implicit enumeration of strongly connected components. In *International Conference on Computer Aided Design 1999*, pages 37–40. ACM Press, 1999.

# A Simple Randomized $k$-Local Election Algorithm for Local Computations

Rodrigue Ossamy

LaBRI
University of Bordeaux I
351 Cours de la Libération
33405 - Talence, France
ossamy@labri.fr

**Abstract.** Most of distributed algorithms encoded by means of local computations [3] need to solve $k-$local election problems to ensure a faithful relabeling of disjoint subgraphs. Due to a result stated in [1], it is not possible to solve the $k-$local election problem for $k \geq 3$ in anonymous networks. Based on distributed computations of rooted trees of minimal paths, we present in this paper a simple randomized algorithm which, with very high probability, solves the $k$-local election problem ($k \geq 2$) in an anonymous graph.

**Keywords:** *Local computations, election in graphs, distributed algorithms, randomized algorithms.*

## 1 Introduction

The problem of election is linked to distributed computations in a network. It aims to choose a unique vertex, called leader, which subsequently is used to make decisions or to centralize some information. For a fixed given positive integer $k$, a $k$-local election problem requires that, starting from a configuration where all processes are in the same state, the network reaches a configuration $\mathcal{C}$ such that for this configuration there exists a non empty set of vertices, denoted $\mathcal{E}$, satisfying:

- each vertex $v \in \mathcal{E}$ is in a special state called *leader* and
- $\forall v \in \mathcal{C}$ and for all vertex $w \neq v$ such that $d(v, w) \leq k$ then $w$ is in the state *lost* (i.e. $w \notin \mathcal{E}$).

We assume that each process has the same local algorithm. This problem is then considered under the following assumptions:

- the network is anonymous: unique identities are not available to distinguish the processes,
- the system is asynchronous,
- processes communicate by asynchronous message passing: there is no fixed upper bound on how long it takes for a message to be delivered,
- each process knows from which channel it receives a message.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 290–301, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Our goal is to perform $k$-local elections such that all the elected vertices should be able to execute graph relabeling steps on disjoint subgraphs of radius $\frac{k}{2}$. We consider a network of processors with arbitrary topology. It is represented as a connected, undirected graph where vertices denote processors and edges denote direct communication links. At every time, each vertex and each edge is in some particular state and this state will be encoded by a vertex or edge label. A distributed algorithm is encoded by means of local relabeling: labels attached to vertices and edges are modified locally, that is on a bounded subgraph of the given graph according to certain rules depending on the subgraph only. The relabeling is performed until no more transformation is possible.

For the sake of time complexity, we assume that each message incurs a delay of at most one unit of time [2]. Note that the delay assumption is only used to *estimate* the performance of our algorithms. This does not imply that our model is synchronous, neither does it affect the correctness of our algorithms. That is, our algorithms work correctly even in the absence of this delay assumption.

Here we first propose and analyze a randomized algorithm that solves the $k$-local election problem for $k \geq 2$. This algorithm is based on distributed computations of minimal paths rooted trees and works under the assumption that each vertex has an unique identity. Afterward we derive a second algorithm which solves the same problem without identities, with very high probability and with an acceptable time complexity of $O(k^2)$.

This paper is organized as follows. Section 2 reviews definitions of graphs and illustrates the notions of graph relabeling systems for distributed algorithms. Section 3 is devoted to the distributed computation of a rooted tree of minimal paths. In Section 4 we present and analyze our algorithms for solving the $k$-local election problem in an anonymous network. Section 5 concludes the paper.

## 2 Definitions and Notations

### 2.1 Undirected Graphs

We only consider finite, undirected and connected graphs without multiple edges and self-loops. If $G$ is a graph, then $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges; two vertices $u$ and $v$ are said to be adjacent if $\{u, v\}$ belongs to $E(G)$. The distance between two vertices $u, v$ is denoted $d(u, v)$. The set of neighbors of $v$ in $G$, denoted $N_G(v)$, is the set of all vertices of $G$ adjacent to $v$. The degree of a vertex $v$ is $|N_G(v)|$. Let $v$ be a vertex and $k$ a non negative integer, we denote by $B_G(v, k)$, or briefly $B(v, k)$, the centered ball of radius $k$ with center $v$, i.e., the subgraph of $G$ defined by the vertex set $V' = \{v' \in V(G) \mid d(v, v') \leq k\}$ and the edge set $E' = \{\{v_1, v_2\} \in E(G) \mid d(v, v_1) < k \text{ and } d(v, v_2) \leq k\}$. Throughout the rest of this paper we will consider graphs whose vertices and edges are labeled with labels from a recursive set $\mathcal{L}$. A graph labeled over $\mathcal{L}$ will be denoted by $(G, \lambda)$, where $G$ is a graph and $\lambda\colon V(G) \cup E(E) \to \mathcal{L}$ is the labeling function. The graph $G$ is called the underlying graph and the mapping $\lambda$ is a labeling of $G$.

### 2.2 Randomized 2-Local Elections

We present in this subsection the randomized procedure: $RL_2$ that solves the 2−local election problem. This procedure was introduced and analyzed in [6]. Let $K$ be a nonempty set equipped with a total order.

$RL_2$**:** Each vertex $v$ repeats the following actions. The vertex $v$ selects an integer $rand(v)$ randomly and uniformly from the set $K$. $v$ sends $rand(v)$ to its neighbors. $v$ receives messages from all its neighbors. Let $Int_w$ be the maximum of the set of integers that $v$ has received from vertices different from $w$. For all neighbors $w$, $v$ sends $Int_w$ to $w$. $v$ receives integers from all its neighbors. $v$ wins the 2−Election in $B(v, 2)$ if $rand(v)$ is strictly greater than all integers received by $v$.

**Fact 1.** *Let $G = (V, E)$ be a graph. After the execution of $RL_2$ in $G$ there are at most $\frac{|V|}{2}$ vertices $v$ of $G$ that have won the 2-local election.*

### 2.3 Graph Relabeling Systems for Encoding Distributed Computation

In this section, we illustrate, in an intuitive way, the notion of graph relabeling systems by showing how some algorithms on networks of processors may be encoded within this framework [4]. According to its own state and to the states of its neighbors (or a neighbor), each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbors and of the corresponding edges may have changed according to some specific *computation rules*. Graph relabeling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labeling of its components (edges and/or vertices), the final labeling being the result,
(C2) they are local, that is, each relabeling changes only a connected subgraph of a fixed size in the underlying graph,
(C3) they are locally generated, that is, the applicability condition of the relabeling only depends on the local context of the relabeled subgraph.

A formal definition of local computations can be found in [4]. Our goal is to explain the convention under which we will describe graph relabeling systems in this paper. If the number of rules is finite then we will describe all rules by their preconditions and relabellings. We will also describe a family of rules by a generic rule scheme ("meta-rule"). In this case, we will consider a generic stargraph of generic center $v_0$ and of generic set of vertices $B(v_0, 1)$. If $\lambda(v)$ is the label of $v$ in the precondition, then $\lambda'(v)$ will be its label in the relabeling. We will omit in the description labels that are not modified by the rule. This means that if $\lambda(v)$ is a label such that $\lambda'(v)$ is not explicitly described in the rule for a given $v$, then $\lambda'(v) = \lambda(v)$. The same definitions also hold for the relabeling of edges.

**An Example:**
The following relabeling system performs the election algorithm in the family of tree-shaped networks. The set of labels is $L = \{N, elected, non\text{-}elected\}$. The initial label on all vertices is $l_0 = N$ and there are two meta-rules that are described as follows.

$R1$ : **Pruning rule**

*Precondition :*
- $\lambda(v_0) = N$,
- $\exists!\ v \in B(v_0, 1), v \neq v_0, \lambda(v) = N$.

*Relabeling :*
- $\lambda'(v_0) :=$ *non-elected*.

$R2$ : **Election rule**

*Precondition :*
- $\lambda(v_0) = N$,
- $\forall v \in B(v_0, 1), v \neq v_0, \lambda(v) \neq N$.

*Relabeling :*
- $\lambda'(v_0) :=$ *elected*.

Let us call a pendant vertex any vertex labeled $N$ having exactly one neighbor with the label $N$. The meta-rule $R1$ consists in cutting a pendant vertex by giving it the label *non-elected*. The label $N$ of a vertex $v$ becomes *elected* by the meta-rule $R2$ if the vertex $v$ has no neighbor labeled $N$. A complete proof of this system may be found in [4].

## 3 Computation of a Rooted Tree of Minimal Paths

Let $G = (V, E)$ be an anonymous network with a distinguished vertex $U$. The problem considered here is to find a tree of $(G, V)$, rooted at $U$, which for any $v \in V$ contains a unique minimal path from $v$ to $U$. This kind of tree is generally used to pass a signal along the shortest path from $v$ to $U$ (see Moore [7]).

To solve the above problem, we can simply fan out from $U$, labeling each vertex with a number which counts its distance from $U$, modulo 3. Thus, $U$ is labeled 0, all unlabeled neighbors of $U$ are labeled 1, etc. More generally, at the $t$th step, where $t = 3m + q, m \in \mathbb{N}, q \in \{0, 1, 2\}$, we label all unlabeled neighbors of labeled vertices with $q$ and we mark the corresponding edges. When no more vertices can be labeled, the algorithm is terminated. It is then quite simple to show that the set of marked edges represents a tree of minimal paths rooted at $U$. In an asynchronous distributed system, where communication is due to a message passing system, we do not have any kind of centralized coordination. Thus, it is not easy for a labeled vertex to find out that all labeled vertices are in the same step $t$. To get around this problem, we have slightly modified and adapted the above procedure for distributed systems.

### 3.1 The Algorithm

Our algorithm works in rounds. All the vertices have knowledge about the computation round in which they are involved, they also have a state that indicates if they are *locked* or *unlocked*. At the end of round $i$, all vertices $v \in \{w \in V | d(w, U) \leq i\}$ are labeled and *locked*. Initially, $U$ is *unlocked* and labeled 0. At round 1, all unlabeled neighbors of $U$ are labeled 1 and *locked*. A vertex $w$ is said to be *marked* for a vertex $v$ if the edge $e = [w, v]$ is marked. For further computations the algorithm has to satisfy the following requirements:

$r_1$**:** Each time an unlabeled vertex is labeled, it is set in the *locked* state.
$r_2$**:** A labeled *unlocked* vertex $v \neq U$ becomes *locked* if:
- $v$ is in the same round as all its *marked* neighbors,
- $v$ does not have any unlabeled vertex in its neighborhood,
- All the *marked* neighbors $w$ of $v$ that satisfy $d(U, w) = d(U, v) + 1$ are *locked*.

$r_3$**:** A *locked* vertex $v$ in round $p$ becomes *unlocked* and increases its round, if it has an *unlocked marked* neighbor $w$ in round $p + 1$.

We encode this procedure by means of a graph relabeling system where the *locked* and *unlocked* states are respectively represented by the labels $F$ and $A$. The root is the only distinguished vertex labeled with $R$. Marked edges are labeled with 1. The set of labels is $L = \{0, 1, (x, d, r)\}$ with $x \in \{N, A, F, R\}$ and $d, r \in \mathbb{N}$. $d$ and $r$ respectively represent the distance from the root vertex (modulo 3) and the computation round of a given vertex. The initial label on the root vertex $U$ is $(R, 0, 1)$ and all the other vertices have the label $l_0 = (N, 0, 0)$. All the edges have initially the label 0. Thus, the rooted tree computation is described by Algorithm 1.

**Algorithm 1.**

*R1 :* **Initializing the first level**

Precondition :
- $\lambda(v_0) = (R, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (N, 0, 0))$.

Relabeling :
- $\lambda'([v_0, v]) := 1$,
- $\lambda'(v) := (F, (d + 1)\%3, r)$.

*R2 :* **Unlock the first level (part 1)**

Precondition :
- $\lambda(v_0) = (R, d, r)$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (F, (d + 1)\%3, r) \wedge \lambda([v_0, v]) = 1)$.

Relabeling :
- $\lambda'(v_0) := (R, d, r + 1)$,

*R3 :* **Unlock the first level (part 2)**

Precondition :

- $\lambda(v_0) = (R, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (F, (d+1)\%3, r-1) \wedge \lambda([v_0, v]) = 1)$.

Relabeling :

- $\lambda'(v) := (A, (d+1)\%3, r)$.

*R4 :* **Unlock the remaining levels**

Precondition :

- $\lambda(v_0) = (A, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (F, (d+1)\%3, r-1) \wedge \lambda([v_0, v]) = 1)$.

Relabeling :

- $\lambda'(v) := (A, (d+1)\%3, r))$.

*R5 :* **Add new leaves to the tree**

Precondition :

- $\lambda(v_0) = (A, d, r)$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) = (N, 0, 0))$.

Relabeling :

- $\lambda'(v) := (F, (d+1)\%3, r))$,
- $\lambda'([v_0, v]) := 1$.

*R6 :* **Lock internal vertices of the tree**

Precondition :

- $\lambda(v_0) = (A, d, r)$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge \lambda(v) \neq (N, 0, 0))$,
- $\forall w \in B(v_0, 1)(w \neq v_0 \wedge \lambda([v_0, w]) = 1 \Rightarrow (\lambda(v) = (F, (d+1)\%3, r) \vee \lambda(v) = (A, (d-1)\%3, r) \vee \lambda(v) = (R, (d-1)\%3, r)))$.

Relabeling :

- $\lambda'(v_0) := (F, d, r)$.

**Lemma 1.** *Let $D$ be the diameter of graph $G$. At the end of round $i$ $1 \leq i \leq D$, all the $1-$labeled edges build a rooted tree $\mathcal{T}_{u_0}$ that contains the root $u_0$ and all $F-$labeled vertices. $\mathcal{T}_{u_0}$ has therefore a depth of $i$.*

*Proof.* We show this lemma by induction on $i$. We recall that initially all vertices different from $U$ are labeled with $(N, 0, 0)$. During the execution of round $i = 1$, only rule $R1$ can be executed. This round ends with the execution of rule $R2$. Thus, only the neighbor vertices of $U$ are $F$-labeled and they build (with $U$) a tree $\mathcal{T}_U^1$ of minimal paths rooted at $U$. $\mathcal{T}_U^1$ has therefore depth 1. Let $\mathcal{T}_U^i$ be the constructed tree after round $i$. By the induction hypothesis we know that all vertices $v \neq U$ of $\mathcal{T}_U^i$ are $F$-labeled. During the computation of round $i+1$, all the $F$-labeled are first unlocked (see rules $R3$ and $R4$). Thereafter, rule $R5$ increases the rooted tree by adding new $F$-marked vertices (at most one new vertex per leave) to $\mathcal{T}_U^i$. At the end of round $i+1$, all $A$-labeled vertices are

*locked* through rule $R6$. From the preconditions and the effects of the rules $R5$ and $R6$ we can deduce that $\mathcal{T}_U^{i+1}$ is a minimal path tree rooted at $U$ and having depth $i+1$.

**Corollary 1.** *Let $v \in V$ be a $F-$labeled vertex and $p = \{v, v_0, v_1 ..., v_j, u_0\}$ a path from $v$ to $u_0$ such that all $v_i (i \leq j)$ are $F-$labeled and $v_i \neq u_0$, $v_i \neq v_k$, $\forall i, k \leq j$. Then $p$ is a minimal path from $v$ to $u_0$.*

Adding two adequate relabeling rules makes it possible to generate rooted trees of minimal paths having a depth $k$ such that $1 \leq k \leq D$, $k \in \mathbb{N}$. Such an improvement and the corresponding proofs are presented in [8].

**Lemma 2.** *Let $\mathcal{T}_U^d$ be a rooted tree of depth $d$. The time complexity of constructing $\mathcal{T}_U^d$ is $O(d^2)$ and the message complexity is $O(|E| + n * d)$.*

*Proof.* Let $\mathcal{T}_U^i$ represents the tree of minimal paths, of depth $i$ and rooted at vertex $U$. We recall that all vertices of $\mathcal{T}_U^i$ must be *locked* and *unlocked* for the computation of $\mathcal{T}_U^{i+1}$. Thus, the worst case time and message complexity for computing one path rooted at $U$ of tree $\mathcal{T}_U^d$ is $\sum_{i=1}^{d} i = O(d^2), 1 \leq d \leq \mathcal{D}$. With $\mathcal{D}$ representing the diameter of $G$. Thus, the message complexity for computing $\mathcal{T}_U^d$ starting from $G$ is $O(|E|+n*d)$ [5]. All the vertices $v$ that satisfy the rules $R2$ and $R3$ can change their labels simultaneously. That is, we assume that a vertex at depth $i$ sends messages to its neighbors at depth $i+1$ simultaneously. The same fact is also true for the rules $R4$ and $R5$. For this reasons, we need $2(d+1)$ time units to construct the tree $\mathcal{T}_U^{d+1}$ from $\mathcal{T}_U^d$ . Thus, the time complexity of our procedure is given by $\sum_{i=1}^{d} 2 * (i+1) = d(d+1) + 2d = O(d^2)$.

## 4 Solving the *k*-Local Election Problem

Starting from the rooted tree algorithm described in Section 3, we intend to design a simple algorithm that should be able to solve the $k$-local election ($k \geq 2$) in an anonymous network. Let $I_u$ be the identity of a vertex $u$ and $(\mathcal{S}, >)$ be a structure model of tuples of the form $(x_1, x_2)$ where $x_1$ and $x_2$ are real numbers and $(x_1, x_2) > (x_3, x_4) \Leftrightarrow (x_1 > x_3) \vee (x_1 = x_3) \wedge (x_2 > x_4)$. Basically this algorithm works in three steps.

**Procedure 1.**

**Step 1:** Each vertex $u$ chooses a random number $r_u$ and takes advantage of its tuple $n_u = (t_u, I_u) \in \mathcal{S}$ to perform a $2-$local election ($RL_2$). The winners and losers of these elections are respectively marked with $W$ and $L$.

**Step 2:** Each $W-$marked vertex $u$ starts the construction of the tree $T_u^d$ (with depth $d$) of minimal paths rooted at $u$.

**Step 3:** Once $T_u^d$ is constructed for a given vertex $u$, the tuples of all the $W-$marked vertices in $T_u^d$ are compared to $n_u$. If $n_u > n_v, \forall v \in T_u^d, v \neq u$ ($v$ is $W$-marked) then $u$ has won the local election in the ball of radius $k = d$ centered on $u$.

The use of minimal paths trees ensures that the tree $T_u^d$ contains all vertices of the set $\{v \in V(G) | d(u,v) \leq d\}$. We remind that Algorithm 1 was designed for a single root. Thus, all labels were related to the computation of the same tree. In Procedure 1, several trees have to be computed in a distributed way. For this reason, the label of each vertex $v$, in the new algorithm, includes a set $\mathcal{L}_v$ of tuples representing the different states of $v$ in the computations of all the rooted minimal paths trees that contain $v$. Furthermore, to encode the marking of edges and to distinguish the different elements of $\mathcal{L}_v$, we relax one specification of our model and require that each vertex has a unique identity. The label of each vertex $v$ also indicates if $v$ has won the $RL_2$ procedure. Moreover, it includes an item that represents the label of $v$ during the computation of the tree of minimal paths rooted at $v$.

*Remark 1.* The use of identities is certainly a weak point of our algorithm. Nevertheless, we will see that without identities, our algorithm solves the $k-$local election with very high probability. Moreover, the structure $(\mathcal{S}, >)$ ensures that at least one vertex terminates the election as winner. We are now ready to present the basics of our algorithm for the $k-$local election problem ($k \geq 2$).

**Definition 1.** *A tuple structure model $(\mathcal{T}, <)$ is an irreflexive total ordering of tuples of the form $(I_v, s_w, m_w, r_v^w, M_w, \mathcal{F}_w^v), \forall v, w \in V$ where:*

- *$I_v$ is the identity of a root node $v$,*
- *$s_w$ is an element of the set $\{R, A, F\}$,*
- *$m_w = d(v,w)\%3$,*
- *$0 \leq r_v^w = d(v,w) \leq k$ is the round of vertex $w$ in the computation of the minimal path tree rooted at $v$,*
- *$M_w$ is the maximal tuple (on the minimal path between $v$ and $w$) known by $w$ so far,*
- *$\mathcal{F}_w^v$ is the identity of the father of vertex $w$ in the minimal paths tree rooted at $v$.*

**Definition 2.** *For all $u, v, i, j \in V$, let $t_i = (I_v, s_i, m_i, r_v^i, M_i, \mathcal{F}_i^v)$ and $t_j = (I_u, s_j, m_j, r_u^j, M_j, \mathcal{F}_j^u)$ be two elements of $(\mathcal{T}, <)$. Then*

- *$t_i = t_j^+$ if and only if $I_v = I_u$, $(m_j + 1)\%3 = m_i$, $r_u^j = r_v^i$, $\mathcal{F}_i^v = I_j$, $s_i = F$ if $s_j = A$ and $s_i = F$ if $s_j = R$. We say $t_j < t_i$.*
- *$t_i = t_j^-$ if and only if $I_v = I_u$, $(m_j - 1)\%3 = m_i$ and $r_v^i = r_u^j$, $\mathcal{F}_j^u = I_i$, $s_i = A$ if $s_j = F$ and $s_i = R$ if $s_j = F$. We say $t_i < t_j$.*

Let $G$ be a graph, for each vertex $v$ we assume that $\lambda(v) = (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v, t_v, \mathcal{E}_v)$ where:

- $\mathcal{I}_v$ is the identity of the vertex $v$,
- $\mathcal{S}_v$ is an item that indicates the state of vertex $v$ ($L$ or $W$). Initially all vertices are in the $L$ state.
- $\mathcal{L}_v$ is a set of elements of $(\mathcal{T}, <)$,

– $t_v$ is the label of vertex $v$ in the tree of minimal paths rooted at $v$,
– $\mathcal{E}_v$ is an item that indicates if $v$ has won the $k-$local election (*elected*,*non_elected*).

Initially $\lambda(v) = (\mathcal{I}_v^0, \mathcal{S}_v^0, \mathcal{L}_v^0, t_v^0)$ for all vertices $v$ with $\mathcal{I}_v^0 = \mathcal{I}_v$, $\mathcal{S}_v^0 \in \{L, W\}$, $\mathcal{L}_v^0 = \emptyset$ and $t_v^0 = (I_v, R, 0, 1, n_v, I_v)$. All vertices are in the state *non_elected*. Procedure 1 (Steps 2 and 3) is then computed by the relabeling rules given in Algorithm 2.

**Algorithm 2.**

*R1 :* **Initializing the first level**

Precondition :

- $\lambda(v_0) = (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0}, t_{v_0}) \wedge \mathcal{S}_{v_0} = W$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \forall t \in \mathcal{L}_v(t = (I_w, x, m_v, r_w^v, M_{v_0}, \mathcal{F}_v^w) \wedge I_w \neq I_{v_0} \wedge x \in \{R, F, A\}))$.

Relabeling :

- $\lambda^{'}(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_{v_0}^+, t_v, \mathcal{E}_v)$.

*R2 :* **Unlock the first level(part 1)**

Precondition :

- $\mathcal{S}_{v_0} = W \wedge t_{v_0} = (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\forall v \in B(v_0, 1)(v \neq v_0 \wedge t_{v_0}^+ \in \mathcal{L}_v)$.

Relabeling :

- $t_{v_0}^{'} := (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0} + 1, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\lambda^{'}(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0}, t_{v_0}^{'}, \mathcal{E}_{v_0})$.

*R3 :* **Unlock the first level (part 2)**

Precondition :

- $\mathcal{S}_{v_0} = W \wedge t_{v_0} = (I_{v_0}, R, m_{v_0}, r_{v_0}^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^{v_0})$,
- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_v(t_i = (I_{v_0}, F, (m_{v_0} + 1)\%3, r_{v_0}^{v_0} - 1, M_v, \mathcal{F}_v^{v_0})))$.

Relabeling :

- $\mathcal{L}_v := \mathcal{L}_v - t_i$,
- $t := (I_{v_0}, F, (m_{v_0} + 1)\%3, r_{v_0}^{v_0} + 1, M_v, \mathcal{F}_v^{v_0})$,
- $\lambda^{'}(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t, t_v, \mathcal{E}_v)$,

*R4 :* **Unlock the remaining levels**

Precondition :

- $\exists v \in B(v_0, 1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_v(\exists t_j \in \mathcal{L}_{v_0}(t_i = (I_p, F, m_v, r_p^v, M_v, \mathcal{F}_v^p) \wedge t_j = (I_p, A, (m_v - 1)\%3, r_p^v + 1, M_{v_0}, \mathcal{F}_{v_0}^p))))$.

Relabeling :

- $\lambda^{'}(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v - t_i, t_v, \mathcal{E}_v)$,
- $t_i := (I_p, A, m_v, r_p^v + 1, M_v, \mathcal{F}_v^p)$,
- $\lambda^{'}(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_i, t_v, \mathcal{E}_v)$.

*R5 :* **Add new leaves to the tree**

Precondition :

- $\exists v \in B(v_0,1)(v \neq v_0 \wedge \exists t_i \in \mathcal{L}_{v_0}(t_i = (I_p, A, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p)))$,
- $\nexists t \in \mathcal{L}_v(t = (I_q, x, m_v, r_q^v, M_v, \mathcal{F}_v^q) \wedge I_p = I_q) \wedge x \in \{R, F, A\}$.

Relabeling :

- $M_v^* := max(M_{v_0}, n_v)$,
- $t_i := (I_p, A, m_{v_0}, r_p^{v_0}, M_v^*, \mathcal{F}_{v_0}^p)$,
- $\lambda'(v) := (\mathcal{I}_v, \mathcal{S}_v, \mathcal{L}_v + t_i^+, t_v, \mathcal{E}_v))$.

*R6 :* **Lock internal vertices of the tree**

Precondition :

- $\exists t_i \in \mathcal{L}_{v_0}(t_i = (I_p, A, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p))$,
- $\forall v \in B(v_0,1)(v \neq v_0 \wedge \exists t_j \in \mathcal{L}_v(t_j = (I_q, x, m_v, r_q^v, M_v, \mathcal{F}_v^q) \wedge \mathcal{F}_v^p = I_{v_0} \wedge I_q = I_p \wedge x \in \{R, F, A\}) \Rightarrow t_j = (I_p, F, (m_{v_0}+1)\%3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \vee t_j = (I_p, A, (m_{v_0}-1)\%3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \vee t_v = (I_p, R, (m_{v_0}-1)\%3, r_p^p, M_p, \mathcal{F}_p^p))$.

Relabeling :

- $\lambda'(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0} - t_i, t_{v_0}, \mathcal{E}_{v_0})$,
- $\forall v \in B(v_0,1)(v \neq v_0 \wedge \exists t_j \in \mathcal{L}_v(t_j = (I_p, F, (m_{v_0}+1)\%3, r_p^{v_0}, M_v, \mathcal{F}_v^p) \wedge \mathcal{F}_v^p = I_{v_0} \wedge I_q = I_p \Rightarrow M_{v_0} := max(M_{v_0}, M_v)))$,
- $t_i := (I_p, F, m_{v_0}, r_p^{v_0}, M_{v_0}, \mathcal{F}_{v_0}^p)$,
- $\lambda'(v_0) := (\mathcal{I}_{v_0}, \mathcal{S}_{v_0}, \mathcal{L}_{v_0} + t_i, t_{v_0}, \mathcal{E}_{v_0})$.

*Remark 2.* To know that a vertex $v$ is already involved in the computation of the tree rooted at a vertex $w$, one has in Algorithm 2 to look for an element $t \in \mathcal{L}_v$ whose root vertex has the same identity as $w$. This fact gives the above rules a more complicated aspect as the rules described in Algorithm 1 . Nevertheless, these rules exactly perform the same. The rules $R5$ and $R6$ ensure that each time an internal vertex $w$ is *locked*, the maximal known tuple $M_w$ is actualized bottom-up. This actualization is done up to and including the vertices of the first level. Some improvements of this algorithm are given in [8].

**Corollary 2.** *During the computation of the rooted minimal paths tree $T_v^D$, at the end of round $i \leq D$, $v$ is aware of the maximum tuple $n_m \in \mathcal{S}$ amongst the tuples generated by all the $W$-marked vertices of $T_v^i$.*

### 4.1 $k$-Local Election for Anonymous Networks

Due to the use of identities, Algorithm 2 is not adapted for anonymous networks. We now present a variant of Procedure 1 that is able to solve the $k$-local election problem in an anonymous network.

**Procedure 2.**

**Step 1:** Each vertex $v$ chooses a random number $n_v$ and performs a 2−local election ($RL_2$). The winners and losers of these elections are respectively marked with $W$ and $L$.

**Step 2:** Each $W-$marked vertex $u$ chooses a random number $n_u^*$ and set its identity to be the number $n_u^*$.

**Step 3:** For each $W-$marked vertex $u$, we construct the tree $T_u^D$ (with depth $D$) of minimal paths rooted at $u$.

**Step 4:** Once $T_u^D$ is constructed for a given vertex $u$, the chosen numbers of all the $W-$marked vertices in $T_u^D$ are compared to $n_u^*$. If $n_u^* > n_v^*, \forall v \in T_u^D, v \neq u$ ($v$ is $W$-marked) then $u$ has won the local election in its ball of radius $k = D$.

Once the first two steps have been performed, the last parts of this procedure can be computed effortlessly by Algorithm 2. Obviously, the correctness of Procedure 2 heavily depends on the absence of random numbers coincidences in the whole network. That is, if two $W$-marked vertices generate the same random number, the algorithm can not guarantee a faithful execution.

*Remark 3.* We assume that each vertex $v$ selects at random *uniformly* and *independently* an integer $rand(v)$ from $\{1, .., N\}$. Let $X$ be a set of vertices and $v$ a given vertex of $X$. Let $|X| = h$. Then under the above assumptions on *rand* we obtain the probability,

$$Pr(rand(v) \neq rand(w), \forall w \in X - \{v\}) = \frac{1}{N} \sum_{i=1}^{N} \left( \frac{N-1}{N} \right)^{h-1}.$$

We need to reduce the value of $Pr(rand(v) = rand(w), \forall w \in X - \{v\})$. To achieve this task, we assume in our framework that the set $X$ represents the set of all $W$-marked vertices in the network and that $|X| < N$. Due to Fact 1 we know that $|X| \leq \frac{|V|}{2}$. Furthermore, the integer $N$, which is the range of selection for vertices, is supposed to be large enough, so that the probability of coincidence of *rand* in the whole network becomes small. Thus,

$$Pr(rand(v) = rand(w), \forall w \in X - \{v\}) = 1 - \frac{1}{N} \sum_{i=1}^{N} \left( \frac{N-1}{N} \right)^{\frac{|V|}{2}-1},$$

$$Pr(rand(v) = rand(w), \forall w \in X - \{v\}) = 1 - \frac{1}{N^{\frac{|V|}{2}}} \sum_{i=1}^{N} (N-1)^{\frac{|V|}{2}-1},$$

$$Pr(rand(v) = rand(w), \forall w \in X - \{v\}) = 1 - \left(1 - \frac{1}{N}\right)^{\frac{|V|}{2}-1}.$$

This means that if $N$ is large enough, the probability that two vertices generate the same random number converges to 0. This assumption is equivalent to the one supposing that all vertices choose at random uniformly and independently a *real* from the interval $[0, 1]$.

**Fact 2.** *With the requirements presented in Remark 3, Procedure 2 solves with high probability the k-local election problem in an anonymous network.*

**Lemma 3.** *The time complexity of solving the $k$-local election for a vertex $v_0$ is the same as the complexity required to compute a tree of minimal paths with depth $k$ and rooted at $v_0$. Thus, the complexity of Procedure 2 is $O(k^2)$.*

*Proof.* Let $n = |V(G)|$. The $RL_2$ procedure has a message complexity of $O(n^2 + \sum_{v \in G} |N_G(v)|) = O(n^2 + n(n-1))$. Under the time assumptions we have made, it is easy to see that a given vertex $v_0$ knows, after a constant time, that it has won or lost the 2-local election. Furthermore, Procedure 2 has also a worst case message complexity of $O(\gamma(|E| + n * k))$. With $\gamma \leq \frac{n}{2}$ representing the number of vertices that have won the $RL_2$ procedure. The time complexity of solving the $k$-local election problem for a given vertex $v_0$ is the same as the time required to compute a tree of minimal paths with depth $k$ and rooted at $v_0$. Thus, the time complexity of our procedure is $O(k^2)$. This means that the time complexity of solving the $k$-local election problem is not constrained by the topology of the network nor by the size of the underlying graph $G$.

## 5 Concluding Remarks

We have presented a new randomized algorithm that, with very high probability, solves the $k-$local election problem ($k \geq 2$) in anonymous networks. The presented protocol has already been successfully used to implement distributed graph reduction algorithms. For further researches, we expect this algorithm to be another step in the computation of more complex problems in the local computation environment and more generally in the distributed computation framework.

## References

1. D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing*, pages 82–93, 1980.
2. B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.
3. M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.
4. I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.
5. N. A. Lynch. *Distributed algorithms*. Morgan Kaufman, 1996.
6. Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inform. Proc. Letters*, pages 313–320, 2002.
7. E.F Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the theory of Switching*, pages 285–292. Cambridge, Massachusetts, Harvard University Press, Cambridg, 2-5 April 1959.
8. Rodrigue Ossamy. A simple randomized k-local election algorithm for local computations. Technical Report 1344-05, LaBRI-University of Bordeaux I, 2005.

# Generating and Radiocoloring Families of Perfect Graphs

M.I. Andreou[1], V.G. Papadopoulou[2], P.G. Spirakis[2], and B. Theodorides[2], and A. Xeros[2,⋆]

[1] Intercollege, Makedonitissas Ave. 46, P.O. Box 24005, 1703 Nicosia, Cyprus
[2] Research Academic Computer Technology Institute (RACTI) and Patras University, Greece. Riga Fereou 61, 26221 Patras, Greece
andreou.m@intercollege.ac.cy,
{viki, spirakis, theodorb, zeros}@cti.gr

**Abstract.** In this work we experimentally study the *min order Radiocoloring problem* (RCP) on *Chordal, Split* and *Permutation* graphs, which are three basic families of *perfect graphs*. This problem asks to find an assignment using the minimum number of colors to the vertices of a given graph $G$, so that each pair of vertices which are at distance at most two apart in $G$ have different colors. RCP is an NP-Complete problem on chordal and split graphs [4]. For each of the three families, there are upper bounds or/and approximation algorithms known for minimum number of colors needed to radiocolor such a graph[4, 10].

We design and implement radiocoloring heuristics for graphs of above families, which are based on the greedy heuristic. Also, for each one of the above families, we investigate whether there exists graph instances requiring a number of colors in order to be radiocolored, close to the best known upper bound for the family. Towards this goal, we present a number generators that produce graphs of the above families that require either (i) a large number of colors (compared to the best upper bound), in order to be radiocolored, called "extremal" graphs or (ii) a small number of colors, called *"non-extremal"*instances. The experimental evaluation showed that random generated graph instances are in the most of the cases *"non-extremal"* graphs. Also, that greedy like heuristics performs very well in the most of the cases, especially for *"non-extremal"* graphs.

## 1 Introduction

The *Problem of Frequency Assignment* (FAP) consists of assigning frequencies to the transmitters of a wireless network exploiting frequency reusability in order to save bandwidth, while keeping the interference caused when nearby stations transmit in the same or close frequency, in acceptable levels. This problem is usually modelled as a *vertex coloring problem*. However, the vertex coloring model fails to describe some realistic scenarios of practical wireless networks, because

⋆ This work has been partially supported by the EU IST/FET projects CRESCCO.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 302–314, 2005.
© Springer-Verlag Berlin Heidelberg 2005

in this case the only requirement is to assign just different colors to adjacent vertices in a graph. Henceforth, a number of generalizations of the vertex coloring problem have been introduced and investigated in the past, towards this direction [8].

Here we study a variation of FAP, called *min order Radiocoloring Problem (min order RCP)* [5], on three basic families of *perfect graphs* that of *permutation*, *chordal* and *split* graphs. The problem consists of assigning colors (frequencies) to the vertices (transmitters) of a graph (network), so that any two vertices of distance at most two apart get different colors. The objective is to minimize the number of distinct colors used. Permutation graphs model well networks where two groups of independent transmitters want to have communication with transmitters from the opposite group. Split graphs model networks where a number of independent transmitters want to communicate with a set of strongly communicated transmitters.

### 1.1 Definitions and Notation

Let $G(V, E)$ be a graph. The *size* of $G$ is the number of its vertices and is denoted by $n$ (i.e., $n = |V|$). W.l.o.g. we assume that $V = \{1, 2, ..., n\}$). We denote by $\Delta(G)$ the maximum degree of $G$. When there is no confusion we omit $G$ and we refer to it simply as $\Delta$. We denote as $C_x$ a cycle on $x$ vertices (i.e., of size $x$) in a graph $G$. A *chord* in a cycle is an edge joining two non-consecutive vertices of a cycle. A cycle graph that does not have chords is called *unchord*. A tree graph is denoted by $T$ and its *root* vertex by $r$. For a vertex $v$ in $T$, we denote as $layer(v)$ its distance from the root plus one (1). The father of $v$ in $T$ is denoted as $father(v)$. The *lowest common ancestor (LCA)* of two vertices $u, w$ in $T$ is denoted as $LCA(u, w)$.

**Definition 1.** *([6]) The* **min order Radiocoloring Problem (min order RCP)** *of a given graph $G$ is the problem of coloring the graph $G$ with a minimum number of colors so that any two vertices of $G$ of distance at most two apart get different colors. The minimum such number is called the* radiochromatic number *of $G$ and is denoted by $\lambda(G)$.*

In this work, we concentrate only on min order RCP. Henceforth, for simplicity reasons, we refer to it as the radiocoloring problem (RCP). In the following, *uniformly random permutation* of numbers 1 to $n$ is a permutation of the vertices in which the position of any number is chosen uniformly random from the positions that are free in the permutation. The first family of perfect graphs considered is that of *permutation graphs*. These graphs can be defined based on a permutation of their vertices, i.e., a permutation $\pi$ of numbers $1, 2, \ldots, n$. Let us think of $\pi$ as the sequence $[\pi_1, \pi_2, \cdots, \pi_n]$. We denote by $\pi_i^{-1}$ the position of number $i$ in the sequence.

**Definition 2.** *[7] Let $\pi = [\pi_1, \pi_2, \cdots, \pi_n]$ be a permutation of numbers 1 to $n$. Then, the permutation graph determined by $\pi$ is the graph $G[\pi] = (V, E)$ with $V = \{1, \cdots, n\}$ and $(i, j) \in E$ iff $(i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0$.*

Observe that for any vertex $i$ of $G[\pi]$, $j \in V$ is a neighbor of $i$ iff $i > j$ and $i$ is on the left of $j$ in permutation $\pi$. A graph $G$ is a permutation graph if there exists a permutation $\pi$ such that $G$ is *isomorphic* to $G[\pi]$. Let a graph $G(V, E)$ and a permutation $\sigma$ of its vertices. We denote by $G_i = (V_i, E_i)$ the induced subgraph of $G$ on the vertices of the set $V_i$, where $V_i = \{\sigma_i, \sigma_{i+1}, \sigma_{i+2}, ..., \sigma_n\}$ and $\sigma_j$ is the j-th vertex in $\sigma$.

**Definition 3.** *[7] An undirected graph $G$ is* chordal *if each of its cycles of size bigger than three has a chord. A vertex $v$ of it, $v$ is a* simplicial *vertex if the neighbors of $v$ form a clique in $G$. The ordering $\sigma$ is a* perfect elimination scheme (PES) *if each vertex $\sigma_i$ is a simplicial vertex in $G_i$, for $1 \leq i \leq n$.*

**Theorem 1.** *[7] An undirected graph $G$ is chordal iff it has a PES.*

Split graphs consist a subfamily of chordal graphs [7].

**Definition 4.** *A* split graph *is a graph $G(V, E)$ of which its vertex set can be split into two sets $K$ and $S$, such that $K$ induces a clique and $S$ induces an independent set (its vertices are not incident to each other).*

In the experimental results shown, we denote $n$ the size of the vertices of the graph $G$, $\Delta$ the maximum degree of $G$, $p$ a probability and $\#Colors$ the number of colors used by the radiocoloring algorithm evaluated on the graph $G$.

### 1.2 A Greedy Radiocoloring Heuristic

A basic radiocoloring heuristic investigated in this work is a simple *greedy* heuristic, called *Radiocoloring First Fit (RFF)* Heuristic. This algorithm generalizes the well known *First Fit (FF)* heuristic for ordinary vertex coloring. It takes as input a predefined ordering $O$ of the vertices of $G$ and produces a radiocoloring of $G$. Most of the heuristics developed in this work apply $RFF$ radiocoloring algorithm on various interesting orderings of the vertices of $G$. We denote by $O_i$ the vertex located at position $i$ of the ordering $O$.

**Radiocoloring First Fit Heuristic ($RFF$)**
*Input*: a graph $G(V, E)$ and an ordering $O$ of its vertices.
*Output*: a radiocoloring of $G$.

1. **For** i := 1 to $n$ **do:**
   color the vertex $O_i$ with the smallest color not assigned to any of its distance one or two neighbors located on its left in the ordering $O$.

### 1.3 Previous Related Work

The radiocoloring problem has been studied in a number of theoretical works [5, 10, 1, 4]. These works consider RCP on a number of interesting families of graphs, such as: general, strongly chordal, paths, cycles, trees, graphs of diameter two, grids, hypercubes, bipartite, planar, chordal, split, treewidth and outerplanar

graphs, providing upper bounds, algorithms and hardness results. In contrast, the problem has not been investigated in the past experimentally. From the best of our knowledge, the only known experimental work on RCP is that of Andreou et al. 3 [3] concentrated on planar graphs.

Here, we study at first time experimentally RCP on three families of perfect graphs: permutation, chordal and split graphs. Most of the theoretical work done for these families of graphs is by Bodlaender et al. and Sakai [4, 10]. Moreover, from the best of our knowledge, this is the first work that concentrates on the problem of generating graph instances of families of perfect graphs, especially when RCP is of concern.

In the sequel, we introduce the definitions of the families of *perfect graphs* studied here and the relative known theoretical results related to our study. A graph $G$ is perfect iff $\chi(H) = \omega(H)$, where $\chi(G)$ is the *chromatic number* of $G$, $H$ is any *induced subgraph* of $G$ and $\omega(H)$ is the *clique number* of $H$, i.e., the size of the maximum clique (complete subgraph) in $H$. The radiocoloring problem for chordal and split graphs becomes $NP$-complete [4] while for permutation graphs its complexity is still unknown. We proceed with a useful proposition on RCP and a related previous work.

**Proposition 1.** *[6] For any graph $G$, $\Delta(G)^2 + 1 \geq \omega(G) \geq \lambda(G) \geq \Delta(G) + 1$.*

**Theorem 2.** *[4] For any permutation graph $G$, $\lambda(G) \leq 3\Delta(G) - 2$. The* Radiocoloring First Fit *heuristic $RFF(G, O)$ (see previous section) radiocolors a permutation graph $G$ this number of colors, when $O = \{1, 2, ..., n\}$.*

**Theorem 3.** *[4] Given a chordal or a split graph $G(V, E)$ and an integer $r$, the problem of deciding whether $\lambda(G) \leq r$ is $NP$-complete.*

**Theorem 4.** *[10] For any chordal graph $G$, $\lambda(G) \leq (\Delta(G) + 3)^2/4$. Heuristic $RFF(G, O)$ radiocolors a chordal graph $G$ using this number of colors, when $O = \{\sigma_n, \sigma_{n-1}, ..., \sigma_1\}$ and $\sigma$ is a PES of $G$.*

**Theorem 5.** *1. [4] For any split graph $G$, $\lambda(G) \leq \Delta(G)^{1.5} + \Delta(G) + 1$. There exists a polynomial time radiocoloring algorithms that computes an assignment of this number of colors.*

*2. [4] For any $\Delta$, there exists a polynomial time generator that constructs a split graph $G$ with $\lambda(G) \geq (1/3)\sqrt{(2/3)}\Delta(G)^{1.5}$.*

### 1.4 Research Objectives and Approach

As we reviewed in previous section, for all three families there are only approximation algorithms or upper bounds known for the problem. An important open question for research on RCP on these families of perfect graphs is how close the known upper bounds are to the optimal solution. This work, we investigate there are instances of graphs of the families of our interest that require a number of colors close to the best known upper bounds in order to be radiocolored.

Towards this goal, we utilize the following research approach: we construct, in polynomial time, graphs for which their radiochromatic number is at least $X$, where $X$ we seek to be close to the best known upper bound of RCP on such graphs. We call such graphs as *extremal* graphs. In particular, to guarantee this, we aim in constructing graphs for which $X$ of their vertices are at distance at most two to each other. Thus, any radiocoloring assignment on them requires at least $X$ distinct colors.

We also generate instances of permutation, chordal and split graphs that besides the necessary properties of their family, they do not present any additional properties. More analytically, these "*average case*" graph instances have $\lambda(G) \approx \Delta(G) + constant$ or $\lambda(G)$ *relatively* small compared to the best upper bound of $\lambda(G)$ for the corresponding graph family. We call such graphs as *"non-extremal"* and we usually construct them using randomness.

We estimate the number of colors needed for radiocoloring the graphs generated, by utilizing (i) known radiocoloring algorithms of guaranteed performance or/and (ii) efficient heuristics, that we provide here.

In particular, we introduce an interesting variation of the greedy radiocoloring heuristic $RFF$ for permutation graphs. We also introduce a new greedy radiocoloring heuristic for split graphs. We evaluate their performance experimentally on the graph instances generated. Also, we compare them with known radiocoloring (approximation or optimal) algorithms, throughout extensive experiments. Our contribution is summarized in the following:

1. We design and implement, polynomial time, generators (algorithmic constructions) that produce extremal permutation, split and chordal graphs. For the case of permutation graphs we manage to produce graphs $G$ such that $\lambda(G) = 2\Delta(G)$ ($3\Delta(G) - 2$ is the best upper bound). In the case of split graphs (also for chordal graphs), we implement a known generator [4] to produce split graphs $G$ such that $\lambda(G) \geq \Delta(G)^{1.5}$ ($\Delta(G)^{1.5} + \Delta(G) + 1$ is the best upper bound).
2. We design and implement, polynomial time, generators for *"non-extremal"* instances permutation, split and chordal graphs. In the most of the cases, these graph instances have $\lambda(G) \approx \Delta(G) + cost$, where *cost* is a constant.
3. We implement known radiocoloring algorithms for chordal and split graphs [4]. We design and implement new heuristics for radiocoloring permutation and split graphs. We evaluate their performance on various graph instances produced by the generators provided. The experimental findings show that the radiocoloring heuristics proposed have very good performance on *"non-extremal"* permutation, chordal and split graphs. Actually they use $\Delta(G) + constant$ colors to radiocolor such a graph $G$. We remark that this number is much smaller than the best known upper bounds for the radiochromatic number of such graphs.

All our implementations are written in the C++ programming language with the support of the LEDA library, [9], in a UNIX environment. The rest of this paper is organized as follows: In Section 2, we provide new generators and a

heuristic for RCP for permutation graphs. In Section 3, we provide generators for chordal graphs and in Section 4 new and known generators and radiocoloring heuristics for split graphs. Finally, in the last section we present our experimental findings and conclude our work. Due to lack of space, the omitted pseudocodes (and the actual codes) of algorithms and proofs of theorems, lemma, claims etc, presented, are included in the full version of the paper [2].

## 2 Permutation Graphs

From the best of our knowledge, the only known result for RCP on a permutation graph $G[\pi]$ is $\lambda(G) \leq 3\Delta(G) - 2$ (Theorem 2) and can be obtained by $RFF(G, O)$, where $O = \{1, 2, ..., n\}$. It is also known that $FF(G, \pi)$ heuristic, computes an optimal *ordinary* vertex coloring of $G[\pi]$, [7]. Having this in mind, we introduce and investigate the $RFF(G, \pi)$ heuristic to radiocolor permutation graphs. Moreover, we investigate whether there exist instances of permutation graphs having a radiochromatic number close to the best known upper bound. Henceforth, we seek to construct graphs with $\Delta(G) < n/3$, so that to be possible to have $\lambda(G)$ close to $3\Delta(G)$ (we need $3\Delta(G) \leq 3\frac{n}{3} \leq n$). We first introduce two random generator that produce, in polynomial time, *"non-extremal"* instances of permutation graphs. Moreover, we provide a deterministic generator that produces graphs with $\lambda(G) = 2\Delta(G)$.

### 2.1 Generators for Permutation Graphs

In the following, we call an integer number $i$ as *big number* or *big vertex* (numbers represent vertices identities) if $n \geq i \geq n/2$, otherwise we call $i$ as *small number* or *small vertex* (i.e., $1 \leq i < n/2$).

**Random Permutation 1 (*RP*1) Algorithm.** The first generator provided, called $RP1$, is based on a random permutation $\pi$ of numbers 1 to $n$, representing the vertices of the graph $G[\pi]$. $RP1$ takes an integer $n$ (the size of the graph to be produced). It first, finds a *uniformly random permutation* $\pi$ of numbers 1 to $n$. Then, it produces the permutation graph $G[\pi]$ applying sequence $\pi$ on the constructive Definition 2.

**Observation 2.1** *Every permutation graph can be produced by* $RP1$ *with positive probability.*

*Claim.* With high probability, a permutation graph produced by the generator $RP1$ has maximum degree close to $n$ (call this *Event 1*). Thus, with high probability, the graph has $\lambda(G) \approx \Delta(G)$, i.e. it is *"non-extremal"*.

**Random Permutation 2 (*RP*2) Algorithm.** The next random generator differentiates from $RP1$ only on the way it places the vertices in the permutation: Each *big* vertex is placed in the second half of the permutation $\pi$ if a Bernoulli

trial, with probability of success $p$, is successful. Otherwise, the vertex is placed in the first half of $\pi$. The opposite holds for a *small* vertex. Note that as $p$ tends to 1, the *Event 1* happens less frequently.

**Deterministic Permutation ($DP$) Algorithm.**

**Deterministic Permutation ($DP$) Algorithm**
*Input*: a positive integer $n$.
*Output*: a permutation graph $G[\pi]$ with $\Delta(G) = \frac{n}{2}$ all of its vertices are at distance at most two apart to each other.

1. Set numbers $\frac{n}{2}+1, \frac{n}{2}+2, ..., n$ in positions $1, 2, ..., \frac{n}{2}$ of a permutation $\pi$, respectively.
2. Set numbers $1, 2, ..., \frac{n}{2}$ in positions $\frac{n}{2}+1, \frac{n}{2}+2, \ldots, n$ of $\pi$, respectively.
3. Create the permutation graph $G[\pi]$ according the constructive Def. 2 on $\pi$.

**Lemma 1.** *Any graph produced by Algorithm DP is a permutation graph. It has maximum degree $\Delta = n/2$ and needs $2\Delta(= n)$ colors to be radiocolored. Also, its vertices are at distance at most two apart to each other.*

### 2.2 Experimental Results for Permutation Graphs

We use the $RFF(G[\pi], \pi)$ heuristic to radiocolor permutation graphs produced by the generators provided here. In the following paragraphs, we present our experimental findings on graphs produced by generators $RP1$, $RP2$ and $DP$.

**Radiocoloring on Graphs produced by $RP1$.** We apply the radiocoloring heuristic $RFF(G[\pi], \pi)$ on permutation graphs generated by $RP1$. Table 1 shows most of representative our experimental findings. All graph instances obtained are radiocolored using $\Delta(G) + const$ colors, where *const* is a constant independent of the size of the graph, i.e. less than $3\Delta(G)$, the best upper bound of $\lambda(G)$ on such graphs, obtained by $RFF(G[\pi], O)$, $O = \{1, 2, ..., n\}$ [4]. Hence, our radiocoloring heuristic has almost optimal performance on such permutation graphs, since $\lambda(G) \geq \Delta(G) + 1$ (Proposition 1). As explained above, and illustrated in the results obtained, the maximum degree of the graphs produced by $RP1$ is close to $n$.

**Radiocoloring on Graphs produced by $RP2$.** We have evaluated the generation algorithm $RP2$ for various values of the probability $p$. Note that the most interesting cases are for values of $p > 0.5$, as explained above. As $p$ increases, the probability to get graphs of smaller maximum degree $G[\pi]$ also increases. Of course, if $p = 1$, then all *big* (*small*) vertices are at the second (first) half of $\pi$. On the other hand, the opposite holds when $p = 0$. When $p = 0.5$, $RP2$ equals to $RP1$.

Table 1 shows the most representative graphs instances obtained by the generator, when $p = 0.75$. As we can see, $RP2$ do not manage to produce graphs that have maximum degree less than $n/2$; their maximum degree is close to $n$, as in $RP1$. The performance of the radiocoloring heuristic $RFF(G[\pi], \pi)$ on such graphs is actually the same as on the instances of the permutation graphs obtained by $RP1$ as seen in the results presented in the table.

**Table 1.** Radiocoloring permutation graphs generated by $RP1$ and $RP2$ using algorithm $RFF(G[\pi], \pi)$

| RP1 | | | PR2 | | | |
|---|---|---|---|---|---|---|
| $\boldsymbol{n}$ | $\boldsymbol{\Delta}$ | **#Colors** | $\boldsymbol{n}$ | $\boldsymbol{\Delta}$ | **#Colors** | **p** |
| 512 | 439 | 483 | 512 | 487 | 494 | 0.75 |
| 600 | 569 | 590 | 600 | 567 | 581 | 0.75 |
| 1000 | 971 | 987 | 1000 | 971 | 984 | 0.75 |

**RadioColoring Graphs produced by *DP*.** $DP$ generates permutation graphs whose vertices are at distance two apart as explained above. In particular, always, $\lambda(G) = n = 2\Delta(G)$. As expected the experimental results verify the theoretical analysis, so we do not present them here.

## 3 Chordal Graphs

Our study on chordal graphs has been concentrated on the generation of such graphs. We provide three generators for chordal graphs. The first two are based on randomness. Moreover, we implement and evaluate the performance the radiocoloring heuristic $RFF(G, \sigma')$, where $\sigma'$ is the reverse of a perfect elimination scheme $\sigma$ of a chordal $G$ of Theorem 4 ([10]).

### 3.1 Generators for Chordal Graphs

We seek to generate chordal graphs with radiochromatic number close to the best upper bound of $\lambda(G) \leq (\Delta(G) + 3)^2/4$ (Theorem 4). It is known that there exists a polynomial time constructor that generates a split graph, thus also a chordal graph, that needs order $\Delta(G)^{1.5}$ colors to be radiocolored (Theorem 5). This algorithm is implemented and evaluated in the following section for split graphs, 4.2. Here, we seek to construct chordal graphs of maximum degree $\Delta(G)$ close to $O(\sqrt{n})$, so that to be able to need the above number of colors in order to be radiocolored. We developed a number of random generators and a deterministic one for chordal graphs. However, it turns out that the graphs produced by the generators are *"non-extremal"*, requiring a few number of colors ($\approx \Delta(G) + const$), compared to the best upper bound of [4].

**Random Chordal 1 (*RC*1) Algorithm.** $RC1$ starts with an initially empty graph $G$ and finds a uniformly random order $\sigma$ of its $n$ vertices. Considering $\sigma$

as a PES of $G$, it selects the neighbors of vertex $\sigma_i$ in $G_i$ ($\sigma_i$ will be a simplicial vertex in $G_i$ for each $i$ from $n$ to 1) so that the maximum degree of $G$ remains at most $\Delta$, where $\Delta$ is a given number. Finally, it inserts appropriate edges in $G$ so that the neighbors of $\sigma_i$ in $G_i$ to form a clique.

*Claim.* Any graph $G$ obtained by $RC1$ is a chordal graph.

**Treelike-Chordal ($TC$) Algorithm.** The second new generator of chordal graphs introduced called *Treelike-Chordal* (TC), utilizes also randomness.

**Treelike-Chordal ($TC$) Algorithm**
*Input*: Two positive integers $n, k$ and a probability $p$.

*Output*: A chordal graph $G = (V, E)$ of size $n$.

1. Create a tree with $n$ vertices so that each vertex (except leaves) has $k$ children.
2. **For** each $u, v \in V$, where $u$ is the grandfather of $v$ **do** :
   Add the edge $(u, v)$ in $G$ with probability $p$.

**Proposition 2.** *Any graph $G$ produced by generator $TC$ is a chordal graph.*

**Deterministic BFS-Chordal Algorithm ($BFS - Chordal$).** The third generator introduced, called $BFS - Chordal$, takes as input an arbitrary undirected connected graph $G$, with $n$ vertices and close to $2n$ edges. Then, it adds edges to the graph so that to become a chordal as follows: It first finds a rooted Breadth First Search $T$ of the initial graph $G$. Next, it labels the vertices of $T$, layer by layer and within each layer it labels them from left to right with consecutive numbers, starting with $label(r) = 1$ for the root $r$. Note that, the edges of the graph connect vertices either of the same layer or of successive layers. The algorithm then adds edges in the initial graph so that to construct a chordal graph. This is achieved by finding the cycles in the initial graph of size bigger than three and we split them into triangles (see [2] for details).

**Lemma 2.** *$BFS - Chordal$ constructs a chordal graph in polynomial time.*

### 3.2 Experimental Results for Chordal Graphs

**Radiocoloring Graphs produced by $RC1$.** The first three columns of Table 2 show our experimental findings when radiocoloring graphs produced by the random generator $RC1$ using algorithm $RFF(G, \sigma')$. Recall that the maximum degree $\Delta$ of these graphs is predefined and is given as input to the generator. Hence, for such graphs we focus our interest on the number of colors used by $RFF(G, \sigma')$ to radiocolor chordal graphs obtained by $RC1$. $\sigma'$ is the reverse of a perfect elimination ordering of $G$. Experiments showed that this number is close to $\Delta$ (not more than $1.12\Delta$). Note that this number is much smaller than $(\Delta(G) + 3)^2/4$, the upper bound known for this radiocoloring algorithm ($RFF(G, \sigma')$) (Theorem 4).

**Table 2.** Radiocoloring Chordal Graphs produced by generators $RC1$, $Treelike - Chordal$ and $BFS - Chordal$ using algorithm $RFF(G, \sigma')$

| RC1 | | | BFS | | | Treelike | | |
|---|---|---|---|---|---|---|---|---|
| $n$ | $\Delta$ | #Colors | $n$ | $\Delta$ | #Colors | $n$ | $\Delta$ | #Colors |
| 1000 | 10 | 11 | 100 | 99 | 100 | 259 | 28 | 29 |
| 1000 | 20 | 21 | 100 | 97 | 98 | 1023 | 8 | 9 |
| 10000 | 40 | 49 | 200 | 194 | 195 | 1093 | 10 | 11 |
| 10000 | 500 | 529 | 200 | 197 | 198 | 1093 | 14 | 15 |
| 10000 | 1000 | 1156 | 300 | 285 | 286 | 1365 | 12 | 13 |
| 10000 | 1000 | 1174 | 300 | 296 | 299 | 1365 | 22 | 23 |

**Radiocoloring on Graphs produced by $TC$.** Our experimental findings for graphs obtained by generator $TC$ are also shown in Table 2. We observe that the maximum degree of these graphs is small compared to $n$. However the performance of the radiocoloring heuristic $RFF(G, \sigma')$ we use remains almost optimal, since it uses at most $\Delta(G) + const$ colors for radiocoloring these graphs.

**Radiocoloring on Graphs produced by $BFS - Chordal$.** Based on the results presented in Table 2, it can be easily seen that the number of colors used by $RFF(G, \sigma')$ is close to $n$ on chordal graphs generated by $BFS - Chordal$ algorithm. This is because, the graphs obtained by this generator have maximum degree close $n$. Observe again that the number of colors used is much smaller than $(\Delta(G) + 3)^2/4$, the upper bound of the radiocoloring algorithm used. The large maximum degree obtained is due to that we insert too many edges in order to split each cycle of an arbitrary sparse graph into triangles.

## 4 Split Graphs

The last family of perfect graphs considered is that of split graphs. We designed and implemented generators for split graphs using randomness. We also implemented the generator of Bodlaender et al. [4] producing split graphs with $\lambda(G) = O(\Delta(G)^{1.5})$. We designed and implemented a new greedy radiocoloring heuristic for such graphs. Also, we implemented a known radiocoloring algorithm of o Bodlaender et al. [4]. We estimate the number of colors needed to radiocolor the graph instances produced by the generators applying our greedy heuristic and the radiocoloring algorithm of Bodlaender et al.

### 4.1 Radiocoloring Algorithms for Split Graphs

**Bodlaender et al. Radiocoloring Algorithm ($BRcSG$).** In [4] it is proved that the following algorithm, uses at most $\Delta(G)^{1.5} + \Delta(G) + 1$ colors to radiocolor any split graph $G$, (Theorem 5).

**Bodlaender's Radiocoloring Algorithm** ($BRcSG$)
*Input*: a split graph $G(V, E)$.
*Output*: a radiocoloring of $G$.

1. Assign to each vertex of the clique $K$ a different color from set $\{1, 2, \ldots, |K|\}$.
2. **If** there is a vertex $u \in S$ with degree greater than $\Delta(G)^{1.5}$ **then**:
   (a) Remove vertex $u$ and call the algorithm recursively.
   (b) Add $u$ back and assign to it the first available color not assigned to a vertex at distance 1 or 2 from $u$ in $G$.
   **else** color $S$ with $\Delta(G)^{1.5}$ colors.

**A new radiocoloring heuristic for Split Graphs (*GRcSG*).** Algorithm $GRcSG$ firstly assigns to each vertex of the clique $K$ a distinct color. Next, for each vertex $v$ of the independent set $S$, it finds the first available color not assigned to any vertex of distance at most 2 from $v$ in $G$ and assigns it to $v$.

### 4.2 Generators for Split Graphs

**Random Split Graph 1 (*RS*1) Algorithm.** $RS1$ constructs a split graph utilizing randomness as follows: First, it computes a uniformly random permutation of the vertices of $G$ (i.e., of numbers $1, \ldots, n$). Then, it chooses the first $pn$ vertices of the sequence and joins them to form the clique $K$ of the split graph to be obtained. For the rest of the vertices, consisting the independent set, $S$, it adds an edge between a vertex $u$ and a vertex $v$ for $u \in K$ and $v \in S$ with probability $q = 1 - p$.

**Random Split Graphs 2 (*RS*2) Algorithm.** This generator is similar to the previous one. The only difference is on the way we add edges between the vertices of the clique $K$ and those of the independent set $S$. The objective is to produce split graphs of maximum degree lower than that of the graphs obtained by $RC1$. $RS2$, first, joins each vertex $v$ of the independent set to close to $npq$ vertices of the clique, where $0 \leq q, p \leq 1$. For each such vertex $v \in S$, it chooses a number close to $npq$, call it $\alpha$. Then, it finds $\alpha$ vertices of $K$ with the lowest degrees and connects them to $v$.

**Deterministic Generation Algorithm for Split Graphs of Bodlaender et al. (*BD*).** [4] presented a deterministic generator for split graphs $G$, we called it $BD$, such that $\lambda(G) \geq (1/3)\sqrt{2/3}\Delta(G)^{1.5}$. The same method applies also for chordal graphs, since every split graph is chordal.

The algorithm first creates a clique $K$ of $\Delta(G)/3 + 1$ vertices. It creates an independent set $S$ of size $(1/3)\sqrt{2/3}\Delta(G)^{1.5}$. This set is partitioned into $\sqrt{\frac{2}{3}\Delta(G)}$ groups, each consisting of $\Delta(G)/3$ vertices. Note that the number of distinct pairs of groups with vertices of $S$ is $\left(\sqrt{\frac{2}{3}\Delta(G)^2}\right)/2 = \Delta(G)/3$.

Then, the algorithm construct a split graph $G(V, E)$ using the clique $K$ and the independent set $S$ defined above as follows: For each pair of groups of vertices of $S$, take one vertex of $K$ and join it with each vertex of these two groups. In this way the maximum degree of $G$ is $\Delta(G)$. Note that all the vertices of the resulting graph are at distance at most two apart.

**A variation of Bodlaender's Generation Algorithm ($BV$).** We designed and implemented a variation of $BD$ Algorithm, called $BV$. The modification differentiates only an edge between a vertex of $S$ to a vertex of $K$ is added in the same way as above but this occurs with probability $p$.

### 4.3 Experimental Results for Split Graphs

In this section we present our experimental findings on split graphs. We discuss about the instances of graphs obtained by our generators and that of Bodlaender's et al. [4]. Finally, we present and compare the radiocolorings produced by Bodlaender's algorithm, $BRcSG$ and the heuristic $GRcSG$ we provide here.

**Radiocoloring on Graphs produced by $RS1$.** We evaluate the performance of the greedy radiocoloring heuristic $GRcSG$ on split graphs produced by $RS1$ generator and compare it with the performance of the radiocoloring algorithm $BRcSG$. We have performed extensive experiments for various values of the probability $p$ given to the generator. Recall that with probability $p$ there is an edge between a vertex of the clique $K$ and a vertex of the independent set $S$. The most interesting cases are when the maximum degree of the graph produced is small compared to $n$. So, we concentrate on values of $p$ less than 0.5. Our results (see [2]) showed that the maximum degree of the graphs produced by $RS1$, when $p = 0.4$ is close to $n/2$. $GRcSG$ uses almost $n$ colors to radiocolor these graphs. This number is increased proportional to the size of the clique $K$. On the other hand, the algorithm provided by algorithm $BRcSG$ has much better performance of graphs obtained by the same generator for the same value of $p$.

**Radiocoloring on Graphs produced by $RS2$, $BD$ and $BV$.** The generator $RS2$, produces graphs with lower maximum graph degree than the graphs

**Table 3.** Radiocoloring Split Graphs produced by generators $RS2$ and $BV$ using the algorithm $GRcSG$

| RS2 | | | | | BV | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $n$ | $\Delta$ | $\lvert K\rvert$ | **#Colors** | **p** | $n$ | $\Delta$ | $\lvert K\rvert$ | **#Colors** | **p** |
| 1000 | 140 | 20 | 143 | 0.2 | 289 | 89 | 33 | 165 | 0.3 |
| 1000 | 333 | 40 | 360 | 0.6 | 2177 | 340 | 129 | 774 | 0.3 |
| 2000 | 550 | 40 | 587 | 0.5 | 2177 | 330 | 129 | 724 | 0.3 |
| 2000 | 356 | 40 | 373 | 0.3 | 3079 | 406 | 163 | 942 | 0.3 |
| 3000 | 1001 | 120 | 1086 | 0.6 | 3079 | 479 | 163 | 1928 | |

obtained by $RS1$. The experiments illustrated in Table 3 show that this decrease results in an analogous decrease in the number of colors needed for the radiocoloring of the produced graph. This quantity (i.e. $\lambda(G)$) is estimated applying algorithm $GColSG$ on the graphs obtained by $RS1$. Thus, the founding suggest that a decrease of $\Delta$ in graphs generated by such a random procedure does not help in generating graphs that need a large number of colors in order to be radiocolored.

$BD$ generator is a deterministic one, as we described above. The experiments performed using this generator evaluate the theoretical results, so we do not present them explicitly. Moreover, we evaluate the performance of the variation of $BD$, algorithm $BV$, introduced here. The experiments shown in table 3 show that the graphs produced by $BV$ need less colors than the graphs produced by $BD$, but this number is quite more than the $\Delta(G)$. Thus, the generator produces graph instances that lie between "*extremal* " and "*non-extremal* " characterizations.

## References

1. G. Agnarsson, R. Greenlaw, M. H. Halldorsson. Powers of chordal graphs and their coloring. Congressus Numerantium, (since Sept 2000).
2. M.I. Andreou V.G. Papadopoulou P.G. Spirakis, B. Theodorides and A. Xeros, "Generating and Radiocoloring Families of Perfect Graphs", CRESCCO Technical Report 2004.
3. M.Andreou, S.Nikoletseas and P.Spirakis: Algorithms and Experiments on Colouring squares of Planar Graphs, in Proceddings of Second International Workshop on Experimental and Efficient Algorithms, WEA 2003, LNCS 2647 Springer 2003, pp. 15-32.
4. H. L. Bodlaendera, T. Kloksb, R. B. Tana,c and J. V. Leeuwena. Approximations for $\lambda$-Coloring of Graphs. a Utrecht University, The Netherlands, b Vrije Universiteit, The Netherlands, c University of Sciences & Arts of Oklahoma, OK 73018, U.S.A.
5. D.A. Fotakis, S.E. Nikoletseas, V.G. Papadopoulou and P.G. Spirakis: NP-Completeness Results and Efficient Approximations for Radiocoloring in Planar Graphs. Journal of Theoretical Computer Science (to appear).
6. D. Fotakis and P. Spirakis: Assignment of Reusable and Non-Reusable Frequencies. International Conference on Combinatorial and Global Optimization (1998).
7. M.C.Golumbic. Algorithmic Graph Theory and Perfect Graphs. Academic Press, Inc., 1980.
8. W.K. Hale: Frequency Assignment: Theory and Applications, Proceedings of the IEEE, vol. 68, No. 12, December 1980.
9. K. Mehlhorn and St. Naher. The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, 1999.
10. D. Sakai. Labeling Chordal Graphs: Distance Two Condition, SIAM J. Disc. Math. 7 (1994) pp. 133-140.

# Efficient Implementation of Rank and Select Functions for Succinct Representation

Dong Kyue Kim[1,⋆], Joong Chae Na[2], Ji Eun Kim[1], and Kunsoo Park[2,⋆⋆]

[1] School of Electrical and Computer Engineering, Pusan National University
[2] School of Computer Science and Engineering, Seoul National University
kpark@theory.snu.ac.kr

**Abstract.** Succinct representation is a space-efficient method to represent $n$ discrete objects by $O(n)$ bits. In order to access directly the $i$th object of succinctly represented data structures in constant time, two fundamental functions, `rank` and `select`, are commonly used. However, little efforts were made on analyzing practical behaviors of these functions despite their importance for succinct representations.

In this paper we analyze the behavior of Clark's algorithm which is the only one to support `select` in constant time using $o(n)$-bit space of extra space, and show that the performance of Clark's algorithm gets worse as the number of 1's in a bit-string becomes fewer and there exists a worst case in which a large amount of operations are needed. Then, we propose two algorithms that overcome the drawbacks of Clark's. These algorithms take constant time for `select`, and one uses $o(n)$ bits for extra space and the other uses $n + o(n)$ bits in the worst case. Experimental results show that our algorithms compute `select` faster than Clark's.

## 1 Introduction

To analyze performance of data structures, the processing time and the amount of used storage are measured in general. With the rapid proliferation of information, it is increasingly important to focus on the storage requirements of data structures. Traditionally, discrete objects such as elements of sets or arrays, nodes of trees, vertices and edges of graphs, and so on, are represented as integers which are the indices of elements in a consecutive memory block or values of logical addresses in main memory. If we store $n$ discrete objects in this way, they occupy $O(n)$ words, i.e., $O(n \log n)$ bits.

Recently, a method to represent $n$ objects by $O(n)$ bits, which is called *succinct representation*, was developed. Various succinct representation techniques have been developed to represent data structures such as sets, static and dynamic

⋆ Supported by "Research Center for Logistics Information Technology (LIT)" hosted by the Ministry of Education & Human Resources Development in Korea.
⋆⋆ Supported by MOST grant M6-0405-00-0022.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 315–327, 2005.
© Springer-Verlag Berlin Heidelberg 2005

dictionaries [7, 14] trees [11], and graphs [16, 8]. Moreover, succinct representation is also applied to permutations [10] and functions [13]. Especially, in the areas of string precessing and bio-informatics, to store efficiently enormous DNA sequence data, full-text index data structures are developed such as compressed suffix trees and arrays [12, 5, 15, 6, 4] and the FM-index [2, 3]. If we use these succinctly represented data structures, we can perform very fast pattern searching using small space.

Most succinct representations use `rank` and `select` as their basic functions. The rank and select functions are defined as follows. Given a static bit-string $A$,

- $\texttt{rank}_A(x)$: Counts the number of 1's up to and including the position $x$ in $A$.
- $\texttt{select}_A(y)$: Finds the position of the $y$th 1 bit in $A$.

For example, if $A = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0$, $\texttt{rank}_A(5) = 2$ and $\texttt{select}_A(2) = 4$.

There have been only three results on `rank` and `select`. Jacobson [7, 8] first considered `rank` and `select` and proposed a data structure supporting the functions in $O(\log n)$ time. Jacobson constructs a two-level directory structure and performs a direct access and a binary search on the directory for `rank` and `select`, respectively. Clark [1] improved Jacobson's result to support `rank` and `select` in constant time by adding lookup-tables to complement the two-level directory of Jacobson's. Munro and Raman [11] proposed a three-level directory structure for `rank`. Recently, Miltersen [9] showed lower bounds for `rank` and `select`. For `select`, Clark's data structure is the only one to support `select` in constant time using $o(n)$-bit space of extra space. Despite the importance of `select` for succinct representation, little efforts were made on analyzing its behavior on different kinds of bit strings.

In this paper we analyze the behavior of Clark's algorithm for `select` on various bit-strings and then we get the following results.

- Its performance gets worse as the number of 1's in the bit-string becomes fewer. Especially when the number of 1's in the bit-string is very few (the portion of 1's is less than 10%), Clark's algorithm becomes quite slow.
- In Clark's algorithm, there exists a worst case in which a large number of accesses to lookup tables is needed.
- In addition, it is difficult to implement Clark's algorithm using a byte-based method because the block sizes of Clark's algorithm are varying.

We propose two algorithms that resolve these drawbacks of Clark's algorithm by transforming the given bit-string $A$ into a bit-string where 1's are distributed quite regularly. Our algorithms support `rank` and `select` in constant time, and Algorithm I uses $o(n)$ bits for extra space and Algorithm II uses $n + o(n)$ bits in worst case. However, Algorithm II works fast and uses less space than Algorithm I in practice. Both algorithms have the following properties.

- Our algorithms are not affected by the distribution of 0's and 1's because we partition a bit-string into blocks of constant size to make a directory for `select`. Hence, the algorithms take uniform retrieval time.

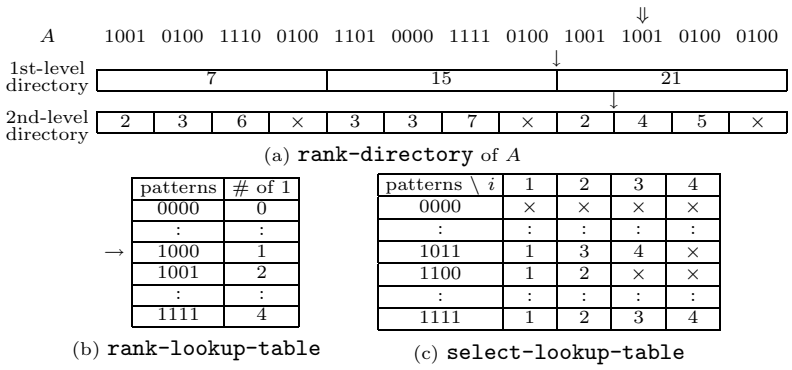



**Fig. 1.** Examples of data structures

- Only a small number of table accesses is always required in our algorithms.
- Our algorithms are easy to implement because directories are composed of constant-sized blocks. Thus, our algorithms can be implemented by a byte-based method which is suitable for modern general-purpose computers.

In experimental results, we show the behavior of Clark's algorithm and show that Algorithm I and Algorithm II resolve the drawbacks of Clark's algorithm. We compare the performance of Clark's algorithm, Algorithm I and II. Our algorithms construct and compute `select` faster than Clark's. Algorithm II using byte-based implementation shows remarkable performance in computing `select`.

## 2 Preliminaries

We first give definitions of some data structures that will be used for `rank` and `select` later. Let $A$ be a static bit-string of length $n$. We denote the $i$th bit by $A[i]$ and the substring $A[i]A[i+1]\cdots A[j]$ by $A[i..j]$. For simplicity, we assume that $\sqrt{\log n}$, $\log n$ and $\log^2 n$ are integers.

We first define a hierarchical directory structure for `rank`. Given a bit-string $A$, we define `rank-directory` of $A$ as the following two-level directory (see Fig. 1 (a)):

- We partition $A$ into *big blocks* of size $\log^2 n$. Each big block of the 1st-level directory records the accumulated number of 1's from the first big block. That is, the $i$th entry contains the number of 1's in $A[1..i\log^2 n]$ for $1 \le i \le \lfloor n/\log^2 n \rfloor$.
- We partition $A$ into *small blocks* of size $\log n$. Each small block of the 2nd-level directory records the accumulated number of 1's from the first small block within each big block. That is, the $i$th entry contains the number of 1's in $A[i'\log^2 n + 1..i\log n]$ for $1 \le i \le \lfloor n/\log n \rfloor$, where $i' = \lfloor i\log n/\log^2 n \rfloor$.

If we have `rank-directory` of $A$, we can get $\mathtt{rank}_A(i)$ for an ending position $i$ of every small block in constant time. For example, $\log^2 n = 16$ and $\log n = 4$

in Fig. 1 (a). Then we can find $\texttt{rank}_A(36) = 17$ by adding the value 15 in the 2nd entry of the 1st-level directory and the value 2 in the 9th entry of the 2nd-level directory, which give the number of 1's in $A[1..32]$ and $A[33..36]$, respectively.

**Lemma 1.** *Given a bit-string $A$ of length $n$,* `rank-directory` *of $A$ can be stored in $o(n)$ bits.*

We define some lookup tables that enable us to compute `rank` and `select` in constant time. For some integer $c > 1$ ($c = 2$ suffices), `rank-lookup-table` is the table where an entry contains the number of 1's in each possible bit pattern of length $(\log n)/c$. Similarly, `select-lookup-table` is the table where an entry contains the position of the $i$th 1 bit in each possible bit pattern of length $(\log n)/c$, for $1 \leq i \leq \log n/c$. Figure 1 (b) and (c) show `rank-lookup-table` and `select-lookup-table` for patterns of length 5, respectively.

**Lemma 2.** *Both* `rank-lookup-table` *and* `select-lookup-table` *can be stored in $o(n)$ bits.*

## 3 Behavior Analysis of Clark's Algorithm

In this section we describe Clark's algorithms [1] for `rank` and `select`, and analyze the behavior of Clark's algorithm for `select`.

### 3.1 Clark's Algorithm

We first describe the algorithm for `rank` and then describe the algorithm for `select`. For $\texttt{rank}_A$, Clark used `rank-directory` of $A$ and `rank-lookup-table`. To get $\texttt{rank}_A(x)$, one first computes $\texttt{rank}_A(i)$ for an ending position $i$ of the small block which includes $A[x]$ using `rank-directory`, and then counts the number of 1's in remaining $x$ mod $\log n$ bits, which can be found by adding at most $c$ entries in `rank-lookup-table` after masking out unwanted trailing bits.

*Example 1.* For bit-string $A$ in Fig. 1, we want to compute $\texttt{rank}_A(38)$. We first get $\texttt{rank}_A(36) = 17$ using `rank-directory` in Fig. 1 (a). We get a bit-pattern 1000 by mask out $A[37..40] = 1001$ with 1100 and get the number of 1's in 1000 using `rank-lookup-table` in Fig. 1 (b). So, we get $\texttt{rank}_A(38) = 18$ by adding 1 to 17.

For $\texttt{select}_A$, Clark used `rank-lookup-table`, `select-lookup-table`, and the following multi-level directory which is stored in $o(n)$ bits.

- The 1st-level directory records the position of every $(\log n \log\log n)$'th 1 bit. That is, the $i$th entry contains $\texttt{select}_A(i \log n \log\log n)$ for $1 \leq i \leq \lfloor n/\log n \log\log n \rfloor$.
- The 2nd-level directory consists of two kinds of tables. Let $r$ be the size of a subrange between two adjacent values in the 1st-level directory and consider this range of size $r$.

  - If $r \geq (\log n \log\log n)^2$, then all answers of `select` in this range are recorded explicitly using $\log n$ bits for each entry.
  - Otherwise, one re-subdivides the range and records the position, relative to the start of the range, of every $(\log r \log\log n)$'th 1 bit.
- Let $r'$ be the size of a subrange between two adjacent values in the 2nd-level directory.
  - If $r' \geq \log r' \log r (\log\log n)^2$, then all answers are recorded explicitly.
  - Otherwise, one records nothing. In this case, `rank-lookup-table` and `select- lookup-table` are used.

Now we describe how to get $\texttt{select}_A(y)$ using the data structures above. To get $\texttt{select}_A(y)$, one first finds the correct 1st-level directory entry, at position $\lfloor y/\log n \log\log n \rfloor$, and compute $r$. If $r \geq (\log n \log\log n)^2$, one can find the value of $\texttt{select}_A(y)$ in the 2nd-level directory. Otherwise, a similar search is performed in the 2nd-level directory resulting in either finding the answer from the 3rd-level directory or scanning a small number of bits using lookup-tables. It was proven that the length of a bit-string scanned using lookup-tables is less than $16(\log\log n)^4$ [1]. So one can perform `select` on a range of $16(\log\log n)^4$ bits using a constant number of accesses to the lookup-tables.

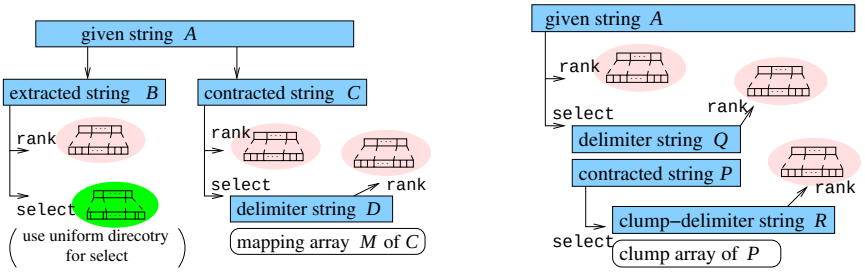
### 3.2 Behavior of Clark's Algorithm

We analyze the behavior of Clark's algorithm for `select` on various bit-strings and then we get the followings.

- The retrieval time of Clark's algorithm for `select` varies according to distribution of 0's and 1's in bit-strings. The performance gets worse as the number of 1's in the bit-string becomes fewer. Especially when the number of 1's in the bit-string is very few (the portion of 1's is less than 10%), Clark's algorithm becomes quite slow.
- In Clark's algorithm, there exists a worst case in which a large number of accesses to lookup-tables is needed. The reason is that $16(\log\log n)^4$ is much larger than $\log n$ in practice although the former is asymptotically smaller than the latter. For example, in case that the length of a bit-string is $2^{32}$, the size of a block for accessing tables is 4096 and the number of table accesses is 256 in the worst case. The number 256 is too big to regard as a constant in practice.
- The structure of directories is irregular because the subranges $r$ and $r'$ of entries vary. It makes it difficult to implement the algorithm using a byte-based implementation method.

## 4 Algorithm I

In this section we present our first algorithm that overcomes the drawbacks of Clark's algorithm for `select`. This algorithm also adopts the approach using




(a) Algorithm I (b) Algorithm II

**Fig. 2.** Structures of Algorithms I and II

multi-level directories and lookup-tables. The difficulty of developing an algorithm for `select` results from the irregular distribution of 1's. While Clark's algorithm overcomes the difficulty by classifying the subranges of directories into two groups: dense one and sparse one, we do it by transforming $A$ into a bit-string where 1's are distributed quite regularly.

### 4.1 Definitions

Given a bit-string $S$ of length $m$, we divide $S$ into blocks of size $b$. There are two kinds of blocks. One is a block where all elements are 0 and the other is a block where there is at least one 1. We call the former *zero-block (ZB)* and the latter *nonzero-block (NZ)*.

- The *contracted* string of $S$ is defined as a bit-string $S_c$ of length $m/b$ such that $S_c[i] = 0$ if the $i$th block of $S$ is a zero-block, $S_c[i] = 1$ otherwise.
- The *extracted* string of $S$ is defined as a bit-string $S_e$ which is formed by concatenating nonzero-blocks of $S$ in order. Hence, the length of $S_e$ is $m$ in the worst case, and the distance between the $i$th 1 bit and the $j$th 1 bit is at most $(j - i + 1)b - 1$ for $i < j$.
- The *delimiter* string of $S$ is defined as a bit-string $S_d$ such that $S_d[i] = 0$ if the $i$th 1 bit and the $(i-1)$st 1 bit of $S$ are contained in the same block, and $S_d[i] = 1$ otherwise. We define $S_d[1] = 1$. Note that the length of $S_d$ is equal to the number of 1's in $S$ and so it is $m$ in the worst case. The value of $\texttt{rank}_{S_d}(i)$ means the number of nonzero blocks up to the block (including itself) containing the $i$th 1 bit of $S$.

*Example 2.* Bit-strings $S$, $S_c$, $S_e$ and $S_d$. We assume that we divide $S$ into blocks of size 4.

| $S_c$ | 1 | | | | 0 | | | | 1 | | | | 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| $S_e$ | 0 | 1 | 1 | 1 | | | | | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| $S_d$ | | 1 | 0 | 0 | | | | | | | 1 | | 1 | 0 | | 0 |

### 4.2 Skeleton of Algorithm I

Let $B$ be the extracted string of $A$, and $C$ be the contracted string of $A$ when dividing $A$ into blocks of size $\sqrt{\log n}$. We transform $A$ into $B$ and $C$, and compute $\texttt{rank}_A$ and $\texttt{select}_A$ using $B$ and $C$. We first consider the properties of $B$ and $C$ and describe how to compute $\texttt{rank}_A$ and $\texttt{select}_A$ using $\texttt{rank}$'s and $\texttt{select}$'s of $B$ and $C$. In the next section we present algorithms and data structures for $\texttt{rank}$'s and $\texttt{select}$'s of $B$ and $C$.

Recall that blocks of $B$ contain at least one 1 bit, and thus the distance between the $i$th 1 bit and the $j$th 1 bit in $B$ is at most $(j-i+1)\sqrt{\log n}-1$ for $i<j$. Bit-string $C$ represents a mapping between $A$ and $B$. The length of $C$ is $n/\sqrt{\log n}$ and the length of $B$ is $n$ in the worst case. See Fig. 2 (a).

*Example 3.* Bit-strings $A$, $B$ and $C$. We assume that $\sqrt{\log n}=5$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | ↓ | ⇓ |
| $A$ | 01101 | 00000 | 00000 | 11010 | 00000 | 01001 | 00100 |
| $C$ | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| $B$ | 01101 | 11010 | 01001 | 00100 | | | |
| | | | ↑ | ⇑ | | | |

We describe how to compute $\texttt{rank}_A(x)$ and $\texttt{select}_A(y)$ using $B$ and $C$. For computing $\texttt{rank}_A(x)$, we find the index $x'$ of $B$ which corresponds to $A[x]$ and compute $\texttt{rank}_B(x')$. Let $x_b$ be the block number of $A$ which contains $A[x]$ and $x_p$ be the position of $A[x]$ in the $x_b$th block of $A$, that is, $x_b=\lceil x/\sqrt{\log n}\,\rceil$ and $x_p=x-(x_b-1)\sqrt{\log n}$. Then,

$$x'=\texttt{rank}_C(x_b-1)\times\sqrt{\log n}+x_p\times C[x_b] \quad \text{and} \quad \texttt{rank}_A(x)=\texttt{rank}_B(x').$$

For computing $\texttt{select}_A(y)$, we compute $\texttt{select}_B(y)$ and find an index of $A$ which corresponds to $B[\texttt{select}_B(y)]$. Let $s_b$ be the block number of $B$ which contains the $y$th 1 bit, that is, $s_b=\lceil \texttt{select}_B(y)/\sqrt{\log n}\,\rceil$. Then,

$$\texttt{select}_A(y)=\texttt{select}_B(y)+(\texttt{select}_C(s_b)-s_b)\sqrt{\log n}.$$

*Example 4.* For $A$ of Example 3, suppose that we want to compute $\texttt{rank}_A(28)$ and $\texttt{select}_A(9)$. For $\texttt{rank}_A(28)$, we get $x_b=6$ and $x_p=3$, that is, $A[28]$ is the 3rd bit in the 6th block of $A$. Because $\texttt{rank}_C(5)=2$, we get $x'=2\times5+3=13$, so $\texttt{rank}_A(28)=\texttt{rank}_B(13)=6$. For $\texttt{select}_A(9)$, we get $\texttt{select}_B(9)=18$, so $s_b=4$. Because $\texttt{select}_C(4)=7$, we get $\texttt{select}_A(9)=18+(7-4)\times5=33$.

### 4.3 Ranks and Selects for $B$ and $C$

For $\texttt{rank}_B$, we build $\texttt{rank-directory}$ of $B$ and $\texttt{rank-lookup-table}$. For $\texttt{rank}_C$, we build $\texttt{rank-directory}$ of $C$ and $\texttt{rank-lookup-table}$. We can compute $\texttt{rank}_B(x)$ and $\texttt{rank}_C(x)$ using these data structures in constant time as in Section 2.

**Select for extracted string $B$.** For $\texttt{select}_B$, we use lookup-tables and a two-level directory which is a little different from $\texttt{rank-directory}$.

- The 1st-level directory records the position of every $\log^2 n$'th 1 bit in $B$. This directory has at most $n/\log^2 n$ entries and each entry requires $\log n$ bits. Thus the space of this directory is $n/\log^2 n \times \log n = o(n)$ bits.
- The 2nd-level directory records the position of every $\sqrt{\log n}$'th 1 bit in the ranges of 1st-level directory. This directory has at most $n/\sqrt{\log n}$ entries and each entry requires $\log(\log^2 n \times \sqrt{\log n})$ bits because all blocks of size $\sqrt{\log n}$ have at least one 1 bit. Thus the space of this directory is $n/\sqrt{\log n} \times 5/2 \times \log\log n = o(n)$ bits.
- We also maintain `rank-lookup-table` and `select-lookup-table`.

We can compute $\texttt{select}_B(y)$ by accessing the directory and the lookup-tables as in `rank`. Because the 2nd-level directory records the position of every $\sqrt{\log n}$'th 1 bit, we need to scan a substring of at most length $\sqrt{\log n} \times \sqrt{\log n}$ using the lookup-tables. It can be done by accessing the lookup-tables at most $c$ times.

**Select for contracted string $C$.** We use a different approach for $\texttt{select}_C$ because the range of contiguous 0's is not bounded in $C$. Let $D$ be the delimiter string of $C$ when dividing $C$ into blocks of size $\log n$. The length of $D$ is $n/\sqrt{\log n}$ in the worst case. See Example 5. We construct the following auxiliary data structures.

- We construct a data structure for $\texttt{rank}_D$. It consists of `rank-directory` of $D$ and `rank-lookup-table`. The value of $\texttt{rank}_D(i)$ means the number of nonzero blocks of size $\log n$ up to the block (including itself) containing the $i$th 1 bit of $C$.
- We construct an array $M$ whose $i$th entry represents the block number of the $i$th nonzero-block in $C$. We call $M$ the *mapping* array of $C$. This array has at most $n/(\log n\sqrt{\log n})$ entries and each entry requires $\log(n/(\log n\sqrt{\log n}))$ bits. Thus the space of this array is $n/(\log n\sqrt{\log n}) \times \log\left(n/(\log n\sqrt{\log n})\right) = o(n)$bits.

*Example 5.* Bit-strings $C$ and $D$, and the mapping array $M$ of $C$. We assume that $\log n = 4$.

| $C$ | 0 1 1 1 | 0 0 0 0 | 0 0 1 0 | 1 1 0 1 |
|---|---|---|---|---|
| $D$ | 1 0 0 | | 1 | 1 0 0 |

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $M$ | 1 | 3 | 4 |

In order to get $\texttt{select}_C(y)$, we first compute the block number containing the $y$th 1 bit in $C$ using $\texttt{rank}_D$ and array $M$. Then, we can find the position of the $y$th 1 bit in $C$ by scanning the bit-string of this block using the lookup-tables.

*Example 6.* Suppose that we want to find $\texttt{select}_C(6)$. The `rank` of the 6th bit in $D$ is 3 and the value of the 3rd entry in array $M$ is 4. Thus, the 6th 1 bit is in the 4th block of $C$. We can also know that the first 3 blocks of $C$ contains four 1 bits using $\texttt{rank}_C$. The final job is to find the 2nd 1 bit in the 4th block using the lookup-tables.

**Theorem 1.** *Algorithm I performs* $\texttt{rank}_A$ *and* $\texttt{select}_A$ *in constant time using* $o(n)$ *bits of extra space.*

# 5 Algorithm II

In this section we describe Algorithm II, which is simpler and more practical than Algorithm I but theoretically uses $n + o(n)$ bits of extra space. We also propose a new implementation method which is suitable for modern general-purpose computers.

## 5.1 Description of Algorithm II

We only describe the algorithm for $\texttt{select}_A$ since the algorithm for $\texttt{rank}_A$ is the same as Clark's. For $\texttt{select}_A$, we use an approach similar to that used for $\texttt{select}_C$ in Section 4.3. When dividing $A$ into blocks of size $\log n$, let $P$ and $Q$ be the contracted string and the delimiter string of $A$, respectively. The length of $P$ is $n/\log n$ and the length of $Q$ is $n$ in the worst case.

We can compute $\texttt{select}_A(y)$ using $\texttt{rank}_Q$ and $\texttt{select}_P$. Because $\texttt{rank}_Q(y)$ gives the number of nonzero blocks up to the block (including itself) containing the $y$th 1 bits of $A$ and $\texttt{select}_P(k)$ represents the block number of the $k$th nonzero block, $\texttt{select}_P(\texttt{rank}_Q(y))$ is the block number containing $y$th 1 bit in $A$. Then, we can find the position of the $y$th 1 bits in $A$ by scanning the bit-string of this block using the lookup-tables. For $\texttt{rank}_Q$, we build `rank-directory` of $Q$. For $\texttt{select}_P$, we use a new approach.

We describe an approach for $\texttt{select}_P$ which uses small space in practice. We call a bundle of contiguous 0's a *clump*. In Example 7, there are 4 clumps in $P$. We define the *clump-delimiter* string $R$ of $P$ as follows: $R[i] = 0$ if the $i$th 1 bit of $P$ is adjacent to the $(i-1)$st 1 bit, $R[i] = 1$ otherwise. We define $R[1]$ as 0 if $P[1] = 1$, and 1 otherwise. See Example 7. The length of $R$ is $n/\log n$ in the worst case. We construct the following auxiliary data structures.

- We construct a data structure for $\texttt{rank}_R$. The value of $\texttt{rank}_R(i)$ means how many clumps there are in front of the $i$th 1 bit of $P$. In Example 7, there are 3 clumps in front of the 7th 1 bit.
- We construct an array where the $i$th entry represents the accumulated number of 0's up to the $i$th clump (including itself) in $P$. We call it the *clump* array of $P$. In practice, most elements of $P$ are 1 because $P$ is a delimiter string. Therefore, there are a few clumps in $P$ and so the size of this table is very small. Note that we do not need to maintain bit-string $P$.

*Example 7.* Bit-strings $P$ and $R$, and the clump array of $P$.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P$: | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | |
| $R$: | | | 1 | 0 | 0 | | 1 | 0 | | | 1 | 0 | 0 | 0 | | 1 | |

The clump array of $P$

| index | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| | 2 | 3 | 5 | 6 |

In order to get $\texttt{select}_P(i)$, we first know how many clumps there are in front of the $i$th 1 bit of $P$ using $\texttt{rank}_R(i)$ and compute the number $j$ of 0's in front of the $i$th 1 bit using the clump array . Then $\texttt{select}_P(i) = i + j$.

**Theorem 2.** *Algorithm II performs* $\texttt{rank}_A$ *and* $\texttt{select}_A$ *in constant time using* $o(n)$ *and* $n + o(n)$ *bits of extra space, respectively.*

### 5.2 Byte-Based Implementation of Algorithm II

We present an efficient implementation method of Algorithm II. The directories and lookup-tables are based on bits while atomic units in modern computers are not bits but bytes. Therefore, we need *bit-operations* such as *bitwise-and, bitwise-or*, and *shift* in order to get the values of entries. It causes inefficiency in time. One method avoiding such inefficiency is to allocate space to entries by the units of bytes. For example, we allocate 2 bytes to an entry which requires 12 bits. However, this method may waste much space. We present a byte-based implementation method reducing waste of space and avoiding bit-operations. This method is applied to $Q$ and $R$ as well as $A$. We assume that $n$ is less than $2^{32}$, the maximum value represented by a word in 32-bit machines. However, our method can be extended to apply to the case of $n \geq 2^{32}$.

The key idea of our method is that ranges of directories and lookup-tables are adjusted to multiples of 8. The following data structure, which we use to implement Algorithm II, is an instance of our implementation method.

- The 1st-level directory contains `rank` for every multiple of $2^8$. Each entry requires at most 32 bits. Particularly, the ranges of 1st-level directory is adjusted to $2^k$, where $k$ is a multiples of 8 in order to allocate space to entries of 2nd-level directory by byte units.
- The 2nd-level directory contains `rank`, for every multiple of $2^5$, within the subranges of size $2^8$. Each entry requires 8 bits.
- For each possible bit pattern of length 8, the rank-lookup-table gives the number of 1's in the pattern. Each entry requires 3 bits but we allocate 8 bits for each entry in order to avoid bit-operations. We may access the rank-lookup-table four times to get `rank`.

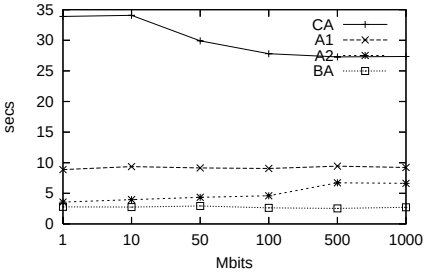
Because each entry in each directory is stored in bits of a multiple of 8, we can find values of entries without bit-operations. So retrievals in our method are fast.

## 6 Experimental Results

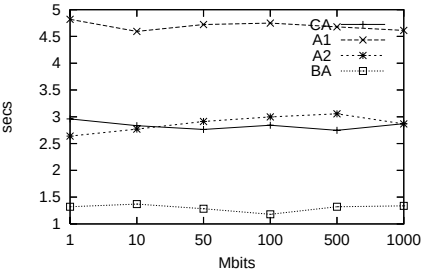
We present experimental results comparing Clark's algorithm (CA) with Algorithm I (A1), Algorithm II (A2), and byte-based Algorithm II (BA). All algorithms except algorithm BA are allowed to use bit-operations.

We used Microsoft Visual C++ 6.0 to implement the algorithms and performed these experiments on the 2.8Ghz Pentium IV with 4GB main memory. We measured retrieval time of `rank` and `select`, construction time, and space of auxiliary data structures. We performed experiments on bit-strings where the ratio of 1's is about 3 % and random bit-strings.

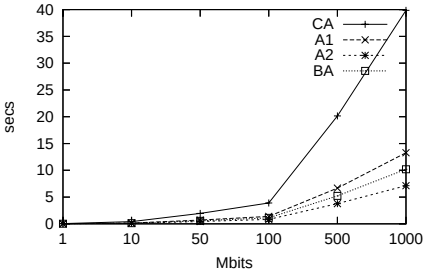
**Experiment 1** (bit-strings with 3 % 1's)**:** This experiment shows the drawbacks of Clark's algorithm pointed out in Section 3.2. Figure 3 (a) and (b) show retrieval time of `select` and `rank`, respectively. In these figures, the vertical axis represents the time taken to perform $10^7$ random queries and the horizontal axis represents the length of bit-string $A$. In `select`, our algorithms are about $3 \sim 10$



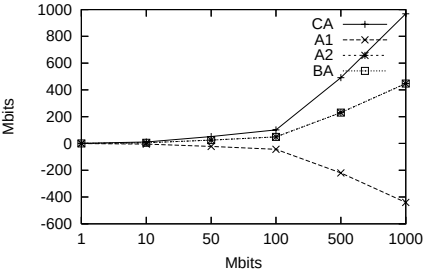

(a) `select` retrieval time




(b) `rank` retrieval time

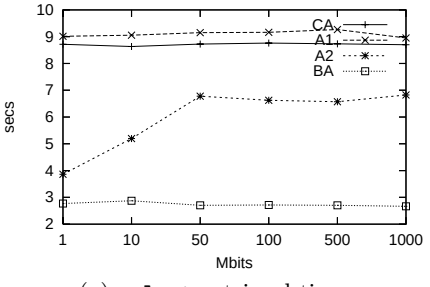



(c) construction time

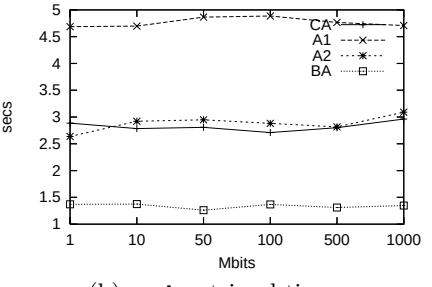



(d) space

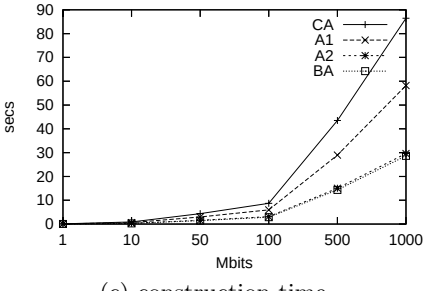
**Fig. 3.** Experimental results on bit-strings with 3 % 1's



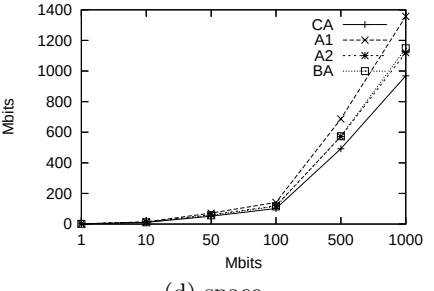

(a) `select` retrieval time




(b) `rank` retrieval time




(c) construction time




(d) space

**Fig. 4.** Experimental results on random bit-strings

times faster than Clark's. In `rank`, Algorithm I is the slowest. The reason is that it needs two `rank`'s. Algorithm II and Clark's algorithm have the same performance in `rank` because two algorithms for `rank` are the same. Figure 3 (c) shows construction time. Clark's algorithm is the slowest due to the complication and irregularity of the data structures. Our algorithms are about $3 \sim 5$ times faster than Clark's. Figure 3 (c) shows space of auxiliary data structures. We do not count the space for a given string $A$. Clark's algorithm requires the most space. The reason why Algorithm I has negative values is that a transformed string $B$ is even shorter than given string $A$.

**Experiment 2** (random bit-strings): This experiment shows general performance of the algorithms. Figure 4 (a) and (b) show retrieval time of `select` and `rank`, respectively. The fact that byte-based Algorithm II is the fastest in both tells efficiency of byte-based implementation in time. In `select`, the performance of Algorithm I and Clark's algorithm are similar and they are the slowest. The reason why Clark's Algorithm is slow is that it needs many accesses to lookup-tables. In every algorithm, retrieval time of `select` is slower than that of `rank`. Figure 4 (c) and (d) show construction time and space of auxiliary data structures, respectively. Our algorithms are about $1.5 \sim 2.5$ times faster than Clark's. Clark's algorithm requires less space than ours. However, the difference on space is not large compared to the differences on retrieval time and construction time.

## References

1. D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, 1988.
2. P. Ferragina and G. Manzini. Opportunistic data structures with applications. *FOCS'00*, pages 390–398, 2000.
3. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. *SODA'01*, pages 269–278, 2001.
4. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. *SODA'03*, pages 841–850, 2003.
5. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *STOC'00*, pages 397–406, 2000.
6. W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *FOCS'03*, pages 251–260, 2003.
7. G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1988.
8. G. Jacobson. Space-efficient static trees and graphs. *FOCS'89*, pages 549–554, 1989.
9. P. B. Miltersen. Lower bounds on the size of selection and rank indexes. *SODA'05*, pages 11–12, 2005.
10. J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. *ICALP'03*, pages 345–356, 2003.
11. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. on Comp.*, 31(3):762–776, 2001.
12. J. I. Munro, V. Raman, and S.S. Rao. Space efficient suffix trees. *J. Alg.*, 39(2):205–222, 2001.

13. J. I. Munro and S. S. Rao. Succinct representations of functions. *ICALP'04*, pages 1006–1015, 2004.
14. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. *ICALP'03*, pages 357–368, 2003.
15. K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. *SODA'02*, pages 225–232, 2002.
16. G. Turan. Succinct representations of graphs. *Disc. App. Math.*, 8:289–294, 1984.

# Comparative Experiments with GRASP and Constraint Programming for the Oil Well Drilling Problem

Romulo A. Pereira, Arnaldo V. Moura, and Cid C. de Souza

Institute of Computing, University of Campinas

**Abstract.** Before promising locations become productive oil wells, it is often necessary to complete drilling activities at these locations. The scheduling of such activities must satisfy several conflicting constraints and attain a number of goals. Here, we describe a Greedy Randomized Adaptive Search Procedure (GRASP) for the scheduling of oil well drilling activities. The results are compared with those from a well accepted constraint programming implementation. Computational experience on real instances indicates that the GRASP implementation is competitive, outperforming the constraint programming implementation.

## 1 Introduction

Oil extracted from oceanic basins is an increasingly important fraction of the total world offer of petroleum and gas. Usually, diverse petroliferous basins are explored, each with hundreds of promising spots where productive oil wells could be located. However, before these places are turned into productive wells they must be developed, that is, a sequence of engineering activities must be completed at each promising spot, to render them ready for oil extraction. Oil derricks and ships are used to complete these activities. These resources are limited and expensive, either in acquisition or rent value, and must be used efficiently.

The oil *well drilling problem* (WDP) can be summarized thus: given a set of promising spots, the activities to be executed at each location, and the available resources, find a scheduling of the activities and resources, fulfilling several conflicting engineering and operational constraints, in such a way as to optimize some objective criteria. In this work, the specific WDP faced by Petrobras (a leading company in deep water oil extraction) is studied. This WDP imposes much more realistic constraints than other similar studies [1]. The constraints are presented in detail, and an heuristic strategy is developed in order to maximize oil production within a given time horizon.

The next section describes the WDP. Section 3 discusses a GRASP implementation for the WDP. Section 4 presents our computational results obtained with this new algorithm and compares them to other results derived from a constraint programming implementation presently running at Petrobras. Finally, some concluding remarks are offered in the last section.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 328–340, 2005.
© Springer-Verlag Berlin Heidelberg 2005

## 2 The Well Drilling Problem

After a well is drilled, the preparation for oil extraction develops in several stages. First, oil derricks place Wet Christmas Trees (or WCTs, structures where hydraulic valves are attached) at the mouth of the wells in order to avoid oil leakage. Later, boats connect pipelines between WCTs and manifolds. Manifolds are metallic structures installed by boats at the sea floor. Their use prevents the need for exclusive pipelines connecting each well to the surface, which would be prohibitively expensive. Once this stage is completed, oil extraction can begin. For that, Stationary Units of Production (SUPs) are anchored at specific locations in the surface, and boats interconnect manifolds to them. SUPs are used to process, and possibly store, the extracted products. Later, ships fetch the products from SUPs to land storage sites or other processing units. If the oil outflow is very high or a SUP does not have storage capacity, a petroliferous platform may be installed at the surface.

The constraints involved in the scheduling of oil development activities are:

C1. *Technological Precedence*: sets an order between pairs of activities. When considering the precedence between the start and finish of the activities in each pair, any of the four possibilities can be present.
C2. *Mark-Activity*: an activity must finish before or initiate after a fixed date, with or without lag time. This date is often related to some external event.
C3. *Baseline*: sets the start date of the activities.
C4. *Use of Resources*: to execute an activity, due to its intrinsic nature, a resource used must match some operational characteristics. For a boat, it must be verified if the on-board equipments can operate at the specified depth. For an oil derrick, its type and capabilities must be verified, as well as the maximum and minimum depth of operation and drilling.
C5. *Concurrence*: two activities at the same well, or executed by the same resource, cannot be simultaneous.
C6. *Unavailability*: resources may be unavailable for a period of time, either for maintenance reasons or due to contract expiration.
C7. *User Defined Sequences*: the user can specify a sequence for the drilling or for the "start production" activities of different wells. These sequences are specified by engineers in order to avoid loss of pressure in the oil field. If well $A$ appears before well $B$ in the sequence, then activity $F_A$ of well $A$ must finish before the start of activity $S_B$ of well $B$. The activities $F_A$ and $S_B$ are either the activity of drilling or the activity of start production of their respective wells, according to the type of the sequence.
C8. *Surface Constraints*: represented by a polygonal security area defined around a well. When a well is inside the restricted area of another well, activities executed at them cannot be simultaneous. These constraints must be verified between pairs of mobile and pairs of mobile and anchored oil derricks.
C9. *Cluster Constraints*: an activity can be part of a cluster, which is a set of activities that must use the same resource.
C10. *Same Derrick*: it is desirable that the same oil derrick executes as much of the activities at a well as possible, in order to avoid unnecessary displacements.

The oil yield is calculated as follows. Each well has an associated outflow and an activity that marks the beginning of its production. When this last activity is concluded, the well is considered in production. The yield is obtained by multiplying the oil outflow by the period between that instant and the established time horizon. If the start production activity is set for after the time horizon, the corresponding yield is disregarded. The objective is to obtain a schedule of all activities, satisfying all constraints, while maximizing the oil yield.

Other goals to be attained by automating the schedule of the activities are:

1. *Faster solutions.* Human made solutions take many hours, even days, to be constructed. A faster method would permit the analysis of different scenarios for the same problem, for example, by adding or removing resources. Furthermore, modifications in already committed plans would not result in new hours, or days, spent in rescheduling.
2. *Better resource allocation.* With an automated scheduling, all highly skilled engineers responsible for the manual scheduling can receive other duties.

From the above description, it can be seen that the WDP is a difficult combinatorial problem. In fact, it is simple to devise a polynomial-time reduction to the classical Job Shop Scheduling problem, showing that the WDP is NP-hard.

The WDP treated here shows several differences from similar problems studied in the literature [1]. To tackle the same problem, a project team from Petrobras developed a Constraint Programming (cf., [2]) model using ILOG's Solver and Scheduler [3]. After four years of development and testing, the tool, named ORCA (Portuguese acronym for "Optimization of Critical Resources in the Production Activity"), became operational and very successful. Nowadays, the ORCA solver is often used by engineers both to define a good schedule for the drilling activities and, also, to analyze the need for acquiring or renting new resources. They confirmed that ORCA generates better solutions than those made by humans. In one real instance, ORCA showed that buying a third oil derrick was unnecessary and that it was better to add a new LSV ship instead. As a result, Petrobras avoided a expenditure of US$ 15 million, while anticipating oil production by 26 days. Despite the good performance of ORCA, searching for even better solutions is still important, since a tenth of a percent of improvement in the oil production may represent millions of dollars in the company's revenue. The next sections show how we obtain such gains using GRASP.

## 3 A GRASP for the WDP

Our search for alternatives to compete with ORCA started with an implementation of Tabu Search [4] for a simpler version of the WDP [1]. However, some issues proved to be particularly difficult to treat, especially the definition of an adequate neighborhood and ways to explore it. After some investigation, GRASP [5] seemed most appropriate for the WDP. Contrary to what occurs with other metaheuristics, such as tabu search or genetic algorithms, which use

a large number of parameters in their implementations, the basic GRASP version requires the adjustment of fewer parameters. Despite its simplicity, GRASP is a well studied metaheuristic which has been successfully applied to a wide variety of optimization problems (cf. [6]). In particular, applications of GRASP to scheduling problems can be found in [7, 8, 9, 10, 11].

The next paragraphs review some GRASP basics and describe our specific implementation designed to solve the WDP, named GRASP-WDP (GRASPW). The model and its algorithms are shown in the subsequent paragraphs.

**GRASP Basics.** In the GRASP methodology each iteration consists of two phases: *construction* and *local search* [5]. Figure 1 illustrates a generic implementation of GRASP, in pseudo-code. The input includes parameters for setting the candidate list size (*ListSize*), the maximum number iterations (*MaxIter*), and the seed (*Seed*) for the random number generator. The iterations are carried out in lines 2-6. Each iteration consists of the construction phase (line 3), the local search phase (line 4) and, if necessary, the incumbent solution update (line 5). In the construction phase, a feasible solution is built, updating the variable *Solution*. Then the local search algorithm seeks a better solution in the neighborhood of *Solution*, according to a given criterion, and updates *Solution*. This process of construction, search and update is executed *MaxIter* times.

```
1: procedure GRASP(ListSize, MaxIter, Seed)
2: for k = 1 a MaxIter do
3:    Solution ← Construct_Solution(ListSize, Seed);
4:    Solution ← Local_Search(Solution);
5:    Update_Solution(Solution, Best_Solution_Found);
6: end for
7: return Best_Solution_Found;
8: end GRASP
```

**Fig. 1.** Pseudo-code of the GRASP Metaheuristic

In the construction phase, a feasible solution is built one element at a time. Figure 2 illustrates a generic implementation of the construction phase, in pseudo-code. Input includes the candidate list size (*ListSize*) and the seed (*Seed*). The iterations are carried out in lines 2-8. At each iteration, the next element to be added is determined by adding all possible elements to a candidate list, ordered with respect to a greedy function that measures the, maybe myopic, benefit of selecting each element. This list is called the *Restricted Candidate List* (RCL). The adaptive component of the heuristic arises from the fact that the benefits associated with every element are updated at each iteration to reflect the changes brought on by the selection of the element in the previous iteration. The probabilistic component is present by the random choice of one of the best candidates in the RCL, but usually not the best one. This way of choosing elements allows for different solutions to be obtained at each iteration, while not

necessarily jeopardizing the adaptive greedy component. The solutions generated by the construction phase are not guaranteed to be locally optimal. Hence, it is almost always beneficial to apply a local search to attempt to improve each constructed solution. The search phase is a standard deterministic local search algorithm that seeks to optimize the solution built in the construction phase.

```
 1: procedure Construct_Solution(ListSize, Seed)
 2: Solution ← 0;
 3: Evaluate the incremental costs of the candidate elements;
 4: while Solution is not a complete solution do
 5:    Build the restricted candidate list, RCL(ListSize);
 6:    Select an element s from the RCL at random;
 7:    Solution ← Solution ∪ {s};
 8:    Reevaluate the incremental costs;
 9: end while
10: return Solution;
11: end Construct_Solution
```

**Fig. 2.** Pseudo-code of the Construction Phase of GRASP

**The GRASPW Implementation.** The GRASPW implementation was constructed using the C/C++ programming language. The heuristic uses two types of integer variables. One represents the beginning of execution of each activity in the corresponding well. These values range between a minimum and a maximum start time, with those values depending on the current partial solution being constructed. The second type of variables represents which resource will execute each activity in its well. Their domains are characterized by a set of the possible resources, of whose one must be chosen to execute the activity. All the constraints described in Section 2 were enforced. Three constraints, namely, C2, C3 and C4, were set while reading the problem data, before the search begins. Note that, in these cases, all values needed to set the constraints are already defined. The other constraints were dealt with during the search for solutions, the variables involved being assigned single values.

The following adaptations were made to the procedure illustrated in Figure 1: (i) the search procedure was interrupted by a time limit instead of by the number of iterations; and (ii) During a complete run of the GRASPW heuristic, the value of *ListSize* can be monotonically incremented by a fixed amount when a predefined interval of time is reached with no improvement on the best solution. Doing so, the algorithm will explore larger regions of the search space. Alternatively, during a run of GRASPW, the value of *ListSize* can be monotonically decremented between iterations, thus focusing into a greedier heuristic. With this scheme we obtain a *dynamic sized RCL* in opposition to the original *static sized RCL*. Note that, as GRASP iterations are independent, one could think that there is no difference between increasing and decreasing the RCL size. However, as we do not know in advance the amount of time the algorithm will

execute at each run or when the RCL size will be altered, we can not anticipate the result of a GRASPW run, when increasing or decreasing the RCL size.

As in the ORCA implementation, we seek solutions with the highest oil yield. To this end the construction phase illustrated in Figure 2 was modified thus:

1. The first time ever the construction phase is initiated, we use *ListSize* equal to one, when the algorithm behaves like a pure greedy heuristic. With few constraints obstructing the greedy heuristic, it tends to generate a good or even very good solution. For example, in two out of twelve real instances, the best solution was found in the first pass of the construction phase.
2. Before line 3, we added a function called *isPressed*. It verifies if any activity of any well has a start time with a small domain and if only one resource can do it at that time. By a start time with a small domain we mean that its minimum and maximum values are very close, such that having only one resource able to execute it indicates that there is almost no flexibility to schedule the activity. If there are such activities, the function schedules their wells. Retarding the schedule of such activities could render the solution infeasible, as other activities may occupy the period of time where those activities would be scheduled. We schedule the wells, and not only the activities, in order to comply with constraint C10.
3. The candidates are defined by the production wells that are available (meaning that there are no wells yet not scheduled which must precede them), or the injection wells that have activities of production wells succeeding them. The activities of injection wells that do not have activities of production wells succeeding them are left to be scheduled after all others. Note that injection wells are not productive and therefore must not be scheduled before production wells, unless there are constraints forcing such a schedule.
4. The evaluation of incremental costs (line 3 of Figure 2) assesses how much oil a well can offer until the end of the time horizon. The RCL is built with those wells that offer the highest yields of oil. Actually, not only the oil offer is considered, but also the oil offer of the constrained successors of that well.
5. In the construction phase, the next element to be introduced in the solution is chosen uniformly from the candidates in the RCL (line 5 of Figure 2). However, any probability distribution can be used to bias the selection.We tried to bias the selection of the candidates proportionally to their oil offer.
6. To schedule the candidate well (line 7 of Figure 2), proceed as follows.
   As long as there are activities not yet scheduled in the well: (i) choose any activity available in the well, *i.e.*, one not yet scheduled and such that there is no other one not yet scheduled in the wells that must precede it; (ii) choose a resource for this activity that can execute it, and that can complete the activity the earliest; (iii) set the start time of the activity at the earliest possible time, *i.e.*, the maximum between the earliest time the resource is available to execute the activity and the minimum start time of the activity; and (iv) all activities that are constrained to succeed the chosen one must have their minimum start times updated to satisfy any constraints.
   The scheduling of a well is done so as to satisfy all constraints, including the seven ones not yet enforced. In case of violations, and this can be tested after

each activity is scheduled, the construction of this solution is aborted and a new one is started. Instead, we could backtrack a few steps, but this would slow down this phase, especially if the first steps were not appropriate.

7. After a well is scheduled, any activities that must succeed it has their minimum start time updated to satisfy any constraints. If that is not possible, the construction of this solution is also aborted.

For the search phase, an appropriate neighborhood was defined, so as to permit explorations quickly leading to better solutions. The 2-exchange local search algorithm based on the disjunctive graph model [12] was used. The same neighborhood was used in [11] for a Job Shop Scheduling problem. In order to apply the 2-exchange local search to the WDP, we swap two elements in the scheduling. For example, if in resource $X$ the scheduling was ... $\rightarrow X1 \rightarrow A \rightarrow X2 \rightarrow$ ... and in resource $Y$ it was ... $\rightarrow Y1 \rightarrow B \rightarrow Y2 \rightarrow$ ..., where $\rightarrow$ represents a conjunctive arc, the result of the swap would be a schedule like ... $\rightarrow X1 \rightarrow B \rightarrow X2 \rightarrow$ ... and ... $\rightarrow Y1 \rightarrow A \rightarrow Y2 \rightarrow$ ..., in resources $X$ and $Y$, respectively. Since the execution time of elements $A$ and $B$ can be different, all activities after them may have their start times updated.

We need to decide, of course, what an element stands for. Some options are: (i) *An activity*: with very small granularity, giving rise to huge neighborhoods [1], and, worse, moving an activity to another position would possibly force us to move also its predecessors and successors in the same well, because of constraints C10; or (ii) *A well*: with higher granularity, but since the sequence of activities in a well may be splited in the present schedule due to constraints C1, some problems now being that moving all activities takes time to verify all constraints, and exchanging the whole well may not be possible even though exchanging only part of it could be; or finally (iii) *Part of a well*: (that is, a maximal set of activities of the same well scheduled consecutively in the same resource) with medium granularity and already satisfying constraint C10.

In our implementation we chose the last possibility, where the local search algorithm exchanges all pairs of parts of two wells, no matter on what resource they have been scheduled. That neighborhood is of size $O(n^2)$, where $n$ is the number of parts of wells. For practical instances, this is one order of magnitude smaller than the neighborhood that uses activities as the moving elements.

To fully specify the local search phase we need a rule that defines how the neighborhood is searched and which solution replaces the current one. This rule is called the *pivoting rule* [13], and examples of it are the *first improvement rule* (FIR) and the *best improvement rule* (BIR). In the first case, the algorithm moves to a neighboring solution as soon as it finds a better solution; in the second case, all neighbors are checked and the best one is chosen. In either case, the worst case running time of each iteration is bounded by $O(n^2)$, where $n$ is the number of elements in the neighborhood. In the next section we present a comparison between these two alternatives.

## 4 Computational Results

In this section, computational results for the GRASPW implementation are given. They are also compared with results obtained with the ORCA implementation over the same real instances. All tests were run on a platform equipped with a Sun SPARC Ultra 60 processor running a Solaris 9 operating system at 450 MHz and with 1024 MB of RAM. Both GRASPW and ORCA were allowed to run for 1800 seconds on each instance.

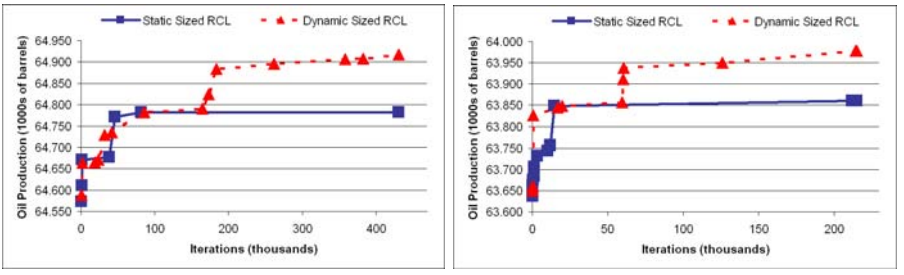
**Typical Instances.** Twenty one real instances provided by Petrobras were used in our tests. Table 4 summarizes the dataset. Columns with the same numerical data refer to distinct instances that differ in the number of other constraints, like C9. The first part of that table displays the instances were no C7 constraints were found. In order to reduce the amount of time spent in testing, in some experiments we used only 7 of these instances, eliminating instances that differed only in the number of clusters (see constraint C9, in Section 2). The lower part shows the nine instances were C7 constraints were present. The ORCA implementation had difficulties to handle these contraints.

**Table 1.** Test instances

| Instance | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # wells | 29 | 22 | 29 | 29 | 17 | 22 | 22 | 29 | 29 | 22 | 29 | 22 |
| # activities | 98 | 107 | 98 | 98 | 111 | 107 | 128 | 98 | 98 | 107 | 98 | 107 |
| # boats | 1 | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 2 |
| # derriks | 3 | 2 | 3 | 3 | 2 | 2 | 3 | 3 | 3 | 2 | 3 | 2 |
| # C7 constr. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Instance | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|
| # wells | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 29 |
| # activities | 107 | 107 | 107 | 107 | 107 | 107 | 107 | 107 | 98 |
| # boats | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 1 |
| # derriks | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| # C7 constr. | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |

**Setting GRASPW Parameters.** In Section 3 we presented the idea of a *dynamic sized RCL*. There are at least two ways we could exploit this idea: we may decrease the number of candidates through time, using a greedier heuristic; or we may increase the number of candidates through time, in order to drive away from a local optimum into new regions of the search tree. In the first case, the initial RCL size is set to $\max(13, w)$, $w$ being the number of wells, and is decreased by one every 300 seconds without improvement. In the second case, we start with $\max(5, w)$ for the initial RCL size and increase it by one every 300 seconds without improvement. The first approach did not yield



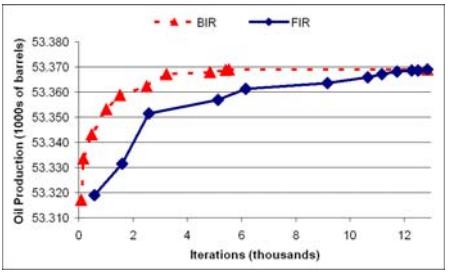

(a) Instance 1 (b) Instance 2

**Fig. 3.** Static Sized RCL x Dynamic Sized RCL

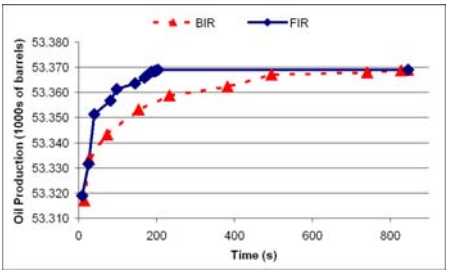
good results when applied to the WDP, generating the same or worse solutions than those found by GRASPW with a static sized RCL. However, the second approach proved promising. Figure 3.a shows the algorithm with dynamic sized RCL generating better solutions after 150 thousand iterations, when the RCL is increased. The same happens in Figure 3.b after 50 thousand iterations, when another real instance is tested. Amongst twelve scenarios tested, four had better solutions with the dynamic sized RCL, totalizing an increase of around 261 thousand barrels of oil. In the other eight scenarios, the same solutions were found.

Another technique tested was to bias the selection towards some particular candidates, those with the highest oil yield, but this strategy did not produce good results. Amongst twelve instances tested, three had slightly worse solutions with such a bias function, totaling a decreasing of 40 thousand barrels of oil. Worse, with the bias function in place, the algorithm takes much more time to find the same solution than when no bias is used. Summing up all the differences in time for the twelve scenarios, with the bias function the algorithm took 3431 more seconds to reach the same results, an average increase of 72%.

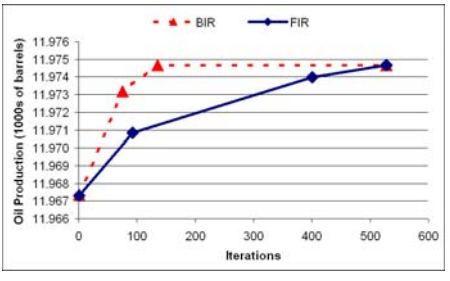
We also considered two options for searching the neighborhood and selecting a new neighbor: the *first improvement rule* (FIR) and the *best improvement rule* (BIR). We tested both of them on seven instances. The BIR heuristics proved to be the best one when finding solutions whose production was equal to a predefined target value and with the least number of iterations (see Figure 4 (a) and (c)). On average, to find a solution with a predefined production, the BIR strategy used about 60% of the number of iterations of the FIR strategy. On the other hand, the FIR strategy was faster in most instances (see Figure 4 (b) and (d)). On average, to find a solution with a predefined production, the FIR strategy used 66% of the time used by the BIR strategy. That is because, on average, a FIR iteration was almost seven times faster than a BIR iteration. Since, to users, running time was deemed important, the FIR strategy was found to better suit this problem.



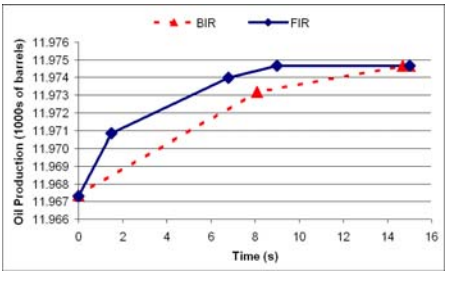

(a) Instance 1 - BIR




(b) Instance 1 - FIR




(c) Instance 2 - BIR

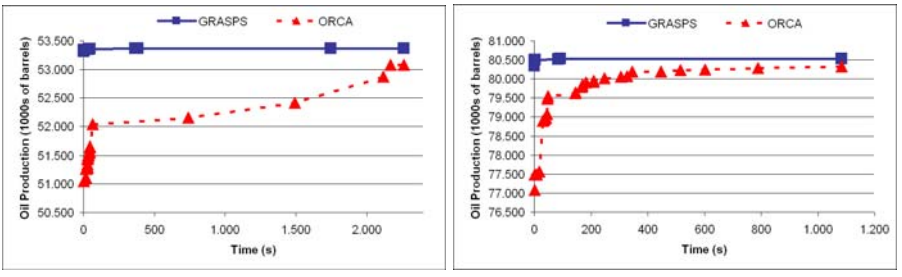



(d) Instance 2 - FIR

**Fig. 4.** BIR × FIR

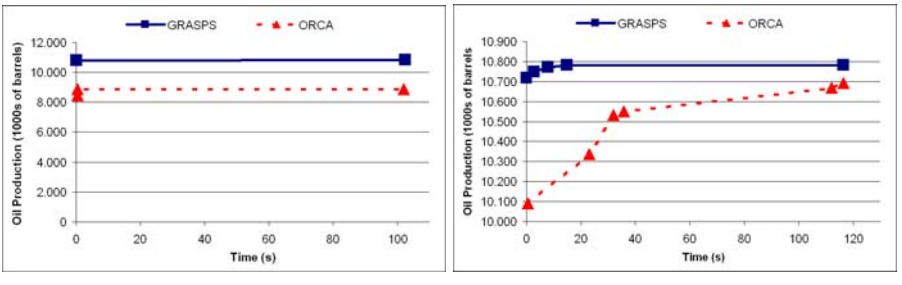
**The ORCA and the GRASPW Implementations.** Our GRASPW implementation found better solutions than the ORCA implementation on all twelve but one of the real instances tested and, in that one, it found the same solution. GRASPW achieved 0.14% more oil production on average which, despite being a small percentage, means an increase of almost one million barrels of oil, in total. Perhaps, the highest gain with GRASPW was in running time. It found solutions with the same production of those generated by ORCA, on average, in only 36.3% of the time used by ORCA on the same instances. Figure 5.a shows that the best GRASPW solution has a production 290 thousand barrels of oil higher than the best ORCA solution. Furthermore, GRASPW found a solution with the same oil production of the best ORCA solution within the first second, while ORCA found it only after 2200 seconds. Similarly, Figure 5.b shows that the best GRASPW solution has a production of almost 200 thousand barrels of oil higher than the best ORCA solution. Again, GRASPW found a solution with the same oil yield as the best ORCA solution within the first second, while ORCA found it only after 1000 seconds.

In all the nine tested scenarios, where the constraints C7 were present, GRASPW found better solutions than ORCA. In one of them, ORCA could not find any solution, while GRASPW found one with about 26 million barrels

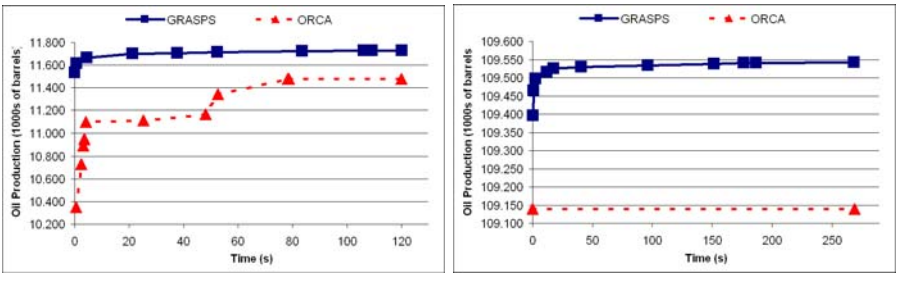



(a) Instance 1 (b) Instance 2

**Fig. 5.** ORCA x GRASPW




(a) Instance 3 (b) Instance 4




(c) Instance 5 (d) Instance 6

**Fig. 6.** ORCA x GRAPS

of oil production. Comparing the other eight scenarios, GRASPW achieved 5.3% more oil production, on average, which means an increase of around 4.6 million barrels of oil, in total. In Figure 6 we present the results over four such instances. In all of them GRASPW was more effective than ORCA.

## 5 Conclusions

Scheduling activities efficiently is of paramount importance to the industry, in general. Petrobras, a leading company in deep water exploration of oil, presented us the WDP, a scheduling problem related to oil well drilling. Here we constrast two approaches to the WDP: the constraint programming tool ORCA and a GRASP implementation, dubbed GRASPW. Computational experiments were carried out on several real instances. We conclude that GRASPW greatly outperforms ORCA. Not only it generates solutions with higher oil production, but often it outputs solutions with the same oil production as ORCA in much less time. We also note that ORCA already produced better results than the manual solutions.

It is worth mentioning that ORCA is built over the ILOG Constraint Programming suite, a set of highly expensive and sophisticated libraries with years of development. Using GRASPW, which was entirely programmed from the ground up, these costs could be averted. In opposition, the ILOG suite favors easiness of development, of maintenance and of understanding of the source code.

*Acknowledgements.* The third author was supported by grants 307773/2004-3 from CNPq and 03/09925-5 from FAPESP.

## References

1. do Nascimento, J.M.: Hybrid computational tools for the optimization of the production of petroleum in deep waters. Master's thesis, Institute of Computing, University of Campinas (2002)
2. Marriott, K., Stuckey, P.J.: Programming with Constraints: An introduction. MIT Press, Cambridge, Massachusetts (1998)
3. ILOG: ILOG Solver 4.4 Reference Manual. ILOG (1999)
4. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Norwell, Massachusetts (1997)
5. Feo, T.A., Resende, M.G.C.: Greedy randomized adaptative search procedures. Journal of Global Optimization **6** (1995) 109–133
6. Festa, P., Resende, M.G.C.: GRASP: An annotated bibliography. Technical report, AT&T Labs (2001)
7. Aiex, R.M., Binato, S., Resende, M.G.C.: Parallel GRASP with path-relinking for job shop scheduling. Parallel Computing **29** (2003) 393–430
8. Bard, J.F., Feo, T.A.: Operations sequencing in discrete parts manufacturing. Management Science **35** (1989) 249–255
9. Feo, T.A., Bard, J.F.: Flight scheduling and maintenance base planning. Management Science **35** (1989) 1415–1432
10. Feo, T.A., Bard, J., Holland, S.: Facility-wide planning and scheduling of printed wiring board assembly. Operations Research **43** (1995) 219–230

11. Binato, S., Hery, W.J., Loewenstern, D., Resende, M.G.C.: A GRASP for job shop scheduling. In Hansen, P., Ribeiro, C.C., eds.: Essays and surveys on metaheuristics, Kluwer Academic Publishers (2001)
12. Roy, B., Sussmann, B.: Les problèmes d'ordonnancement avec contraintes disjonctives. In: Note DS No 9 bis, Paris, SEMA (1964)
13. Yannakakis, M.: Computational complexity. In Aarts, E.H.L., Lenstra, J.K., eds.: Local Search in Combinatorial Optimization, Chichester, John Wiley & Sons (1997) 19–55

# A Framework for Probabilistic Numerical Evaluation of Sensor Networks: A Case Study of a Localization Protocol

Pierre Leone[1,2], Paul Albuquerque[2], Christian Mazza[3], and Jose Rolim[1]

[1] Computer Science Department, University of Geneva, 1211 Geneva 4, Switzerland
[2] LII, Ecole d'Ingénieurs de Genève, HES-SO, 1202 Geneva, Switzerland
[3] Mathematics Department University of Geneva 1211 Geneva 4, Switzerland

**Abstract.** In this paper we show how to use stochastic estimation methods to investigate the topological properties of sensor networks as well as the behaviour of dynamical processes on these networks. The framework is particularly important to study problems for which no theoretical results are known, or can not be directly applied in practice, for instance, when only asymptotic results are available. We also interpret Russo's formula in the context of sensor networks and thus obtain practical information on their reliability. As a case study, we analyse a localization protocol for wireless sensor networks and validate our approach by numerical experiments. Finally, we mention three applications of our approach: estimating the number of pivotal sensors in a real network, minimizing the number of such sensors for robustness purposes during the network design and estimating the distance between successive localized positions for mobile sensor networks.

## 1 Introduction

Recent technological improvements have favoured the development of small, inexpensive, low-power, multifunctional sensor nodes. These tiny sensor nodes are the numerous sensing, data processing, and communicating components of an extended network: a wireless sensor network [1]. There are several types of sensors (*e.g.* thermal, visual, infrared, acoustic or radar), which are able to monitor their vicinity (*e.g.* measure temperatures, noise levels or vehicular movements). Wireless sensor networks can thus help monitor and control physical environments through collection and distribution of data in which the usually densely scattered sensors collaborate to perform some specific action. The collected data must be routed back through a multihop infrastructureless architecture to a distinguished node which can then communicate to a task manager via the Internet or by satellite. In most cases, the environment to be monitored does

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 341–353, 2005.
© Springer-Verlag Berlin Heidelberg 2005

not have an existing infrastructure neither for energy nor communication and is often very rough. This influences the design of a sensor network with respect to fault tolerance, scalability, production costs, sensor network topology, hardware constraints, transmission media, computational capacity, and power consumption. The constraints are often due to the inexpensive nature and ad-hoc method of deployment of the sensors. As a matter-of-fact, the major concern is to extend the lifetime and robustness of sensor networks. Particular care must given to energy efficient routing, localization algorithms and system design. To this end, energy consumption for processing and communication must be optimized. It is also required that sensor networks display some form of self-organization for both communication and positioning so as to adapt to changing connectivity (*e.g.* addition or failure of nodes) as well as changing environmental conditions.

Application areas include military, environment, health and chemical processing. In particular, wireless sensor networks can perform bio-system analysis or surveillance of seismic activity through area monitoring. They can for example help to identify the type, concentration, and location of pollutants in air or water. In such settings, sensors monitoring a given part of the entire area should know their location in order to provide information on *what* happens together with *where* it happens. Towards this goal, sensors could be equipped with GPS systems. However, this solution would have two main drawbacks: the cost and the energy consumption [17, 18]. Indeed, for large scale sensor networks the individual price of GPS systems becomes an important factor in the network design. Moreover, the energy requirement would shorten the lifespan of the network. Therefore, many strategies have been developed to ensure localization of the entire set of sensors while minimizing the number of GPS systems involved in the localization process. Surveys on the existent strategies can be found in [5, 23] where they are classified and analysed. We also refer to [8, 9] in which the authors make an empirical analysis of different protocols, in particular for localization. With respect to complexity, the localization problem turns out to be NP-hard [2]. It can be worthwhile to use stochastic estimation methods to obtain numerical information about the process. In relation to this, let us mention the inspiring work [20, 21, 22] that makes strong use of randomization and stochastic processes.

In this paper, we show how to use stochastic estimation methods, first introduced in [15], to investigate the topological properties of sensor networks as well as the behaviour of dynamical processes on these networks. The framework is particularly important to study problems for which no theoretical results are known, or can not be directly applied in practice, for instance, when only asymptotic results are available. The framework is also useful, for example, for network design or optimisation of network parameters. Moreover, we interpret the well-known Russo formula [7, 16] in the context of sensor networks and we thus obtain practical information on the network reliability. We point out in the existing literature [6, 19] suitable numerical estimation methods and implement them for

our practical analysis. Concretely, our framework is applied to the problem of localization in wireless sensor networks.

The paper is organised as follows. In the next section, we introduce a model of sensor networks and a localization protocol. Section 3 presents the probabilistic model, theoretical results concerning the probability of success of the localization proocol and the stochastic estimation methods to be implemented. Numerical experiments are then performed and discussed in section 4. We go on in section 5 with three applications of the framework developed previously. We draw some conclusion in the final section.

## 2 Model of Sensor Networks and Localization Protocol

We consider sensors scattered over a unit square region $[0,1]^2$ with a uniform distribution. A given sensor can communicate with all the sensors located in a region corresponding to a sector of a disk (see Figure 1). The motivation for considering such a geometry for the transmission of the signal comes from optical networks [9] as well as from the development of smart antennas for radio transmission. Indeed, if radio waves are chosen as the mean of transmission of the information, the problem of interferences between simultaneous emissions has to be faced. A way of addressing this is to choose a shape for the antenna emission pattern which minimizes the probability of interferences, provided a solution exists! Results in the field of percolation [4] show that the shape of the antenna emission pattern has an influence on the percolation probability and, quite surprisingly, that the circle seems to be the worst shape. In this paper interferences are not taken into account and we assume that transmission is supported by protocols able to recover from lost data. We do not either deal directly with energy consumption. The radius of emission $r$ as well as the angle of emission $\alpha$ are kept constant through time and are the same for all sensors. The orientation $\beta$ is also kept constant. It is drawn from a uniform distribution on the circle ($[0, 2\pi[$). Notice that the assumption that the angle of emission is constant with time is not essential. Actually our analysis remains valid if the

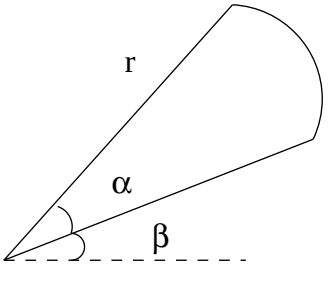



**Fig. 1.** Sensor with radius of emission $r$, angle of emission $\alpha$ and orientation of the emission $\beta$

angles change independently. This is also the case if the sensors move in such a way that they remain uniformly distributed on the square region.

The localization protocol whose properties we investigate is the following. The total number of sensors is fixed equal to $n$ out of which a subset, henceforth referred to as *localized* sensors, is equipped with GPS systems. For each sensor, this occurs independently with probability $p$. Localized sensors make their positions known to connected sensors. A sensor which receives 3 positions is able to locate itself, for example by triangulation using distance estimates. Then, a sensor which has just computed its position sends it out so that the localization process goes on until no new sensor becomes localized. We say that the process is *successful* if the percentage of sensors that are eventually localized, is greater than a fixed threshold.

## 3 Probabilistic Model and Recursive Estimation

We begin by describing the probabilistic model for the point process which accounts for the uniform scattering of the sensors over the square area $[0,1]^2$, and the random selection of an orientation for information transmission. Recall that $n$ denotes the total number of scattered sensors. A point configuration $V = \{(x_1, y_1, \theta_1), \ldots, (x_n, y_n, \theta_n)\}$ of $n$ sensors is an unordered sequence of triplets $(x_i, y_i, \theta_i)$ where $(x_i, y_i)$ are coordinates and $\theta_i$ an orientation in which sensor $i$ transmits. The coordinates, respectively the orientations, are random variables independently and uniformly distributed in $[0,1]$, respectively in the circle $[0, 2\pi[$. Let $S_i$ be the disk sector described in Figure 1 with origin $(x_i, y_i)$. We define the following incidence relation on $V$: there is an oriented edge from sensor $X_i$ located at $(x_i, y_i)$ to sensor $X_j$ located at $(x_j, y_j)$ if $(x_j, y_j) \in S_i$. Clearly, an edge indicates that sensor $X_i$ can transmit data to sensor $X_j$.

**Definition 1.** *We denote by $\Lambda$ the entire set of sector graphs $(V, A)$ where $A$ is the set of edges associated to the point configuration $V$.*

The space $\Lambda$ can be made into a probability space. For our purpose, we do not need to exhibit how this probability space is defined and hence, the construction is omitted and we refer to [12] for similar constructions.

Let us formulate the localization process probabilistically. We first obtain a sector graph in $\Lambda$ resulting from the scattering of sensors in $[0,1]^2$. A random orientation for the emission is assigned to each sensor which in turn receives a GPS system with probability $p$. So $p$ is roughly the density of localized sensors among the total set of sensors at the beginning of the localization process. We denote by $\eta = \eta(p)$ a possible assignment of GPS, where $\eta(i) = 1$ if sensor $i$ is localized and 0 otherwise. The set of all GPS assignments is denoted by $\Omega$. It is a probability space with the power set $\mathcal{P}(\Omega)$ as $\sigma-$algebra and the product probability measure. The process of localization starts according to the protocol described at the end of the previous section. We consider the event whether the process is successful or not. Thus, we define a Bernoulli random

variable $L : \Lambda \times \Omega \to \{0,1\}$ which depends on the parameter $p$, i.e. $L = 1$ with probability $m(p) := P_p(\{L = 1\})$. The primary task in the analysis of the chances of success for the localization process is to obtain a plot of the function $m(p)$. Theoretical results about this function can then be derived to support the effectiveness of our numerical methods.

**Theorem 1.** *The function $m(p)$ is polynomial in $p$.*

*Proof.* To a given sector graph we associate its adjacency matrix. Denoting by *Adj* the set of adjacency matrices there is then a measurable map $\Lambda \to Adj$ which defines a probability measure on the set *Adj*. The random variable $L : \Lambda \times \Omega \to \{0,1\}$ can actually be seen as a random variable $L : Adj \times \Omega \to \{0,1\}$, we thus keep the same notation for both random variables. Since *Adj* and $\Omega$ are finite sets and since each element belonging to $Adj \times \Omega$ has polynomial probability in $p$, all events belonging to $\mathcal{P}(Adj \times \Omega)$ are polynomial in $p$.

□

**Definition 2.** *We define a partial order on $\Omega$ by setting $\eta \leq \eta'$ if $\eta(i) \leq \eta'(i)$*

**Definition 3.** *An event $B \subset \Omega$ is said to be increasing if its indicator function $I_B$ is increasing, namely $I_B(\eta) \leq I_B(\eta')$ whenever $\eta \leq \eta'$.*

**Proposition 1.** *Given a sector graph $g \in \Lambda$ the event $\{L = 1 \,|\, g\}$ is increasing.*

*Proof.* The event $\{L = 1 \,|\, g\}$ depends only on the GPS assignment vector $\eta$ and it is clear that adding a localized sensor can do nothing but increase the indicator function of $\{L = 1 \,|\, g\}$. Formal arguments based on coupling can be found in [10].

□

**Definition 4.** *Given a sector graph $g$, a sensor $i$ is called* pivotal *if $\eta \in \{L = 1 \,|\, g\}$ and $\eta' \notin \{L = 1 \,|\, g\}$ where $\eta'$ is obtained from the assignment $\eta$ by changing $\eta(i) = 1$ to $\eta(i) = 0$.*

We must emphasize the following fact: given a sector graph, a pivotal sensor can neither fail nor be removed without compromising the success of the localization algorithm. Hence, the number of pivotal sensors gives a crucial information about the reliability of the localization process. The derivative of the function $m(p)$ has a representation in terms of pivotal elements.

**Theorem 2.** *The derivative of the function $m(p) := P_p(\{L = 1\})$ satisfies*

$$m'(p) := \frac{d}{dp} P_p(\{L = 1\}) = E_p(\textit{number of pivotal sensor for } \{L = 1\}). \quad (1)$$

*Moreover, the function $P_p(\{L = 1\})$ is increasing with $p$.*

*Proof.* By proposition 1 the event $\{L = 1 \,|\, g\}$ is increasing, hence we can apply Russo's formula [7, 10, 12, 16]. In our context, the formula states

$$\frac{d}{dp} P_p(\{L = 1 \,|\, g\}) = E_p(\text{number of pivotal sensor for } \{L = 1 \,|\, g\}). \qquad (2)$$

Taking the expectation of (2), we get at once the right hand side of (1). The probability $P_p(\{L = 1 \,|\, g\})$ being polynomial in $p$ the expectation and the derivative commute to obtain the left hand side of (1). The right hand side of (1) is positive implying that the function $P_p(\{L = 1\})$ is increasing.

□

**Definition 5.** *We call the function* $\frac{d}{dp} P_p(\{L = 1\})$ *the instability coefficient function.*

Our numerical analysis is based on stochastic estimation methods, first introduced in [15]. This work stimulated many researches because of the vast domain of applications ranging from non-parametric and parametric estimation to stochastic optimisation, including automation and remote control. Research in this field is synthesized in [14]. In [11] the authors already applied a similar procedure to implement a solution to the energy balanced data propagation in wireless sensor networks. The main drawback of the procedure is the relatively slow rate of convergence which is of order $\mathcal{O}(\#\text{step}^{-1/2})$.

We assume that the total number of sensors $n$, the radius of emission $r$ and the angle of emission $\alpha$ are fixed. We choose an initial value for the parameter $p_1$ which is the probability that a sensor is equipped with a GPS system. We draw a sample sector graph and a configuration of sensors which are localized accordingly to the value of the parameter $p_1$. Then, we start the localization process which yields a sample for the random variable $L : \Lambda \to \{0, 1\}$. The parameter $p_1$ is updated with the rule [15]

$$p_{j+1} = p_j + \frac{1}{j}(y - L(p_j))$$

with $y \in [0, 1]$ kept fixed. If the parameterized probability function $m(p) = P(\{L = 1 \,|\, p\})$ satisfies some regularity conditions [3, 14, 15], then it is known that $p_j \to p(y)$ as $j \to \infty$ where $p(y)$ is such that $m(p(y)) = y$. Hence, by repeating this procedure with different values of $y$, we get a set of points $(p(y), y)$ of the curve $p \mapsto m(p)$.

**Theorem 3.** *The sequence* $(p_j)_{j \geq 1}$ *recursively defined by*

$$p_{j+1} = p_j + \frac{1}{j}(y - L(p_j))$$

*converges almost surely to a value* $p$ *such that* $m(p) = y$*, for any initial value* $p_1$*.*

*Proof.* The results in theorem 2 imply that the hypothesis (see [3, 14, 15]) ensuring the convergence of the scheme are fulfilled.

□

The derivative of $m(p)$ must also be estimated since it is related to the expected number of pivotal sensors (theorem 2). This problem was first solved in [19] (see also [6, 14]). The numerical scheme requires to simulate the localization process twice per random sector graph with different values for the paremeter $p$ as stated in the following theorem.

**Theorem 4.** *The intertwined numerical scheme ([19]) recursively defined by*

$$a_j = \frac{1}{j} \sum_{k=1}^{j} \frac{L(p_j + j^{-\gamma}) - L(p_j - j^{-\gamma})}{2j^{-\gamma}}.$$

*and*

$$p_{j+1} = p_j + \big(\frac{1}{j} + \frac{1}{\sqrt{j}}\big) \frac{1}{2a_j} \big(y - L(p_j + j^{-\gamma}) + y - L(p_j - j^{-\gamma})\big),$$

*where* $0 < \gamma \leq 0.5$*, converges almost surely for any initial value* $p_1$*. The sequence* $p_j$ *converges to a value* $p$ *which satisfies* $m(p) = y$ *and* $a_j \to m'(p)$ *with* $m'(p)$ *the derivative of* $m$ *at* $p$*.*

The simulations in the next section apply the previous scheme with $\gamma = 0.4$.

## 4 Numerical Simulations

We propose a numerical scaling analysis of the localization process by plotting the probability of success $m(p)$ and its derivative $m'(p)$ for various antenna radiation patterns. We performed three sets of experiments. In the first one, we keep the angle of emission and the density of sensors constant while the radius of emission varies. In the second one, the angle changes while the radius and the density remain constant. In the last one, we vary the density keeping the other two parameters constant. These simulations are devoted to the numerical analysis of the impact of the antenna radiation pattern on the success of the localization process. This problem is connected to open questions in the field of percolation theory mentioned in the introduction and discussed in [4]. It is in particular important for the development of smart antennas for wireless networks. This set of experiments also allows us to validate our framework based on stochastic estimation methods.

We now present our simulations. In Figures 2, 3 and 4, we plot the graphs $p \mapsto m(p)$ on the left and $p \mapsto m'(p)$ on the right. For each set of parameters, angle of emission, radius of emission and density, the computations are performed three times because of the slow rate of convergence. This gives us some confidence on the convergence of the process when the measurement is taken.

The first set of experiments is shown in Figure 2. It allows us to numerically investigate the impact of increasing the area of the antenna radiation pattern through an increase of the radius of emission while the angle of emission is kept

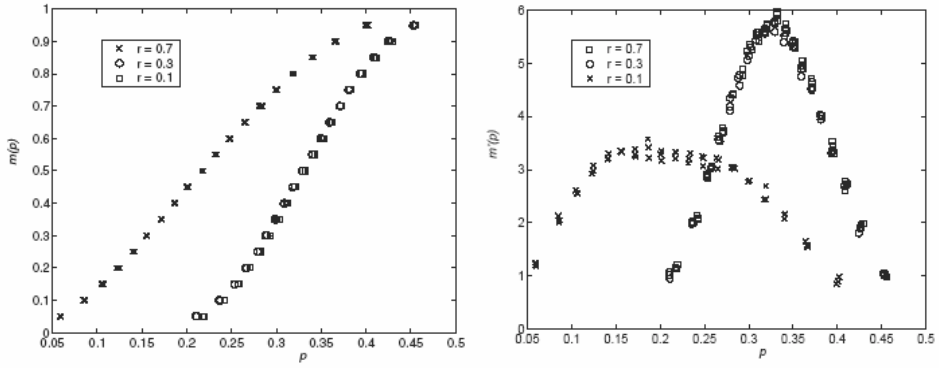



**Fig. 2.** Graphs of the regression function $m(p)$ on the left and $m'(p)$ on the right with $\alpha = 1.$ and $r = 0.1, 0.3, 0.7$

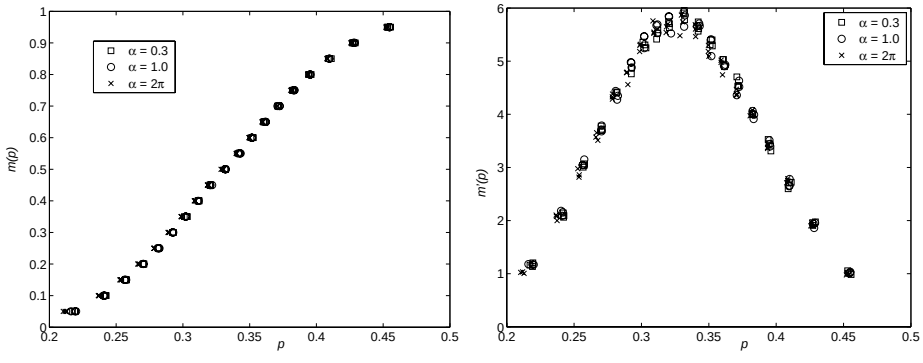



**Fig. 3.** Graphs of the regression function $m(p)$ on the left and $m'(p)$ on the right with $\alpha = 0.3, 1.0, 2\pi$ and $r = 0.1$

constant. We chose $n = 50$ for the total number of sensors and $\alpha = 1.0$ radian for the angle of emission. The radius of emission varies from $r = 0.1$ to $r = 0.7$.

For the second set of experiments, we still have $n = 50$ for the total number of sensors and the radius of emission is constant with $r = 0.1$. The angle of emission changes from $\alpha = 0.3$ to $\alpha = 2\pi$ radian. Results are presented in Figure 3.

The last set of experiments deals with the impact of the density of sensors on the success of the localization process. We successively chose $n = 50, 100, 200$ while the radius of emission with $r = 0.1$ and the angle of emission with $\alpha = 2\pi$ are kept constant. Numerical results are presented in Figure 4.

Although the numerical analysis of this type of problem should be done in a more systematic way than in this work, we still try to give some interpretation of the results. Figure 2 shows that increasing the radius of emission improves the probability of success of the localization protocol. This is expected since the area covered by the emission and hence the expected number of sensors prone to receive the data, are increased. However, when the radius is small, the probability of success seems to depend little on the radius. This can be observed from the

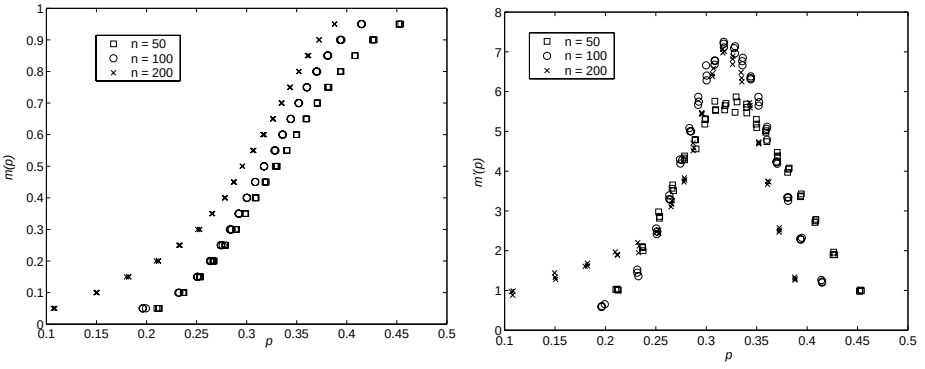



**Fig. 4.** Graphs of the regression function $m(p)$ on the left and $m'(p)$ on the right with $\alpha = 2\pi$ and $r = 0.1$ and $n = 50, 100, 200$ from top to bottom

superposition on Figure 2 of the curves with radius of emission $r = 0.1, 0.3$. With a larger radius of emission $r = 0.7$, the probability of success increases significantly and the process becomes more robust since the number of pivotal sensors decreases. Further numerical investigations with a finer tuning of the radius will allow us to monitor more carefully what happens when the radius of emission increases. For instance, whether there seems to be a phase transition or not. Suprisingly results on Figure 3 shows no dependency on the angle of emission $\alpha$. This might be due to the relatively small value of the radius. One can infer that the probability of success of the localization algorithm does not only depend on the area covered by the emission but also on the shape of the emission radiation pattern. Results on Figure 4 show that increasing the density of the sensors increases the chances of success of the process. More interestingly, we observe that the maximum expected number of pivotal sensors seems to depend little on the density. Indeed, on Figure 4 right, we observe a maximum at about 7 pivotal sensors. In our opinon, it is worth investigating this particular point further, since it could reflect some structural properties of the random graphs.

## 5 Applications

We now describe three practical applications of the framework introduced in this paper. The first application is concerned with the problem of estimating the instability coefficient, hence the expected number of pivotal sensors in a given networks. The second application is oriented towards network design. We assume that the total number of sensors to be scattered in a given region as well as their characteristics are known. We look for a placement of the sensors in the region which minimises the expected number of pivotal sensors in order to make the network more robust to failures. The last application deals with mobile sensor networks in an informal manner.

### 5.1 Networks Robustness Estimation

We consider a given network with $n$ sensors, $k$ of them being localized with a GPS system. We assume that the process of localization described in section 2 is successful. The problem is to estimate the number of pivotal sensors. If we assume that the probability for a sensor to be localized is $p$, the expected total number of localized sensors $X$ is $E(X) = np$. We search for the probability that $X = k$. Let $\epsilon < 1$ be the order of certainty in the following analysis. Using Chernoff's bound [13], we know that with

$$\delta(np, \epsilon) = \sqrt{\frac{4\ln(1/\epsilon)}{np}}, \quad 0 < \delta < 1, \tag{3}$$

we have

$$Pr\big((1-\delta)np < X < (1+\delta)np\big) < e^{-np\delta^2/2}. \tag{4}$$

Clearly, the right hand side of (4) has to be made as small as possible. We rewrite the expression on the left side replacing $\delta$ with the expression (3) to obtain

$$\left(1 - \sqrt{\frac{4\ln(1/\epsilon)}{np}}\right)np < k < \left(1 + \sqrt{\frac{4\ln(1/\epsilon)}{np}}\right). \tag{5}$$

Given that we have fixed the order of uncertainty $\epsilon$, we obtain

$$\frac{2\ln(1/\epsilon) + k - 2\sqrt{\ln(1/\epsilon)^2 + k\ln(1/\epsilon)}}{n} < p... \tag{6}$$

$$... < \frac{2\ln(1/\epsilon) + k + 2\sqrt{\ln(1/\epsilon)^2 + k\ln(1/\epsilon)}}{n} \tag{7}$$

For example, assuming $n = 50$ and $k = 5$, we get with $\epsilon = 0.1$

$$0.28 < p < 0.36. \tag{8}$$

If we assume that the radius of emission $r = 0.1$ and the angle of emission $\alpha = 1$, we can look at Figure 2 to determine that the expected number of pivotal sensors lies in the interval $[4.0, 5.5]$.

### 5.2 Network Design

Assume we are given a fixed number of sensors $n$ with specified characteristics and we would like to monitor a given area. The problem is to place the sensors in such a way that a given process will succeed in the network and ensure that the placement maximises the robustness of the network. Although, the process we consider is evidently not the localization process, it could be a flooding process for example, we keep in mind the situation of the previous section as

an illustration. We assume that we have already computed a figure similar to the one in Figure 2 for the particular process under investigation. Consider for example the right picture in Figure 2. We would like to exhibit a configuration of sensors which minimises the expected number of pivotal sensors, i.e. maximises the robustness of the network. This can be done using the same type of probabilistic method: choose a value of the parameter $p$ which minimises the expected number of pivotal elements and get sample points through simulations. For this application, it would be useful to use a stochastic estimation for the variance of the expected number of pivotal sensors in order to bound the real number of pivotal elements.

### 5.3 Mobile Sensor Networks

Consider a sensor network with mobile sensors monitoring a given area. We assume for cost and energy consumption reasons that not every sensor is equipped with a GPS system. Thus, whenever sensors want to send data, a localization process must take place because the data is meaningless without the location of the sensor. Therefore, a probabilistic estimate of the maximal or typical distance a sensor travels between localized positions is required. Our framework can apply to the situation where every sensor needs to be localized at a given time. We assume the motion of the sensors is represented by a suitable Markov chain $X(t)$ in the set of random sector graphs. For every discrete time $t$, the localization process on the random sector graph $X(t)$ can be successful or not. We denote this random variable by $L(X(t)) \in \{0, 1\}$. The process $X(t)$ possesses limit laws which entails that $\lim_{t\to\infty} P(\{L(X(t)) = 1\}) = \nu$. The limit $\nu$ can be inferred from numerical simulations as described in section 3. For $t$ large enough, we have $P(\{L(X(t)) = 1\}) \approx \nu$ and the probability that the interval between two successes of the localization process is $n$ can therefore be approximated by $(1 - \nu)^{n-1}\nu$ (geometric distribution).

## 6 Conclusion

The probabilistic framework developed in this paper is validated by the simulations and can therefore be applied for handling difficult problems numerically. Concerning the analysis of the localization protocol, we observed that the changes in the shape of the probability of success are not important when the radius or the angle of emission vary, provided that the density of sensors is relatively small. However, the shape of the plots vary considerably more with the increase of the density as was observed in the numerical experiments. A complete analysis of the protocol would require more systematic numerical computations. We emphasize that the aim of this paper was to build the framework for stochastic numerical evaluation, validate our approach and point out possible applications. The applications listed in the previous section, demonstrate the potential of the method.

## References


1. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, Vol. 38, Issue 4, pp. 393–422 (2002).
2. J. Aspnes, D. Goldenberg and Y.R. Yang. On the Computational Complexity of Sensor Network Localization. *In Proc. of the 1st Int'l Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, Turku, Finland, LNCS 3121, Springer Verlag (2004).
3. J.R. Blum. Approximation Methods which Converge with Probability One. *Ann. Math. Stat.*, Vol. 25, No. 2, pp. 382–386 (1954).
4. L. Booth, J. Bruck, M. Franceschetti and R. Meester. Covering Algorithms, Continuum Percolation and the Geometry of Wireless Networks. *Ann. Appl. Prob.*, Vol. 13, No. 2, pp. 722–741 (2003).
5. N. Bulusu, J. Heidmann and D. Estrin. GPS-less low-cost outdoor localization for very small devices. *IEEE Personal Communications, Special Issue on Smart Spaces and Environments*, Vol. 7, No. 5, pp. 28–34 (October 2000).
6. D.L. Burkholder. On a class of stochastic approximation processes. *Ann. Math. Stat.*, Vol. 27, pp. 1044–1059 (1956).
7. J.T. Chayes and L. Chayes. The mean-field bound for the order of parameter of Bernoulli percolation. *Percolation Theory and Ergodic Theory of Infinite Particle Systems*, Minneapolis, Minn., 1984–1985, IMA, Vol. Math. Appl. 8, Springer, New York, pp. 49–71 (1987).
8. J. Diaz, J. Petit and M. Serna. A random graph model for optical networks of sensors. *IEEE Transactions on Mobile Computing*, Vol. 2, No. 3, pp. 186–196 (2003).
9. J. Diaz, J. Petit and M. Serna. Evaluation of Basic Protocols for Optical Smart Dust Networks. *In Proc. of the 2nd Int'l Workshop on Experimental and Efficient Algorithms (WEA 2003)*, Ascona, Switzerland, LNCS 2647, Springer Verlag (May 2003).
10. G. Grimmett. *Percolation*. A Series of Comprehensive Studies in Mathematics, Vol. 321, 2nd ed., Springer Verlag (1999).
11. P. Leone, S. Nikoletseas and J. Rolim. An adaptative Blind Algorithm for Energy Balanced Data Propagation in Wireless Sensor Networks, submitted to DCOSS 2005.
12. R. Meester and R. Roy. *Continuum percolation*. Cambridge University Press (1996).
13. R. Motawni and P. Raghavan. *Randomized Algorithms*. Cambridge University Press (1995).
14. M.B. Nevel'son and R.Z. Has'minskii. Stochastic Approximation and Recursive Estimation. *Translation of Mathematical Monographs*, Vol. 47, American Math. Soc. (1976).
15. H. Robbins and S. Monro. A stochastic approximation method. *Ann. Math. Stat.*, Vol. 22, pp. 400–407 (1951).
16. L. Russo. A note on percolation. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, Vol. 56, pp. 229–237 (1981).
17. M. Srivastava. Part II: Sensor Node Platforms & Energy Issues. *Tutorial on Wireless Sensor Networks, Mobicom'02* (2002). Avalaible at http://nesl.ee.ucla.edu/tutorials/mobicom02.
18. M. Srivastava. Part III: Time & Space Problems in Sensor Networks. *Tutorial on Wireless Sensor Networks, Mobicom'02* (2002). Avalaible at http://nesl.ee.ucla.edu/tutorials/mobicom02.
19. J.H. Venter. An extension of the Robbins-Monro procedure. *Ann. Math. Stat.*, Vol. 38, pp. 181–190 (1967).

20. I. Chatzigiannakis, P. Spirakis and S. Nikoletseas. Efficient and Robust Protocols for Local Detection and Propagation in Smart Dust Networks. *ACM Mobile Networks and Applications*, Vol. 10, Issue 1-2, pp. 133–149 (February 2005).
21. I. Chatzigiannakis, T. Dimitriou, M. Mavronicolas, S. Nikoletseas and P. Spirakis. A Comparative Study of Protocols for Efficient Data Propagation in Smart Dust Networks. *Parallel Processing Letters*, Vol. 13, No. 4, pp. 615–627 (December 2003).
22. I. Chatzigiannakis, T. Dimitriou, S. Nikoletseas and P. Spirakis. A Probabilistic Algorithm for Efficient and Robust Data Propagation in Smart Dust Networks. *In Proc. of the 5th European Wireless Conf. on Mobile and Wireless Systems beyond 3G*, pp. 344–350 (2004).
23. A.Rao, S.Ratnasamy, C.Papadimitriou, S.Shenker and I.Stoica.Geographic Routing without Location Information. *MOBICOM'03*, pp. 96–108 (2003).

# A Cut-Based Heuristic to Produce Almost Feasible Periodic Railway Timetables*

Christian Liebchen

TU Berlin, Institut für Mathematik, Sekr. MA 6-1,
Straße des 17. Juni 136, D-10623 Berlin, Germany
liebchen@math.tu-berlin.de

**Abstract.** We consider the problem of satisfying the maximum number of constraints of an instance of the Periodic Event Scheduling Problem (PESP). This is a key issue in periodic railway timetable construction, and has many other applications, e.g. for traffic light scheduling.

We generalize two (in-) approximability results, which are known for MAXIMUM-K-COLORABLE-SUBGRAPH. Moreover, we present a deterministic combinatorial polynomial time algorithm. Its output violates only very few constraints for five real-world instances.

## 1 Introduction

The Periodic Event Scheduling Problem (PESP) has been introduced by Serafini and Ukovich ([18]). This powerful model has many practical applications, e.g. traffic light scheduling ([7]) and periodic railway timetabling. In the context of the second application, it has been exemplified in several studies that exact optimization can be very difficult for real-world instances ([16, 10, 12]). These studies made even use of sophisticated MIP models and several problem specific classes of valid inequalities.

Nevertheless, the PESP has proven to be sufficiently powerful to model the vast majority of the requirements which practitioners impose. For a concise review of the modeling capabilities of the PESP— including the minimization of the amount of rolling stock required to operate a timetable, and even some decisions of line planning—we refer to [9]. In particular, the 2005 timetable of the Berlin underground is the first one which has been computed by mathematical optimization methods ([8]). At the level of strategical planning, both Nederlandse Spoorwegen and Deutsche Bahn AG use software which is based on the PESP-model ([17, 10]).

The reported difficulties motivate to have a look at local search procedures. This does also hold for models for timetabling that are based on the quadratic semi-assignment problem (QSAP), for which Daduna and Voß consider the minimization of passenger transfer times ([4]). For some PESP-models, the genetic algorithm that has been proposed by Nachtigall and Voget ([13]), constitutes a competitive alternative ([10, 5]).

* Supported by the DFG Research Center "Mathematics for key technologies" (MATHEON) in Berlin.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 354–366, 2005.
© Springer-Verlag Berlin Heidelberg 2005

However, sometimes it is already difficult to come up with a feasible solution, and the performance may depend on the quality of the initial population.

*Contribution*. This is the first time, the question of satisfying as many PESP-constraints as possible is addressed formally. We are able to generalize two (in-) approximability results that were established for MAXIMUM-K-COLORABLE-SUBGRAPH.

Moreover, we present a deterministic combinatorial polynomial time algorithm, whose output violates only very few constraints. In a computational study on five practical instances—ranging from long-distance traffic over regional traffic down to undergrounds—we exhibit its superiority compared to three other heuristics, two of these being previously published in related contexts.

## 2 Problem Definition

We start by defining the PESP formally, and then derive the problem of satisfying the maximum number of constraints of a PESP instance. We include the constant integer period time $T$ of the input network in the name of the problem—just as with the constant $K$ for K-VERTEX-COLORABILITY.

An instance $I = (D, \ell, u)$ of T-PESP consists of a directed graph $D = (V, A)$ and vectors $\ell$ and $u$ of lower and upper time bounds for the arcs. As usual, we set $n := |V|$ and $m := |A|$, and we denote by $G(D)$ the underlying undirected graph of $D$. We may assume $\ell_a \leq u_a$. In the case of $\ell$ and $u$ being integral vectors, we call an instance of T-PESP *integral*. A (feasible) solution of a T-PESP instance is a vector $\pi : V \to [0, T)$—which may represent time values of, say, hourly recurring departure/arrival events within a public transportation network, see [9]—fulfilling periodic constraints of the form

$$(\pi_j - \pi_i - \ell_a) \bmod T \leq u_a - \ell_a,\ a = (i, j) \in A, \tag{1}$$

or $\pi_j - \pi_i \in [\ell_a, u_a]_T$ for short. Sometimes, we will refer to a constraint (1) by its arc $a$. In practice, often a (linear) objective function over the *slack times* $(\pi_j - \pi_i - \ell_a) \bmod T$ has to be minimized. We mention two simple but useful properties of T-PESP.

**Lemma 1 ([18]).** *If the underlying undirected graph $G(D)$ of $D$ of an instance $I$ of* T-PESP *is a forest, then $I$ has a feasible solution.*

*Remark 1 ([11]).* For every feasible integral instance of T-PESP, there exists an integral solution $\pi \in \{0, \ldots, T-1\}^V$, because a resulting LP is totally unimodular.

*Remark 2.* Notice that we allow parallel arcs explicitly. They provide the ability to model disjunctive constraints ([18]), which are extremely useful in practice ([9]).

The input for MAX-T-PESP is the same as for T-PESP. A solution of MAX-T-PESP is a vector $\pi : V \to [0, T)$ which maximizes

$$|\{a = (i, j) \subseteq A \,|\, \pi_j - \pi_i \in [\ell_a, u_a]_T\}|.$$

Besides providing initial solutions for local search algorithms for PESP minimization problems, MAX-T-PESP has an intrinsic application in practice. For instance, during the evenings' service—where $T$ equals $10$ minutes—the Berlin underground aims at offering a changeover waiting time of at most five minutes for a maximum number of connections ([8]). Currently, this can be offered for the $48$ most important connections. Among the next $86$, of approximately $110$ remaining connections, $55$ do not exceed this bound. In [17], a similar goal is reported for Dutch railways (NS).

## 3 Approximability of MAX-T-PESP

Odijk ([14]) proposed the most convenient proof of the NP-completeness of T-PESP by polynomially transforming K-VERTEX-COLORABILITY to T-PESP. Recall that there are two canonical optimization variants of K-VERTEX-COLORABILITY. The most popular is to compute the chromatic number of a graph. But this question is of no practical relevance for the construction of periodic railway timetables, because the period time of the transportation system is a fixed constant—passengers would never accept a period time of, say, $53$ minutes.

Consider the other optimization variant, namely MAXIMUM-K-COLORABLE-SUBGRAPH. Here, we seek for a $K$-coloring of the vertices such that a minimum number of edges relates two vertices sharing the same color. We summarize some of the main properties of MAXIMUM-K-COLORABLE-SUBGRAPH and relate them to MAX-T-PESP.

**Theorem 1 ([15]).** MAXIMUM-K-COLORABLE-SUBGRAPH *is MAXSNP-hard.*

**Theorem 2.** MAX-T-PESP *is MAXSNP-hard.*

*Proof.* We provide an L-reduction from MAXIMUM-K-COLORABLE-SUBGRAPH to MAX-T-PESP. Consider an instance of MAXIMUM-K-COLORABLE-SUBGRAPH being defined on a graph $G(V, E)$. Let $D = (V, A)$ be an arbitrary orientation of $G$. Set $T = K$ and define $\ell_a = 1$, $u_a = T - 1$ for every $a \in A$.

Let $A'$ be any subset of $A$ and consider the instance $I' := (D' = (V, A'), \ell|_{A'}, u|_{A'})$ of T-PESP. Due to Remark 1, we know that from every solution of $I'$ we can derive an integral solution $\pi' \in \{0, \dots, T-1\}^V$ of $I'$ in polynomial time. By the choice of $\ell$ and $u$, we may interprete $\pi'$ as a vertex coloring.

Let $E' \subseteq E$ be the projection of $A'$ into $G$. Then, there exists a bijection between $K$ colorings of $G$ which respect the edges in $E'$ and vectors $\pi \in \{0, \dots, T-1\}$ which respect the constraints (1) for the corresponding set $A'$. Trivially, the above construction constitutes an $L$-reduction—in particular $\alpha = \beta = 1$. □

**Corollary 1.** *There is no PTAS for* MAX-T-PESP*, unless P=NP.*

**Corollary 2.** MAX-T-PESP *can be approximated within some fixed constant ratio.*

**Proposition 1 ([19]).** MAXIMUM-K-COLORABLE-SUBGRAPH *can be approximated with ratio* $\frac{K-1}{K}$.

*Remark 3.* Recall that for dense graphs, a PTAS for MAXIMUM-K-COLORABLE-SUBGRAPH can be derived using the techniques of Arora, Karger, and Karpinski ([1, 3]). Further, notice that the ratio of $\frac{K-1}{K}$ can be improved by applying the semidefinite programming techniques due to Goemans and Williamson ([6, 3]). However, this improvement becomes arbitrarily small for large values for $K$.

Proposition 1 might sound promising for railway timetabling. There, often $T$ is at least 60 minutes. Hence, the precision used by the German railway infrastructure company (6s) yields even $T = 600$. Further, bear in mind that the K-COLORING requirement can be respected for every arc being incident to vertices with degree at most $K-1$.

We should analyze, if we may profit from Proposition 1 for MAX-T-PESP. Unfortunately, this is limited, in particular if many constraints with small *span ratio* $\rho_a = \frac{\Delta_a+1}{T}$ are involved, $\Delta_a := u_a - \ell_a$ being the *span* of a constraint, i.e. with $\rho_a \ll 1$. Notice that the span ratio is invariant under any scaling of the time precision. We say that a constraint (1) is *symmetric*, if $u_a = T - \ell_a$. Further, we call an instance of T-PESP *span homogeneous* if there exists an $\alpha \in [0, T)$ such that $\Delta_a = \alpha$ for every $a \in A$.

On the one hand, neither symmetric nor span homogeneous instances seem to have any practical motivation in railway timetabling. On the other hand, as they constitute the bridge to MAXIMUM-K-COLORABLE-SUBGRAPH, they will enable us to generalize both, algorithms and approximation guarantees from MAXIMUM-K-COLORABLE-SUBGRAPH to MAX-T-PESP. More specifically, in the following section we provide a $\rho$-approximation algorithm for the span homogeneous integral MAX-T-PESP.

## 4 Heuristics for MAX-T-PESP

We present four heuristics for MAX-T-PESP. The first and the second one were previously published ([13, 18]). The third one is inspired by Vitanyi's approximation algorithm for MAXIMUM-K-COLORABLE-SUBGRAPH ([19]). The fourth one constitutes a considerable improvement of the first heuristic and is the main target of the computational study in Section 5.

### 4.1 MST Heuristic

Algorithm 1 can already be found in the pioneering work of Serafini and Ukovich ([18]). It is based on Lemma 1 and thus ensures $n - 1$ constraints of $I$ to be satisfied. By choosing the spans as weights for the MST computation, we ensure that $n - 1$ very tight constraints—i.e. hopefully the most difficult ones—are always satisfied.

A more detailed analysis requires to specify how to choose the value for $\pi_v$ in Step 6. By the following lemma, we illustrate that a global rule which does not take into account local configurations of a specific instance provides only poor results.

**Lemma 2.** *For every $T \geq 4$, the approximation ratio of Algorithm 1 for* MAX-T-PESP *is $\Theta(\frac{n}{m})$, if in Step 6 $\pi_v$ is selected such that the lower bound of arc $a$ becomes tight.*

*Proof.* Consider the complete graph $K_n$. Orient its edges such that $a = (i, j) \in A$ implies $i < j$. Set $\ell_a = 0$ and $u_a = 1$ if $a = (1, v)$, $v \in \{2, \ldots, n\}$, $\ell_a = 1$ and

**Algorithm 1** MST heuristic ([18]) for MAX-T-PESP

**Input**: Instance $I = (D, \ell, u)$ of MAX-T-PESPwith $D$ being a connected digraph
**Output**: Vector $\pi \in [0, T)^V$ and spanning tree $F \subseteq A$ of $D$, such that the vector $\pi$ is feasible for all arcs $a \in F$

1: Compute a minimum spanning tree $F \subseteq A$ of $D$ with respect to the arcs' span $\Delta_a = u_a - \ell_a$.
2: Choose an arbitrary $v \in V$.
3: Set $\pi_v := 0$ and $V' := \{\pi_v\}$.
4: **while** $V' \neq V$ **do**
5: Choose $v \in \Gamma(V')$, i.e. a neighbor of $V'$ and let $a \in F$ be the arc connecting $v$ to $V'$.
6: Set $\pi_v$ to a value in $[0, T)$ such that constraint $a$ is satisfied.
7: $V' \leftarrow V' \cup \{v\}$
8: **end while**

$u_a = T - 1$ otherwise. Algorithm 1 assigns the the same value $\pi_v$ to every vertex. Hence, precisely the $n - 1$ constraints induced by the tree arcs are satisfied.

Now, set $\pi_v = 0$ for $v$ odd, and $\pi_v = 1$ otherwise. The constraints of the tree arcs are still satisfied. But any arc that connects an odd vertex with an even vertex becomes satisfied, too. Roughly speaking, this is every second arc of $K_{n-1}$. Hence, $\Theta(m)$ constraints can be satisfied. □

**Corollary 3.** *For planar graphs, Algorithm 1 ensures an approximation ratio of* $\frac{1}{3}$.

### 4.2 Local Improvements

A vector $\pi$ computed by Algorithm 1 can be improved locally by the following algorithm, which is due to Nachtigall and Voget ([13]). Observe that the performance of Algorithm 2 depends on the order in which the vertices are processed.

**Algorithm 2** Local improvement algorithm for MAX-T-PESP

**Input**: Instance $I = (D, \ell, u)$ of MAX-T-PESP and a vector $\pi \in [0, T)^V$
**Output**: Vector $\pi' \in [0, T)^V$

1: $\pi' \leftarrow \pi$
2: **for all** $v \in V$ **do**
3: Compute the minimal value $t \in [0, T)$ such that with $\pi'_v + t$ a maximum number of arcs in the cut $\delta(\{v\})$ are satisfied.
4: $\pi'_v \leftarrow \pi'_v + t$
5: **end for**

Given a set of vertices $X$ and a vector $\pi$, we introduce sets $P(X, \pi)$ such that for $X = \{v\}$ the set $P(\{v\}, \pi')$ contains candidate values for $t$ in Step 3 of Algorithm 2,

$$P(X,\pi) := \bigcup_{a=(i,j)\in\delta^+(X)} \{((\pi_j - \pi_i) - \ell_a) \bmod T, ((\pi_j - \pi_i) - u_a) \bmod T\} \cup \bigcup_{a=(i,j)\in\delta^-(X)} \{(\ell_a - (\pi_j - \pi_i)) \bmod T, (u_a - (\pi_j - \pi_i)) \bmod T\} \cup \{0\}. \quad (2)$$

**Lemma 3.** *Let $t$ be the choice of Algorithm 2, when processing vertex $v$. Then, $t \in P(\{v\}, \pi')$.*

*Proof.* We know that $\pi'_v + t$ is in the intersection of periodic intervals whose bounds we collect in $\{\pi'_v + t' \mid t' \in P(\{v\}, \pi')\}$. □

**Corollary 4.** *Algorithm 2 can always be implemented with runtime $O(m^2 n)$. For integral instances and integral input vectors $\pi$, we obtain $O(n \cdot \min\{m, T\} \cdot m)$.*

*Proof.* We know $|P(\{v\}, \pi')| \leq 2m + 1$. Remark 1 guarantees $|P(\{v\}, \pi')| \leq T$ for integral instances. □

Notice that we may face $n \in o(\delta(\{v\}))$ because of Remark 2.

*Remark 4.* Further improvements can be achieved by executing Algorithm 2 repeatedly.

### 4.3 An Approximation Algorithm for Span Homogeneous MAX-T-PESP

Algorithm 3 works much similar to Algorithm 2. Surely, they are of a different flavor, as Algorithm 3 does not require any input vector $\pi$. But observe that the only difference

**Algorithm 3** Approximation algorithm for span homogeneous integral MAX-T-PESP

**Input**: Instance $I = (D, \ell, u)$ of MAX-T-PESP
**Output**: Vector $\pi \in [0, T)^V$

1: $V' \leftarrow \emptyset$
2: **for all** $v \in V$ **do**
3: $A' \leftarrow \{a \in A \mid \exists u \in V' : a = (u, v) \text{ or } a = (v, u)\}$
4: Set $\pi_v$ to the minimum value in $[0, T)$, such that a minimum number of constraints $a \in A'$ are violated.
5: $V' \leftarrow V' \cup \{v\}$
6: **end for**

between Algorithm 3 and applying Algorithm 2 to $\pi = \mathbf{0}$ is that here only such arcs are taken into account that connect the current vertex with vertices that were already processed. This will enable us to compute a lower bound for the approximation ratio of MAX-T-PESP restricted to span homogeneous integral instances, cf. Corollary 2.

**Lemma 4.** *For every arc $a \in A$ there exists precisely one vertex $v \in V$ such that $a \in A'$ in Step 3 of Algorithm 3.*

*Proof.* The arc $a = (u, v) \in A$ is in $A'$, if and only if the second of its two vertices is processed by the for-loop. □

**Theorem 3.** *For span homogeneous integral instances $I = (D, \ell, u)$ of* MAX-T-PESP *with span $\Delta$, the vector $\pi$ produced by Algorithm 3 satisfies at least $\frac{\Delta+1}{T}|A|$ constraints.*

*Proof.* By Lemma 4, we may decompose the analysis into the $|V|$ iterations. Moreover, by Remark 1 and an analogon of Lemma 3, we consider only integer vectors $\pi$.

Denote by $f_v(t)$ the number of arcs that are feasible if we set $\pi_v$ to $t$. By our assumption on the span homogeneity of $I$, for every $a \in A'$ there are precisely $\Delta + 1$ values for $\pi_v$, such that $a$ is respected. This yields

$$\sum_{t=0}^{T-1} f_v(t) = |A'| \cdot (\Delta + 1).$$

Trivially, there exists some $t \in \{0, \dots, T-1\}$ such that

$$f_v(t) \geq \left\lceil \frac{\Delta + 1}{T} |A'| \right\rceil.$$

□

**Corollary 5.** *Algorithm 3 is a $\rho$-approximation algorithm for the span homogeneous integral* MAX-T-PESP *with span ratio $\rho$.*

**Proposition 2.** *The runtime of Algorithm 3 is bounded by the runtime of Algorithm 2.*

*Remark 5.* By considering symmetric span homogeneous instances of MAX-T-PESP with $\ell = \mathbb{1}$ it gets obvious that Theorem 3 is a generalization of Vitanyi's result ([19]) for MAXIMUM-K-COLORABLE-SUBGRAPH. However, both the algorithm and its analysis became more direct than in the original paper. Finally, we may conclude that an approximation ratio of $\rho$ cannot be tight—at least for symmetric instances, cf. Remark 3.

### 4.4 Cut Improvements of the MST Heuristic

The last heuristic we propose is motivated by the poor quality of Algorithm 1, and by Lemma 1. For a spanning tree $F \subseteq A$ and an arc $a \in F$ consider the set $C_a$ of arcs of the fundamental cut induced by $a$ and $F$. Observe that for every $a' \in C_a \setminus \{a\}$ there is a unique cycle in $F \cup \{a'\}$. Algorithm 4 considers these cycles. We refer to applying cut improvements (Algorithm 4) to an output vector of Algorithm 1 as *cut heuristic*.

**Proposition 3.** *The runtime of the cut heuristic is $O(m^2 n)$. For integral instances and integral input vectors $\pi$, we achieve $O(n \cdot \min\{m, T\} \cdot m)$.*

*Proof.* We know $|\delta(X)| \leq m$ and $|F| = n-1$. Further, we will find $t$ in the set $P(X, \pi)$ and, hence, the analysis of Corollary 4 deploys. □

**Proposition 4.** *For every $T \geq 3$ there are instances of* MAX-T-PESP*, where Algorithm 4 examines $\Theta(nm)$ arcs in total.*

**Algorithm 4** Cut-based improvements for MAX-T-PESP

**Input**: Instance $I = (D, \ell, u)$ of MAX-T-PESP with $D$ being a connected digraph, a vector $\pi \in [0, T)^V$, and a spanning tree $F$
**Output**: Vector $\pi' \in [0, T)^V$
1: $\pi' \leftarrow \pi$
2: **for all** $a \in F$ **do**
3: Let $C_a \subseteq A$ be the arc set of the fundamental cut induced by the arc $a = (i, j)$ and the tree $F$, and define $X \subset V$ such that $\{i\} \in X$ and $\delta(X) = C_a$.
4: Compute the minimal value $t \in [0, T)$ such that $\pi' + t \cdot \chi_X$ satisfies a maximum number of constraints in $\delta(X)$. $\{\chi_X \in \{0, 1\}^V$ being the characteristic vector of $X\}$
5: $\pi' \leftarrow \pi' + t \cdot \chi_X$
6: **end for**

*Proof.* Again, consider the complete graph $K_n$ and orient its edges such that $a = (i, j) \in A$ implies $i < j$. Define the feasible intervals of the constraints as follows:

$$[\ell_a, u_a]_T = \begin{cases} [1, 2]_T, \text{ if } a = (i, i+1),\ i = 1, \ldots, n-1, \\ [0, 2]_T, \text{ otherwise.} \end{cases}$$

Thus, the spanning tree will be a path.

Let us assume $n = 3k + 1, k \in \mathbb{N}$, for notational convenience. Consider the $\frac{n-1}{3}$ fundamental cuts that are induced by the tree arcs $a = (i, i+1), i = k+1, \ldots, 2k$. Each of these contains all the $\Theta(m)$ arcs $a = (i, j)$ with $i \leq k+1$ and $j \geq 2k+1$. □

Theorem 2 and Proposition 3 imply that there are feasible instances of T-PESP, for which the cut heuristic fails to produce a feasible solution, otherwise $P = NP$.

*Example 1.* Consider the feasible T-PESP instance in Figure 1 ($T \geq 6$). After shifting any solution such that $\pi_1 = 0$, the unique feasible solution is $\pi^\star = (0, 0, 1, 2)^t$.

Let Algorithm 1 ensure the lower bounds of the constraints become tight for the tree arcs. This yields $\pi \equiv 0$ and the only non-tree arc is violated. Hence, the cut heuristic will only change $\pi$, if it can obtain $\pi = \pi^\star$—occasionally after shifting. But this is impossible, because $\pi^\star$ carries three distinct values and in every iteration of Algorithm 4, only one single offset can be applied to some set of vertices. □

Nevertheless, our computational study in Section 5 will reveal the notable benefit the cut heuristic is able to achieve. Unfortunately, our theoretical analysis does not

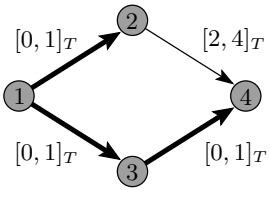



**Fig. 1.** Feasible instance of 6-PESP, but the cut heuristic fails to produce a feasible solution

reflect this quality. This is mainly caused by non-empty pairwise intersections of the fundamental cuts and by the multi-stage character of the cut heuristic.

*Remark 6.* Observe that the cut heuristic immediately extends to the so-called EXTENDED-PESP, in which the vertices may have different period times.

## 5 Computational Study

We will compare the cut heuristic for MAX-T-PESP to Algorithm 1. Further, we analyze how the local improvement heuristic performs when applied to the output vectors of these two algorithms. We will also apply the approximation algorithm for span homogeneous instances to the five practical data sets that we consider. We start by describing these. Notice that each of these five instances permits feasible timetables.

The first pair of data sets, ICE small and ICE big, share the same basic network. In particular, ICE small is a subset of ICE big, resulting from the deletion of certain traffic lines. In turn, the lines contained in ICE big are a subset of a strategic planning scenario of the long-distance service of Deutsche Bahn AG (DB AG). Since the underlying infrastructure has the same capacity, it shall be easier to construct a feasible timetable for ICE small than for ICE big. Most constraints are to ensure a minimal headway between two consecutive trains. But there are even some single tracks in these high-speed networks, e.g. "Hildesheimer Kurve". The data set ICE big has been the subject of an earlier extended computational study ([10]). Its results motivate a potential for local search techniques, even when compared to CPLEX© 8.1 ([2]), or to ILOG SOLVER.

The second pair of data sets model one of the five largest federal states of Germany. In PS Regio, only regional trains are considered. In PS all, we also include the long-distance traffic serving that state and fix the timetables on tracks outside the federal state to a planning scenario provided by DB AG. Notice that many regional trains use single tracks. Moreover, in the geographic intersection of PS all and ICE big, there is more traffic in PS all, because the full passenger traffic is included. Notice that the algorithms can deal with the three different periods that occur in these data sets, cf. Remark 6.

The last data set models the Berlin underground. Here, one tries to ensure a maximal waiting time of five minutes for the $48$ most important connections. But this must not happen at the price of stopping times which exceed $2.5$ minutes—travel times between stations being fixed. Also, some operational constraints have to be obeyed.

Table 1 provides additional information on the real-world instances, and on the resulting graph models. For the latter, we only mention classification numbers for the graph, in which redundancies have been eliminated by contraction steps ([11]). We also ignore arcs with $\Delta_a \geq T - 1$. Given the fact that the first four data sets make even use of parallel arcs, cf. Remark 2, none of the data sets induces a dense graph, cf. Remark 3.

*Computational Results*. We start by filling the theoretical benchmarks in the last row of Table 1 with life. Also in our context, when applied to instances that arise in practice, the approximation algorithm performs much better than can be guaranteed in general.

**Table 1.** Classification numbers of the real-world problems and graph models

| Quantity | ICE small | ICE big | PS Regio | | PS all | U Berlin |
|---|---|---|---|---|---|---|
| Real-world problems | | | | | | |
| Period times of the lines | 120’ | 120’ | 60’, 120’ | | 30’, 60’, 120’ | 10’ |
| Time precision | 60” | 60” | 6” | | 6” | 30” |
| Pairs of traffic lines | 11 | 31 | 66 | | 98 | 8 |
| # fixed pairs of lines | 0 | 0 | 0 | | 40 | 1 |
| # partly fixed pairs of lines | 0 | 0 | 0 | | 9 | 0 |
| Resulting graph models | | | | | | |
| Period time $T$ | 120 | 120 | 600 | 1200 | $\leq$1200 | 20 |
| Number of vertices $n$ | 69 | 173 | 160 | 192 | 343 | 38 |
| Number of arcs $m$ | 304 | 1102 | 341 | 387 | 1224 | 83 |
| Cyclomatic number $\mu$ | 236 | 930 | 377 | | 882 | 46 |
| Average span ratio $\overline{\rho}$ | 73.6% | 82.5% | 61.4% | 55.9% | — | 32.3% |
| Max. number of violated arcs for $\overline{\rho}$ span homog. instances | 80 | 192 | — | | — | 57 |

**Table 2.** Number of infeasibilities left by Algorithm 3

| Strategy | ICE small | ICE big | PS Regio | PS all | U Berlin |
|---|---|---|---|---|---|
| index | **6** | **22** | **31** | 81 | 12 |
| degree | 15 | 30 | 58 | 72 | 19 |
| intensity | 18 | 28 | 42 | **52** | **7** |
| Algorithm 3 plus local improvement | | | | | |
| index | **4** | 12 | **25** | 47 | 8 |
| degree | 8 | **11** | 36 | 41 | 14 |
| intensity | 9 | 14 | 34 | **38** | **6** |

Notice that we investigate three different strategies for Algorithm 3 to select the next vertex $v$: by index (increasingly), by the vertices’ degree (decreasingly), and by the intensity of the incident constraints, $\sum_{a\in\delta(\{v\})} T - (\Delta_a + 1)$, (decreasingly). Table 2 shows that none of these strategies dominates the two others.

Before presenting the results of our computations for the cut heuristic, we specify how we made use of the degrees of freedom left by Algorithms 1, 2, and 4. We run the MST heuristic for eleven *target spans* $p \in \{\frac{k}{10} \mid k \in \{0,\ldots,10\}\}$, and compute $\pi$ such that $(\pi_v - \pi_u - \ell_a) \bmod T = p \cdot \Delta_a$, for every arc $a = (u,v) \in F$. For the cut improvements, we choose cuts that have (many) infeasible arcs and many arcs with small span ratio foremost—we omit the full details due to space limitations. Finally, we apply Algorithm 2 repeatedly, until no more change appears, cf. Remark 4.

We either start with the result of the MST heuristic (“tree”) or of the cut heuristic (“cut”). These may be improved locally (“locImp”). In Figure 2 every chart represents the results obtained for one of the five data sets. For every data set, the four heuristics have been executed eleven times each, with different values for the target

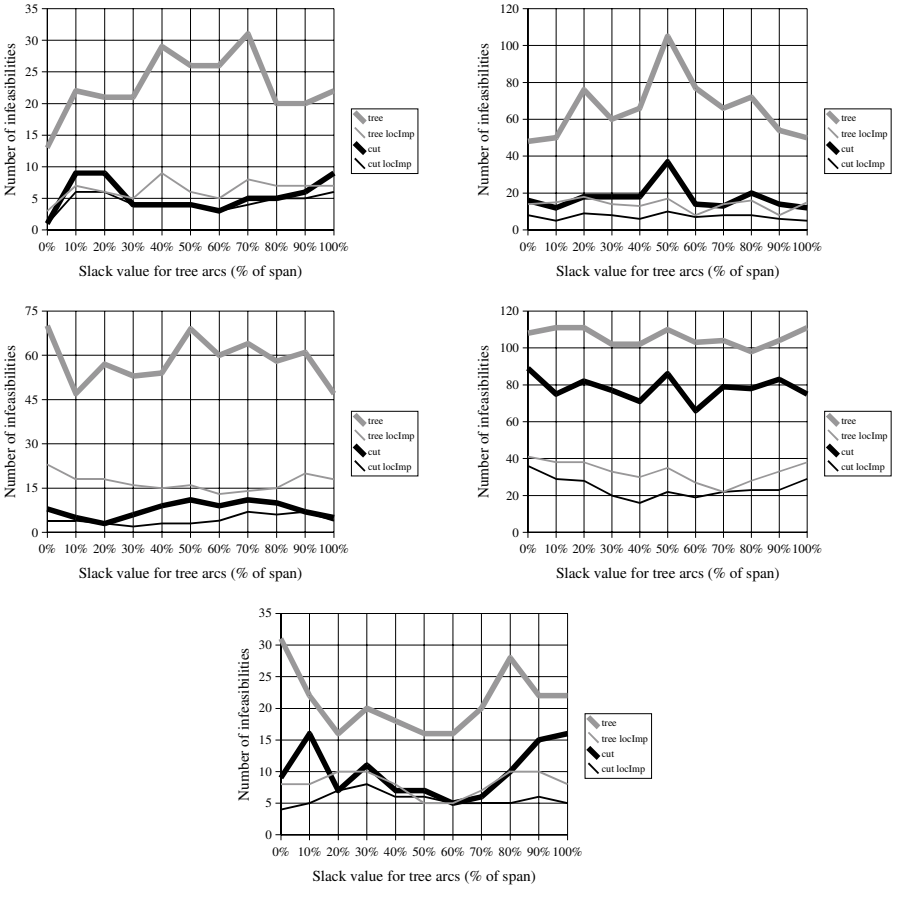



**Fig. 2.** Performance of MST heuristic and cut heuristic plus occasional local improvement for five data sets: ICE small, ICE big, PS Regio, PS all, and U Berlin (from left to right)

span to be applied by the initial MST heuristic. On the ordinate, the number of arcs that are violated by the output vector of a heuristic is given.

Let us summarize the main observations to be torn out of Figure 2 and relate them to the practical performance of our approximation algorithm:

- Both, Algorithm 3 and the cut heuristic are much superior to the pure MST heuristic. Only for PS all, the cut heuristic does not outperform Algorithm 3 significantly.
- For each of the five data sets, the best solutions are obtained by locally improving output vectors of the cut heuristic.
- For PS Regio, the worst solution obtained by the pure cut heuristic is still better than any locally improved output of Algorithms 1 and 3.

Recall that heuristics for MAX-T-PESP shall provide good initial input vectors for local search algorithms. Unfortunately, we have to admit that in some spot tests on

ICE big, we did not perceive any significant improvement in the performance of a genetic algorithm, when fed with locally improved output vectors of the cut heuristic.

## 6 Conclusions

We addressed the problem of satisfying as many constraints of a PESP-instance as possible. We proved it to be MAXSNP-hard and provided an approximation algorithm with constant approximation ratio $\rho$, $\rho$ being the span ratio of a span homogeneous integral instance of MAX-T-PESP. Moreover, we proposed a new heuristic that provides much better solutions for MAX-T-PESP than two heuristics that were previously published.

Unfortunately, the theoretical analysis of this cut heuristic stays limited. Hopefully, our promising computational results attract other researchers to join the theoretical analysis of the cut heuristic—or even to design further approximation algorithms for MAX-T-PESP. This is of importance, because PESP-techniques have just entered the practice of timetable design. And practice bears many instances, on which the existing algorithms leave space for improvements that, in turn, are really required by practice...

## Acknowledgments

I am thankful to Stefan Felsner and Marco Lübbecke for initial hints and discussions.

## References

1. Arora, S., Karger, D., and Karpinski, M. (1995) Polynomial Time Approximation Schemes for Dense Instances of NP-hard Problems. Proceedings of the 27th Annual ACM Symposium on Theory of Computing, 284–293, ACM Press, New York
2. CPLEX 8.1 (2004) `http://www.ilog.com/products/cplex`, ILOG SA, France.
3. Crescenzi, P. and Kann, V. (2005) A compendium of NP optimization problems. `http://www.nada.kth.se/~viggo/problemlist/compendium.html`
4. Daduna, J.R. and Voß, S. (1995) Practical Experiences in Schedule Synchronization. In Daduna, J.R. et al.: Computer-Aided Transit Scheduling— Proceedings of the Sixth International Workshop on Computer-Aided Scheduling of Public Transport, LNEMS 430, 39–55, Springer
5. Engelhardt-Funke, O. and Kolonko, M. (2004) Analysing stability and investments in railway networks using advanced evolutionary algorithms. International Transactions in Operational Research **11**, 381–394
6. Goemans, M. and Williamson, D. (1994) .878-Approximation Algorithms for MAX CUT and MAX 2SAT. Proceedings of the 26th Annual ACM Symposium on Theory of Computing, 422–431, ACM Press, New York
7. Hassin, R. (1996) A Flow Algorithm for Network Synchronization. Operations Research **44**, 570–579
8. Liebchen, C. (2005) Der Berliner U-Bahn Fahrplan 2005: Realisierung eines mathematisch optimierten Angebotskonzepts. Tagungsbericht Heureka '05 Optimierung in Transport und Verkehr, FGSV Verlag, in German

9. Liebchen, C., Möhring, R.H. (2004) The Modeling Power of the Periodic Event Scheduling Problem: Railway Timetables—and Beyond. Preprint 020/2004, Mathematical Institute, TU Berlin, Germany
10. Liebchen, C., Proksch, M., and Wagner, F.H. (2004) Performance of Algorithms for Periodic Timetable Optimization. Preprint 021/2004, Mathematical Institute, TU Berlin, Germany
11. Lindner, T. (2000) Train Schedule Optimization in Public Rail Transport. Ph.D. Thesis, TU Braunschweig, Germany
12. Martin, A., Achterberg, T., and Koch, T. (2003) MIPLIB 2003, `http://www.zib.de/miplib`
13. Nachtigall, K. and Voget, S. (1996) A genetic algorithm approach to periodic railway synchronization. Computers and Operations Research **23**, 453–463
14. Odijk, M. (1997) Railway Timetable Generation. Ph.D. Thesis, TU Delft, The Netherlands
15. Papadimitriou, C.H. and Yannakakis, M. (1991) Optimization, Approximation, and Complexity Classes. Journal of Computer and System Sciences **43**, 425–440
16. Peeters, L. (2003) Cyclic Railway Timetable Optimization. Ph.D. Thesis, Erasmus Universiteit Rotterdam, The Netherlands
17. Schrijver, A. and Steenbeek, A. (1993) Dienstregelingontwikkeling voor Nederlandse Spoorwegen N.V.—Rapport Fase 1. Report, CWI, The Netherlands, In Dutch
18. Serafini, P., Ukovich, W. (1989) A mathematical model for periodic scheduling problems. SIAM Journal on Discrete Mathematics **2**, 550–581
19. Vitanyi, P.M.B. (1981) How Well Can a Graph be $n$-Colored? Discrete Mathematics **34**, 69–80

# GRASP with Path-Relinking for the Weighted Maximum Satisfiability Problem*

Paola Festa[1] and Panos M. Pardalos[2], Leonidas S. Pitsoulis[3], and Mauricio G.C. Resende[4]

[1] Department of Mathematics and Applications, University of Napoli Federico II, Compl. MSA, Via Cintia, 80126, Napoli, Italy
paola.festa@unina.it
[2] Department of Industrial and Systems Engineering, University of Florida, 303 Weil Hall, Gainesville, FL 32611, USA
pardalos@ufl.edu
[3] Department of Mathematical and Physical Sciences, School of Engineering, Aristotle University of Thessaloniki, Thessaloniki, GR54124, Greece
pitsouli@gen.auth.gr
[4] Internet and Network Systems Research Center, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, NJ 07932, USA
mgcr@research.att.com

**Abstract.** A GRASP with path-relinking for finding good-quality solutions of the weighted maximum satisfiability problem (MAX-SAT) is described in this paper. GRASP, or Greedy Randomized Adaptive Search Procedure, is a randomized multi-start metaheuristic, where at each iteration locally optimal solutions are constructed, each independent of the others. Previous experimental results indicate its effectiveness for solving weighted MAX-SAT instances. Path-relinking is a procedure used to intensify the search around good-quality isolated solutions that have been produced by the GRASP heuristic. Experimental comparison of the pure GRASP (without path-relinking) and the GRASP with path-relinking illustrates the effectiveness of path-relinking in decreasing the average time needed to find a good-quality solution for the weighted maximum satisfiability problem.

## 1 Introduction

A propositional formula $\Phi$ on a set of $n$ Boolean variables $V = \{x_1, \ldots, x_n\}$ in conjunctive normal form (CNF), is a conjunction on a set of $m$ clauses

* AT&T Labs Research Technical Report TD-68QNGR. January 17, 2005. The research of Panos M. Pardalos is partially funded by NSF and CRDF grants.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 367–379, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$\mathbb{C} = \{C_1, \ldots, C_m\}$. Each clause $C_i$ is a disjunction of $|C_i|$ literals, where each literal $l_{ij}$ is either a variable $x_j$ or its negation $\neg x_j$. Formally, we write

$$\Phi = \bigwedge_{i=1}^{m} C_i = \bigwedge_{i=1}^{m} \left( \bigvee_{j=1}^{|C_i|} l_{ij} \right).$$

A clause is satisfied if at least one of its literals evaluates to 1 (true), which means that either one of the unnegated Boolean variables has the value of 1 or a negated variable has the value of 0. The propositional formula is said to be satisfied if all of its clauses are satisfied. In the Satisfiability problem (SAT), one must decide whether there exists an assignment of values to the variables such that a given propositional formula is satisfied. SAT was the first problem to be shown to be NP-complete [8]. The Maximum Satisfiability problem (MAX-SAT) is a generalization of SAT, where given a propositional formula, one is interested in finding an assignment of values to the variables which maximizes the number of satisfied clauses. Generalizing even further, if we introduce a positive weight $w_i$ for each clause $C_i$, then the weighted MAX-SAT problem consists of finding an assignment of values to the variables such that the sum of the weights of the satisfied clauses is maximized. The MAX-SAT has many applications both theoretical and practical, in areas such as complexity theory, combinatorial optimization, and artificial intelligence [5]. It is an intractable problem in the sense that no polynomial time algorithm exists for solving it unless P = NP, which is evident since it generalizes the satisfiability problem [11].

Due to the computational complexity of the MAX-SAT there has been an extensive research effort devoted to the development of approximation and heuristic algorithms for solving it. An $\epsilon$-approximate algorithm for the MAX-SAT is a polynomial time algorithm which finds a truth assignment to the variables that results in a total weight of the satisfied clauses that is at least $\epsilon$ times the optimum ($0 < \epsilon < 1$). We will refer to $\epsilon$ of an approximation algorithm as its *performance ratio*. The first approximation algorithms for the MAX-SAT were introduced in [18], where Johnson presented two algorithms with performance rations $(k-1)/k$ and $(2^k-1)/2^k$, where $k$ is the least number of literals in any clause. For the general case $k = 1$ they both translate to a 1/2-approximation algorithm, while it has been shown in [7] that the second algorithm is in fact a 2/3-approximation algorithm. A 3/4-approximation algorithm, based on network flow theory, was presented by Yannakakis in [32] and also in [14] by Goemans and Williamson. Currently the best deterministic polynomial time approximation algorithm for MAX-SAT achieves a performance ratio of 0.758 and is based on semidefinite programming [15], while there is also a randomized algorithm with performance ratio 0.77 [3]. Better approximation bounds for special cases of the problem in which, for instance, we restrict the number of literals per clause or impose the condition that the clauses are satisfiable have also been found [9, 20, 31]. With respect to inapproximability results, it is known [17] that unless P = NP there is no approximation algorithm with performance ratio

greater than 7/8 for the MAX-SAT in which every clause contains exactly three literals, thereby limiting the general case as well.

Local search is the main ingredient for most of the heuristic algorithms that have appeared in the literature for solving the MAX-SAT, where in conjunction with various techniques for escaping local optima they provide solutions which exceed the theoretical upper bound of approximating the problem. We can divide the heuristic algorithms that have appeared in the literature into two main classes. The first class being those heuristics which use the history of the search in order to construct a new solution, such as Tabu Search [16], HSAT [12] and Reactive Search [4], and those that are not history sensitive such as Simulated Annealing [30], GSAT [29] and GRASP [22, 24]. Surveys of approximation and heuristic algorithms for solving the MAX-SAT can be found in [5, 16].

GRASP is a constructive multi-start metaheuristic which has been applied to a wide range of well known combinatorial optimization problems with favorable experimental results [23]. In [24, 25], Resende, Pitsoulis, and Pardalos describe a GRASP implementation for solving the weighted MAX-SAT, and report extensive computational results on a set of weighted SAT benchmark instances [19] that indicate that the heuristic produces good quality solutions. Each iteration consists of two phases: a construction phase where a solution is constructed in a greedy randomized fashion; and a local search phase where the local optimum is found in the neighborhood of the constructed solution. GRASP can therefore be thought of as a *memoryless* procedure, where past information from previous solutions is not used for the construction of a new solution. In this paper, we show how memory can be incorporated in the GRASP for weighted MAX-SAT proposed in [24]. At each iteration of the GRASP heuristic, a path of feasible solutions linking the current solution with a solution from a set of elite (or good-quality) solutions previously produced by the algorithm is explored. Path-relinking has been used as a memory mechanism in GRASP [27] resulting in faster convergence of the algorithm.

The remainder of the paper is organized as follows. In Section 2, we briefly state the implementation of GRASP for the MAX-SAT from [24], while in Section 3 we describe how to apply path-relinking for the MAX-SAT. Finally, in Section 4, computational results are presented which demonstrate empirically that path-relinking results in faster convergence of GRASP.

## 2 GRASP for the Weighted MAX-SAT

The construction and local search phase of GRASP are described in detail in [24], while in [25] a complete Fortran implementation is given along with extensive computational runs. In this section, we provide a brief description in order to facilitate the discussion of path-relinking that will follow in the next section. Given a set of clauses $\mathbb{C}$ and a set of Boolean variables $V$, let us denote by $\mathbf{x} \in \{0,1\}^n$ the *truth assignment* which corresponds to the truth values assigned to the variables, while let $c(\mathbf{x})$ denote the sum of the weights of the satisfied clauses as implied by $\mathbf{x}$. Without loss of generality we can assume that all the

```
procedure GRASP(MaxIter,RandomSeed)
1     c_best := 0;
2     do k = 1,..., MaxIter →
3           x :=ConstructSolution(RandomSeed);
4           x :=LocalSearch(x);
5           if c(x) > c_best →
6                 x_best := x;
7                 c_best := c(x_best);
8           endif;
9     od;
7     return x_best
end GRASP;
```

**Fig. 1.** Pseudo-code of GRASP for maximization problem

weights $w_i$ of the clauses are positive integers. Given any two truth assignments $\mathbf{x}, \mathbf{y} \in \{0,1\}^n$ let us denote their *difference set*

$$\Delta(\mathbf{x}, \mathbf{y}) := \{i : x_i \neq y_i, i = 1, \ldots, n\} \tag{1}$$

and their *distance*

$$d(\mathbf{x}, \mathbf{y}) := |\Delta(\mathbf{x}, \mathbf{y})| = \sum_{i=1}^{n} |x_i - y_i|. \tag{2}$$

which is the Hamming distance, and will be used as a measure of proximity between two solutions. The GRASP procedure is shown in Figure 1. In the construction phase of the algorithm (line 3), let us denote by $\gamma_j^+$ and $\gamma_j^-$ the gain in the objective function value if we set the unassigned variable $x_j$ to 1 and 0, respectively, and by $X \subseteq V$ the set of already assigned variables. We compute the best gain

$$\gamma^* := \max\{\gamma_j^+, \gamma_j^- : j \text{ such that } x_j \in V \setminus X\}$$

and keep only those $\gamma_j^+$ and $\gamma_j^-$ that are greater or equal to $\alpha \cdot \gamma^*$ where $0 \leq \alpha \leq 1$ is a parameter. A random choice $\gamma_k^+(\gamma_k^-)$ among those best gains corresponds to a new assignment $x_k = 1$ ($x_k = 0$), which is added to our partial solution $X = X \cup \{x_k\}$. After each such addition to the partial solution, the gains $\gamma_j^+$ and $\gamma_j^-$ are updated, and the process is repeated until $|X| = n$. The parameter $\alpha$ reflects the ratio of randomness versus greediness in the construction process, where $\alpha = 1$ corresponds to a pure greedy selection for a new assignment and $\alpha = 0$ to a pure random assignment. Having completed a truth assignment $\mathbf{x}$, we apply local search (line 4) in order to guarantee local optimality. The 1-flip neighborhood is used in the local search, which is defined as

$$N_1(\mathbf{x}) := \{\mathbf{y} \in \{0,1\}^n : d(\mathbf{x}, \mathbf{y}) \leq 1\}, \tag{3}$$

where a depth-first search approach is employed in the sense that the current solution is replaced with a solution in the neighborhood which has greater cost. The search is terminated when the current solution is the local optimum.

## 3 Path-Relinking

Path-relinking was originally proposed by Glover [13] as an intensification strategy exploring trajectories connecting elite solutions obtained by tabu search or scatter search. Given any two elite solutions, their common elements are kept constant, and the space of solutions spanned by these elements is searched with the objective of finding a better solution. The size of the solution space grows exponentially with the the distance between the *initial* and *guiding* solutions and therefore only a small part of the space is explored by path-relinking. Path-relinking has been applied to GRASP as an enhancement procedure in various problems [1, 2, 6, 21, 26, 28], where it can be empirically concluded that it speeds up convergence of the algorithm. A recent survey of GRASP with path-relinking is given in [27].

We now describe the integration of path-relinking into the pure GRASP algorithm described in Section 2. Path-relinking will always be applied to a pair of solutions $\mathbf{x}, \mathbf{y}$, where one is the solution obtained from the current GRASP iteration, and the other is a solution from an elite set of solutions. We call $\mathbf{x}$ the *initial solution* while $\mathbf{y}$ is the *guiding solution*. The set of elite solutions will be denoted by $\mathcal{E}$ and its size will not exceed `MaxElite`. Let us denote the set of solutions spanned by the common elements of $\mathbf{x}$ and $\mathbf{y}$ as

$$S(\mathbf{x}, \mathbf{y}) := \{\mathbf{w} \in \{0,1\}^n : w_i = x_i = y_i, i \notin \Delta(\mathbf{x}, \mathbf{y})\} \setminus \{\mathbf{x}, \mathbf{y}\}, \tag{4}$$

where it is evident that $|S(\mathbf{x}, \mathbf{y})| = 2^{n-d(\mathbf{x},\mathbf{y})} - 2$. The underlying assumption of path-relinking is that there exist good-quality solutions in $S(\mathbf{x}, \mathbf{y})$, since this space consists of all solutions which contain the common elements of two good solutions $\mathbf{x}, \mathbf{y}$. Taking into consideration that the size of this space is exponentially large, we will employ a greedy search where a path of solutions

$$\mathbf{x} = \mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{d(\mathbf{x},\mathbf{y})}, \mathbf{w}_{d(\mathbf{x},\mathbf{y})+1} = \mathbf{y},$$

is built, such that $d(\mathbf{w}_i, \mathbf{w}_{i+1}) = 1, \quad i = 0, \ldots, d(\mathbf{x}, \mathbf{y})$, and the best solution from this path is chosen. Note that since both $\mathbf{x}, \mathbf{y}$ are local optima in some neighborhood $N_1$ by construction[1], in order for $S(\mathbf{x}, \mathbf{y})$ to contain solutions which are not contained in the neighborhoods of $\mathbf{x}$ or $\mathbf{y}$ we must have $d(\mathbf{x}, \mathbf{y}) > 3$. Therefore we need not apply path-relinking between any two solutions which are not sufficiently far apart, since it is certain that we will not find a new solution that is better than both $\mathbf{x}$ and $\mathbf{y}$.

The pseudo-code which illustrates the exact implementation for the path-relinking procedure is shown in Figure 2. We assume that our initial solution will

[1] Where the same metric $d(\mathbf{x}, \mathbf{y})$ is used.

```
procedure PathRelinking(y, E)
1     Randomly select a solution x ∈ {z ∈ E : d(y, z) > 4};
2     w_0 := x;
3     w* := x;
4     for k = 0, ..., d(x, y) − 2 →
5           max:= 0
6           for each i ∈ Δ(w_k, y) →
7                 w :=flip(w_k, i);
8                 if c(w) > max →
9                       i* := i;
10                      max:= c(w);
11                fi;
12          rof;
13          w_{k+1} :=flip(w_k, i*);
14          if c(w_{k+1}) > c(w*) → w* := w_{k+1};
15    endfor;
16    return (w*);
end PathRelinking;
```

**Fig. 2.** Pseudo-code of path-relinking for maximization problem

always be the elite set solution while the guiding solution is the GRASP iterate. This way we allow for greater freedom to search the neighborhood around the elite solution. In line 1, we select at random among the elite set elements, an initial solution $\mathbf{x}$ that differs sufficiently from our guiding solution $\mathbf{y}$. In line 2, we set the initial solution as $\mathbf{w}_0$, and in line 3 we save $\mathbf{x}$ as the best solution. The loop in lines 4 through 15 computes a path of solutions $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{d(\mathbf{x},\mathbf{y})-2}$, and the solution with the best objective function value is returned in line 16. This is achieved by advancing one solution at a time in a greedy manner, as illustrated in lines 6 through 12, while the operation $\texttt{flip}(\mathbf{w}_k, i)$ has the effect of negating the variable $w_i$ in solution $\mathbf{w}_k$. It is noted that the path of solutions never enters the neighborhood of $\mathbf{y}$.

The integration of the path-relinking procedure with the pure GRASP is shown in Figure 3, and specifically in lines 6 through 11. The pool of elite solutions is initially empty, and until it reaches its maximum size no path relinking takes place. After a solution $\mathbf{y}$ is found by GRASP, it is passed to the path-relinking procedure to generate another solution. Note here that we may get the same solution $\mathbf{y}$ after path-relinking. The procedure $\texttt{AddToElite}(\mathcal{E}, \mathbf{y})$ attempts to add to the elite set of solutions the currently found solution. A solution $\mathbf{y}$ is added to the elite set $\mathcal{E}$ if either one of the following conditions holds:

1. $c(\mathbf{y}) > \max\{c(\mathbf{w}) : \mathbf{w} \in \mathcal{E}\}$,
2. $c(\mathbf{y}) > \min\{c(\mathbf{w}) : \mathbf{w} \in \mathcal{E}\}$ and $d(\mathbf{y}, \mathbf{w}) > \beta n, \quad \forall \mathbf{w} \in \mathcal{E}$, where $\beta$ is a parameter between 0 and 1 and $n$ is the number of variables.

```
procedure GRASP+PR(MaxIter,RandomSeed)
1     c_best := 0;
2     E := ∅;
3     do k = 1, ..., MaxIter →
4           x :=ConstructSolution(RandomSeed);
5           x :=LocalSearch(x);
6           if |E| =MaxElite →
7                 x :=PathRelinking(x, E);
8                 AddToElite(E, x);
9           else
10                E := E ∪ {x};
11          endif;
12          if c(x) > c_best →
13                x_best := x;
14                c_best := c(x_best);
15          endif;
16    od;
17    return x_best
end GRASP+PR;
```

**Fig. 3.** Pseudo-code of GRASP with path-relinking for maximization problem

If $\mathbf{y}$ satisfies either of the above, it then replaces an elite solution $\mathbf{z}$ of weight not greater than $c(\mathbf{y})$ and most similar to $\mathbf{y}$, i.e. $\mathbf{z} = \text{argmin}\{d(\mathbf{y}, \mathbf{w}) : \mathbf{w} \in \mathcal{E}$ such that $c(\mathbf{w}) \leq c(\mathbf{y})\}$.

## 4 Computational Results

In this section, we report on an experiment designed to determine the effect of path-relinking on the convergence of the GRASP for MAX-SAT described in [25][2]. After downloading the Fortran source code, we modified it to enable recording of the elapsed time between the start of the first GRASP iteration and when a solution is found having weight greater or equal to a given target value. We call this pure GRASP implementation `grasp`. Using `grasp` as a starting point, we implemented path-relinking making use of the local search code in `grasp`. The GRASP with path-relinking implementation is called `grasp+pr`. To simplify the path-relinking step, we use $\beta = 1$ when testing if a solution can be placed in the elite set. This way only improving solutions are put in the elite set. We were careful to implement independent random number sequences for the pure GRASP and the path-relinking portions of the code. This way, if the same random number generator seeds are used for the GRASP portion of the code,

[2] The Fortran subroutines for the GRASP for MAX-SAT described in [25] can be downloaded from `http://www.research.att.com/~mgcr/src/maxsat.tar.gz`.

**Table 1.** Test problems used in experiment. For each problem, the table lists its name, number of variables, number of clauses, the target weight used as a stopping criterion, and the percentage deviation of the target from the optimal solution

| problem | variables | clauses | target | rel. error |
|---|---|---|---|---|
| `jnh1` | 100 | 800 | 420739 | 0.044% |
| `jnh10` | 100 | 800 | 420357 | 0.115% |
| `jnh11` | 100 | 800 | 420516 | 0.056% |
| `jnh12` | 100 | 800 | 420871 | 0.013% |
| `jnh201` | 100 | 850 | 394222 | 0.047% |
| `jnh202` | 100 | 850 | 393870 | 0.076% |
| `jnh212` | 100 | 850 | 394006 | 0.059% |
| `jnh304` | 100 | 900 | 444125 | 0.092% |
| `jnh305` | 100 | 900 | 443815 | 0.067% |
| `jnh306` | 100 | 900 | 444692 | 0.032% |

the GRASP solutions produced in each iteration are identical for the GRASP and GRASP with path-relinking implementations. Consequently, GRASP with path-relinking will never take more iterations to find a target value solution than the pure GRASP. Since the time for one GRASP with path-relinking iteration is greater than for one pure GRASP iteration, we seek to determine if the potential reduction in number of iterations of GRASP with path-relinking will suffice to make the total running time of GRASP with path-relinking smaller than that of pure GRASP.

The Fortran programs were compiled with the `g77` compiler, version 3.2.3 with optimization flag `-O3` and run on a SGI Altix 3700 Supercluster running RedHat Advanced Server with SGI ProPack. The cluster is configured with 32 1.5-GHz Itanium-2 processors (Rev. 5) and 245 Gb of main memory. Each run was limited to a single processor. User running times were measured with the `etime` system call. Running times exclude problem input.

We compared both variants on ten test problems previously studied in [25] [3]. Optimal weight values are known for all problems. The target weight values used in the experiments correspond to solutions found in [25] after 100,000 GRASP iterations and are all near-optimal. Table 2 shows test problem dimensions, target values, and how close to optimal the targets are.

Since `grasp` and `grasp+pr` are both stochastic local search algorithms, we compare their performance by examining the distributions of their running times. For each instance, we make 200 independent runs of each heuristic (using different random number generator seeds) and record the time taken for the run

[3] The test problems can be downloaded from `http://www.research.att.com/~mgcr/data/maxsat.tar.gz`.

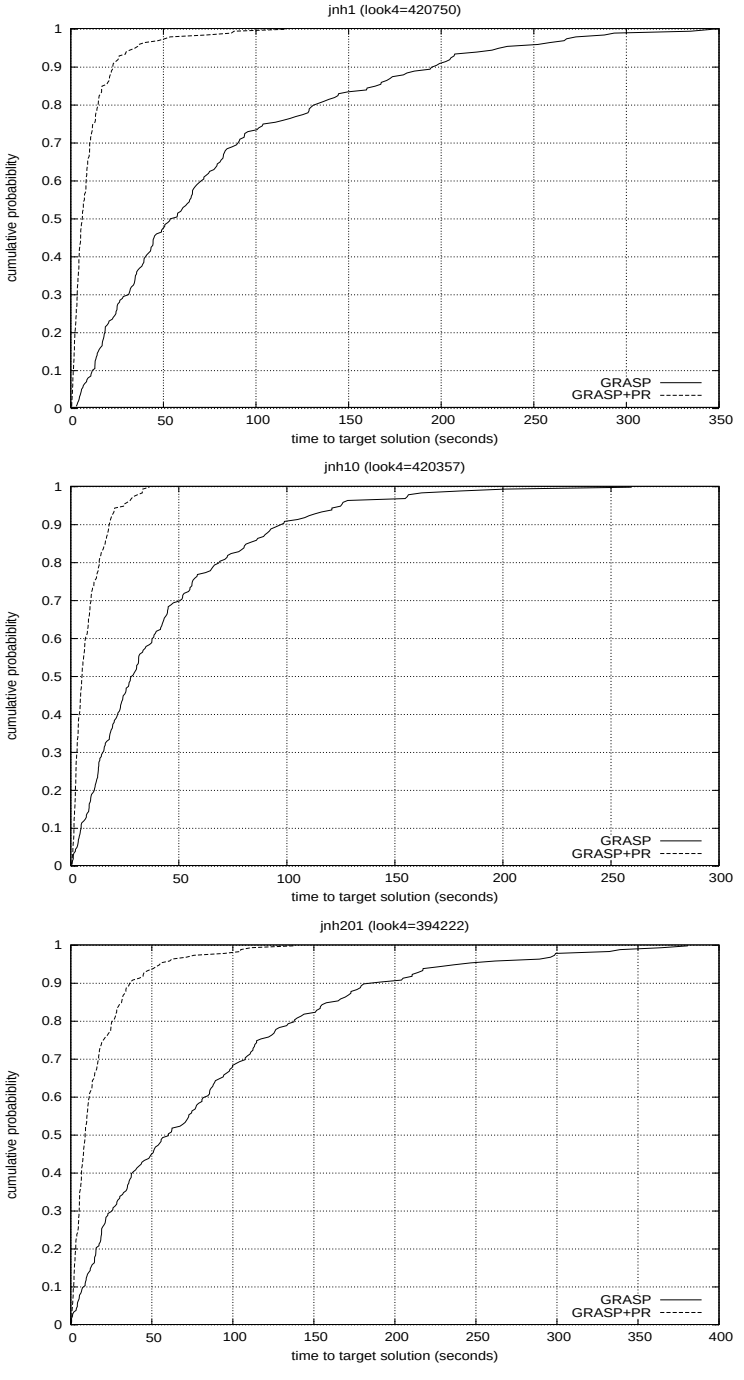



**Fig. 4.** Time to target distributions comparing `grasp` and `grasp+pr` on instances `jnh1`, `jnh10`, and `jnh201`

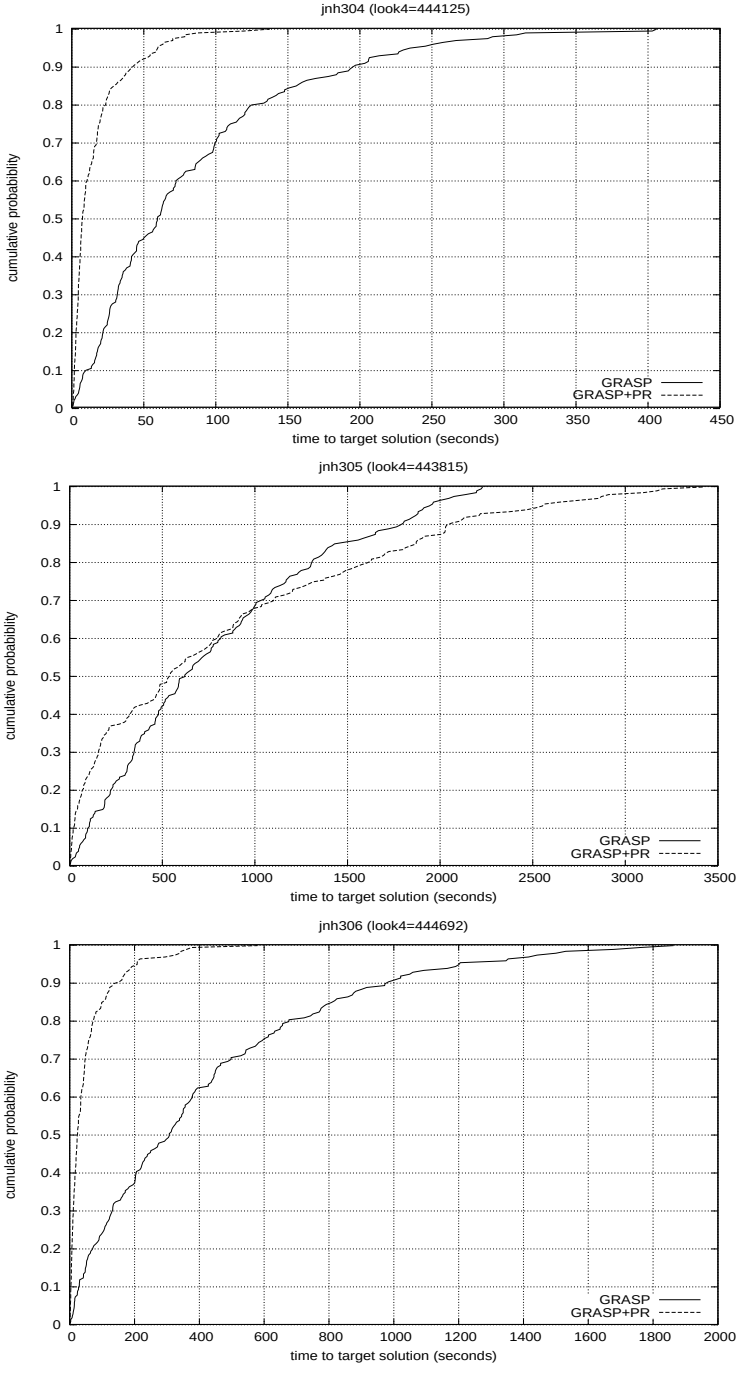



**Fig. 5.** Time to target distributions comparing `grasp` and `grasp+pr` on instances `jnh304`, `jnh305`, and `jnh306`

to find a solution with weight at least as large as the given target value. For each instance/heuristic pair, the running times of each heuristic are sorted in increasing order. We associate with the $i$-th sorted running time ($t_i$) a probability $p_i = (i - \frac{1}{2})/200$, and plot the points $z_i = (t_i, p_i)$, for $i = 1, \ldots, 200$. These plots are called the time to target plots and were first introduced in [10]. These plots display the empirical probability distributions of the random variable *time to target solution*. Figures 4 and 5 are time to target plots for a subset of the test instances.[4]

We make the following observations about the experiments.

- Each heuristic was run a total of 2000 times in the experiments.
- Though the maximum number of GRASP iterations was set to 200,000, both algorithms took much less than that to find truth assignments with total weight at least as large as the target weight on all 200 runs on each instance.
- On all but one instance, the time to target curves for `grasp+pr` were to the left of the curves for `grasp`.
- The relative position of the curves implies that, given a fixed amount of computing time, `grasp+pr` has a higher probability than `grasp` of finding a target solution. For example, consider instance `jnh1` in Figure 4. The probabilities of finding a target at least as good as 420750 in at most 50 seconds are 48% and 97%, respectively, for `grasp` and `grasp+pr`. In at most 100 seconds, these probabilities increase to 73% and 99%, respectively.
- The relative position of the curves also implies that, given a fixed probability of finding a target solution, the expected time taken by `grasp` to find a solution with that probability is greater than the time taken by `grasp+pr`. For example, consider instance `jnh306` in Figure 5. For `grasp` to find a target solution with 50% probability we expect it to run for 329 seconds, while `grasp+pr` we expect a run of only 25 seconds. For 90% probability, `grasp` is expected to run for 984 seconds while `grasp+pr` only takes 153 seconds.
- The only instance on which the time to target plots intersect was `jnh305`, where `grasp+pr` took longer to converge than the longest `grasp` run on 21 of the 200 runs, Still, two thirds of the `grasp+pr` were faster than `grasp`.

## References

1. R.M. Aiex, S. Binato, and M.G.C. Resende. Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing*, 29:393–430, 2003.
2. R.M. Aiex, M.G.C. Resende, P.M. Pardalos, and G. Toraldo. GRASP with path relinking for three-index assignment. *INFORMS J. on Computing*, 2005. In press.

[4] The raw data as well as the plots of the distributions for all of the test problems are available at `http://www.research.att.com/~mgcr/exp/gmaxsatpr`.

3. T. Asano. Approximation algorithms for MAX-SAT: Yannakakis vs. Goemans-Williamson. In *5th IEEE Israel Symposium on the Theory of Computing and Systems*, pages 24–37, 1997.
4. R. Battiti and M. Protasi. Reactive search, a history-sensitive heuristic for MAX-SAT. *ACM Journal of Experimental Algorithms*, 2(2), 1997.
5. R. Battiti and M. Protasi. Approximate algorithms and heuristics for the MAX-SAT. In D.Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 77–148. Kluwer Academic Publishers, 1998.
6. S.A. Canuto, M.G.C. Resende, and C.C. Ribeiro. Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks*, 38:50–58, 2001.
7. J. Chen, D. Friesen, and H. Zheng. Tight bound on johnson's algorithm for MAX-SAT. In *Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, pages 274–281, 1997.
8. S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
9. U. Feige and M.X. Goemans. Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT. In *Proceeding of the Third Israel Symposium on Theory of Computing and Systems*, pages 182–189, 1995.
10. T.A. Feo, M.G.C. Resende, and S.H. Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42:860–878, 1994.
11. M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.
12. I.P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 28–33, 1993.
13. F. Glover. Tabu search and adaptive memory programming: Advances, applications and challenges. In R.S. Barr, R.V. Helgason, and J.L. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer Academic Publishers, 1996.
14. M.X. Goemans and D.P. Williamson. A new $\frac{3}{4}$ approximation algorithm for the maximum satisfiability problem. *SIAM Journal on Discrete Mathematics*, 7:656–666, 1994.
15. M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of Association for Computing Machinery*, 42(6):1115–1145, 1995.
16. P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
17. J. Hastad. Some optimal inapproximability results. *Journal of the ACM*, 48:798–859, 2001.
18. D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
19. D.S. Johnson and M.A. Trick, editors. *Cliques, coloring, and Satisfiability: Second DIMACS Implementation Challenge*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.
20. H. Karloff and U. Zwick. A $\frac{7}{8}$-approximation algorithm for MAX-3SAT. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 406–415, 1997.
21. M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

22. M.G.C. Resende and T.A. Feo. A GRASP for Satisfiability. In D.S. Johnson and M.A. Trick, editors, *Cliques, coloring, and Satisfiability: Second DIMACS Implementation Challenge*, number 26 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 499–520. American Mathematical Society, 1996.
23. M.G.C. Resende and L.S. Pitsoulis. Greedy randomized adaptive search procedures. In P.M. Pardalos and M.G.C. Resende, editors, *Handbook of Applied Optimization*, pages 168–183. Oxford University Press, 2002.
24. M.G.C. Resende, L.S. Pitsoulis, and P.M. Pardalos. Approximate solutions of weighted MAX-SAT problems using GRASP. In D.-Z. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 393–405. American Mathematical Society, 1997.
25. M.G.C. Resende, L.S. Pitsoulis, and P.M. Pardalos. Fortran subroutines for computing approximate solutions of weighted MAX-SAT problems using GRASP. *Discrete Applied Mathematics*, 100:95–113, 2000.
26. M.G.C. Resende and C.C. Ribeiro. A GRASP with path-relinking for private virtual circuit routing. *Networks*, 41:104–114, 2003.
27. M.G.C. Resende and C.C. Ribeiro. GRASP and path-relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers*, pages 29–63. Springer, 2005.
28. C.C. Ribeiro, E. Uchoa, and R.F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing*, 14:228–246, 2002.
29. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability instances. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.
30. W.M. Spears. Simulated annealing for hard satisfiability problems. In D.S. Johnson and M.A. Trick, editors, *Cliques, coloring, and Satisfiability: Second DIMACS Implementation Challenge*, number 26 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 533–555. American Mathematical Society, 1996.
31. L. Trevisan. Approximating satisfiable satisfiability problems. *Algorithmica*, 28(1):145–172, 2000.
32. M. Yannakakis. On the approximation of maximum Satisfiability. In *Proceedings of the Third ACM-SIAM Symposium on Discrete Algorithms*, pages 1–9, 1992.

# New Bit-Parallel Indel-Distance Algorithm

Heikki Hyyrö[1], Yoan Pinzon[2,⋆], and Ayumi Shinohara[1,3]

[1] PRESTO, Japan Science and Technology Agency (JST), Japan
helmu@cs.uta.fi
[2] Department of Computer Science, King's College, London, UK
pinzon@dcs.kcl.ac.uk
[3] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
ayumi@i.kyushu-u.ac.jp

**Abstract.** The task of approximate string matching is to find all locations at which a pattern string $p$ of length $m$ matches a substring of a text string $t$ of length $n$ with at most $k$ differences. It is common to use Levenshtein distance [5], which allows the differences to be single-character insertions, deletions, substitutions. Recently, in [3], the IndelMYE, IndelWM and IndelBYN algorithms where introduced as modified version of the bit-parallel algorithms of Myers [6], Wu&Manber [10] and Baeza-Yates&Navarro [1], respectively. These modified versions where made to support the indel distance (only single-character insertions and/or deletions are allowed). In this paper we present an improved version of IndelMYE that makes a better use of the bit-operations and runs 24.5 percent faster in practice. In the end we present a complete set of experimental results to support our findings.

## 1 Introduction

The *approximate string matching problem* is to find all locations in a text of length $n$ that contain a substring that is similar to a query pattern string $p$ of length $m$. Here we assume that the strings consist of characters over a finite alphabet. In practice the strings could for example be English words, DNA sequences, source code, music notation, and so on. The most common similarity measure between two strings is known as Levenshtein distance [5]. It is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to transform one of the strings into the other. In a comprehensive survey by Navarro [7], the $O(k\lceil m/w\rceil n)$ algorithm of Wu and Manber (WM) [10], the $O(\lceil (k+2)(m-k)/w\rceil n)$ algorithm of Baeza-Yates and Navarro (BYN) [1], and the $O(\lceil m/w\rceil n)$ algorithm of Myers (MYE) [6] were identified as the most practical verification capable approximate string matching algorithms under Levenshtein distance. Here $w$ denotes the computer word size. Each of these algorithms is based on so-called *bit-parallelism*. Bit-parallel algorithms make use

⋆ Part of this work was done while visiting Kyushu University. Supported by PRESTO, Japan Science and Technology Agency (JST).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 380–390, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of the fact that a single computer instruction operates on bit-vectors of $w$ bits, where typically $w = 32$ or 64 in the current computers. The idea is to achieve gain in time and/or space by encoding several data-items of an algorithm into $w$ bits so that they can be processed in parallel within a single instruction (thus the name bit-parallelism).

In [3] the three above-mentioned bit-parallel algorithms were extended to support the indel distance. In this paper we improve the running time of one of those algorithms, namely, IndelMYE. IndelMYE is a modify version of Myers algorithm [6] that supports the indel distance instead of the more general Levenshtein distance. The new version (called IndelNew) is able to compute the horizontal differences of adjacent cell in the dynamic programming matrix more efficiently. Hence, the total number of bit-operations decreases from 26 to 21. We run extensive experiments and show that the new algorithms has a very steady performance in all cases, achieving and speedup of up to 24.5 percent compare with its previous version.

This paper is organised as follows. In Section 2 we present some preliminaries. In Sections 3 we explain the main bit-parallel ideas used to create the new algorithm presented in Section 4. In Section 5 we present extensive experimental results for the three bit-parallel variants presented in [3] and two dynamic programming algorithms. Finally, in Section 6 we give our conclusions.

## 2 Preliminaries

We will use the following notation with strings. We assume that strings are sequences of characters from a finite character set $\Sigma$. The alphabet size, *i.e.* the number of distinct characters in $\Sigma$, is denoted by $\sigma$. The $i$th character of a string $s$ is denoted by $s_i$, and $s_{i..j}$ denotes the substring of $s$ that begins at its $i$th position and end at its $j$th position. The length of string $s$ is denoted by $|s|$. The first character has index 1, and so $s = s_{1...|s|}$. A length-zero empty string is denoted by $\varepsilon$.

Given two strings $s$ and $u$, we denote by $ed(s, u)$ the edit distance between $s$ and $u$. That is, $ed(s, u)$ is defined as the minimum number of single-character insertions, deletions and/or substitutions needed in order to transform $s$ into $u$ (or vice versa). In similar fashion, $id(s, u)$ denotes the indel distance between $s$ and $u$: the minimum number of single-character insertions and/or deletions needed in transforming $s$ into $u$ (or vice versa).

The problem of approximate searching under indel distance can be stated more formally as follows: given a length-$m$ pattern string $p_{1..m}$, a length-$n$ text string $t_{1..n}$, and an error threshold $k$, find all text indices $j$ for which $id(p, t_{j-h..j}) \leq k$ for some $1 \leq h < j$. Fig. 1 gives an example with $p =$"ACGC", $t =$"GAAGCGACTGCAAACTCA", and $k = 1$. Fig. 1(b) shows that under indel distance $t$ contains two approximate matches to $p$ at ending positions 5 and 11. In the case of regular edit distance, which allows also substitutions, there is an additional approximate occurrence that ends at position 17 (see Fig. 1(a)). Note that Fig. 1 shows a minimal *alignment* for each occurrence. For strings $s$ and $u$,

the characters of $s$ and $u$ that correspond to each other in a minimal transformation of $s$ into $u$ are vertically aligned with each other. In case of indel distance and transforming $s$ into $u$, $s_i$ corresponds to $u_j$ if $s_i$ and $u_j$ are matched, $s_i$ corresponds to $\varepsilon$ if $s_i$ is deleted, and $\varepsilon$ corresponds to $u_j$ if $u_j$ is inserted to $s$. In case of Levenshtein distance, $s_i$ corresponds to $u_j$ also if $s_i$ is substituted by $u_j$.

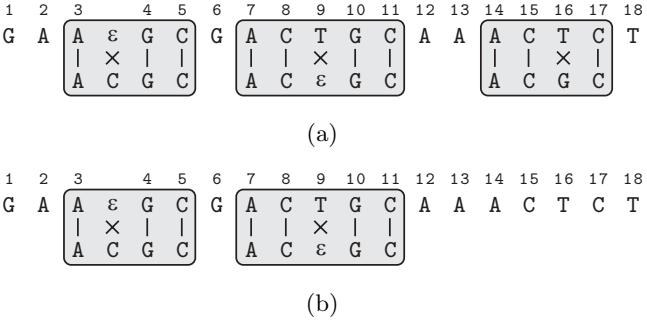



**Fig. 1.** Example of approximate string matching with $k = 1$ difference under (a) Levenshtein distance and (b) indel distance. Grey boxes show the matches and corresponding alignments. In the alignments we show a straight line between corresponding characters that match, and a cross otherwise. Hence the number of crosses is equal to the number of differences

We will use the following notation in describing bit-operations: '&' denotes bitwise "AND", '|' denotes bitwise "OR", '$\hat{}$' denotes bitwise "XOR", '$\sim$' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. The $i$th bit of the bit vector $V$ is referred to as $V[i]$ and bit-positions are assumed to grow from right to left. In addition we use superscript to denote bit-repetition. As an example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^2 1^3 0$. Fig. 2 shows a simple high-level scheme for bit-parallel algorithms. In the subsequent sections we will only show the sub-procedures for preprocessing and updating the bit-vectors.

---

**Algo-BitParallelSearch**($p_1 \dots p_m$, $t_1 \dots t_n$, $k$)
1. ▷ Preprocess bit-vectors
2. **Algo-PreprocessingPhase()**
3. **For** $j \in 1 \dots n$ **Do**
4. ▷ Update bit-vectors at text character $j$ and check if a match was found
5. **Algo-UpdatingPhase()**

---

**Fig. 2.** A high-level template for bit-parallel approximate string matching algorithms

## 3 Bit-Parallel Dynamic Programming

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms in practice for the Levenshtein edit distance [5]. The first of these was the $O(kn(m/w))$ algorithm of Wu & Manber [10], where $w$ is the computer word size. Later Wright [9] presented an $O(mn \log_\sigma /w))$ algorithm, where $\sigma$ is the alphabet size. Then Baeza-Yates & Navarro followed with their $O((km/w)n)$ algorithm. Finally Myers [6] achieved an $O((m/w)n)$ algorithm, which is an optimal speedup from the basic $O(m/n)$ dynamic programming algorithm. With the exception of the algorithm of Wright, the bit-parallel algorithms dominate the other verification capable algorithms with moderate pattern lengths [7].

The $O(\lceil m/w \rceil n)$ algorithm of Myers [6] is based on a bit-parallelization of the dynamic programming matrix $D$. The $O(k\lceil m/w \rceil n)$ algorithm of Wu and Manber [10] and the $O(\lceil (k+2)(m-k)/w \rceil n)$ algorithm of Baeza-Yates and Navarro [1] simulate a non-deterministic finite automaton (NFA) by using bit-vectors.

For typical edit distances, their dynamic programming recurrence confines the range of possible differences between two neighboring cell-values in $D$ to be small. Fig. 3 shows the possible difference values for some common distances. For both Levenshtein and indel distance, {-1,0,1} is the possible range of values for vertical differences $D[i,j] - D[i-1,j]$ and horizontal differences $D[i,j] - D[i,j-1]$. The range of diagonal differences $D[i,j] - D[i-1,j-1]$ is {0,1} in the case of Levenshtein distance, but {0,1,2} in the case of indel distance.

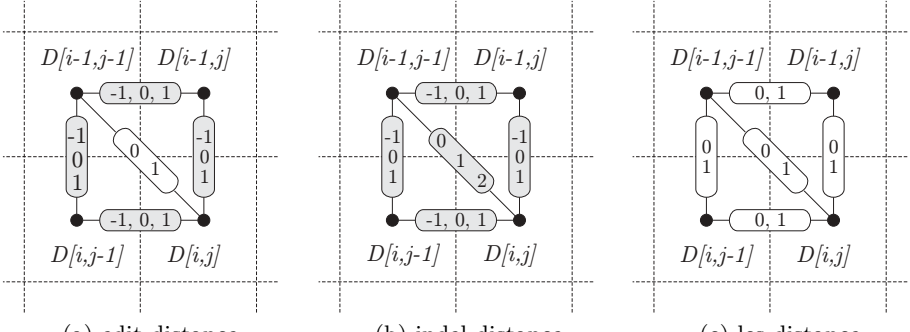



**Fig. 3.** Differences between adjacent cells. White/grey boxes indicate that one/two bit-vectors are needed to represent the differences

The bit-parallel dynamic programming algorithm of Myers (MYE) makes use of the preceding observation. In MYE the values of matrix $D$ are expressed implicitly by recording the differences between neighboring cells. And moreover, this is done efficiently by using bit-vectors. In [4], a slightly simpler variant of

MYE, the following length-$m$ bit-vectors $Zd_j$, $Nh_j$, $Ph_j$, $Nv_j$, and $Pv_j$ encode the vertical, horizontal and diagonal differences at the current position $j$ of the text:

— $Zd_j[i] = 1$ iff $D[i,j] - D[i-1,j-1] = 0$
— $Ph_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 1$
— $Nh_j[i] = 1$ iff $D[i,j] - D[i,j-1] = -1$
— $Pv_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 1$
— $Nv_j[i] = 1$ iff $D[i,j] - D[i-1,j] = -1$

The crux of MYE is that these difference vectors can be computed efficiently. The basic idea is that, given the vertical difference $D[i-1,j] - D[i-1,j-1]$ (left vertical difference in Fig. 4), the diagonal difference $D[i,j] - D[i-1,j-1]$ fixes the value of the horizontal difference $D[i,j] - D[i,j-1]$. And subsequently, in symmetric fashion, the diagonal difference also fixes the vertical difference $D[i,j] - D[i-1,j]$ after the previous horizontal difference $D[i,j] - D[i,j-1]$ is known. These observations determine the order in which MYE computes the difference vectors. The overall scheme is as follows. The algorithm maintains only the value of interest, $D[m,j]$, explicitly during the computation. The initial value $D[m,0] = m$ and the initial vectors $Pv_0 = 1^m$ and $Nv_0 = 0^m$ are known from the dynamic programming boundary values. When arriving at text position $j > 0$, MYE first computes the diagonal vector $Zd_j$ by using $Pv_{j-1}$, $Nv_{j-1}$ and $M(t_j)$, where for each character $\lambda$, $M(\lambda)$ is a precomputed length-$m$ match vector where $M(\lambda)_i = 1$ iff $p_i = \lambda$. Then the horizontal vectors $Ph_j$ and $Nh_j$ are computed by using $Zd_j$, $Pv_{j-1}$ and $Nv_{j-1}$. Finally the vertical vectors $Pv_j$ and $Nv_j$ are computed by using $Zd_j$, $Nh_j$ and $Ph_j$. The value $D[m,j]$ is maintained incrementally during the process by setting $D[m,j] = D[m,j-1] + (Ph_h[m] - Nh_h[m])$ at text position $j$. A match of the pattern with at most $k$ errors is found at position $j$ whenever $D[m,j] \leq k$. Fig. 5 shows the complete MYE algorithm.

At each text position $j$, MYE makes a constant number of operations on bit-vectors of length-$m$. This gives the algorithm an overall time complexity $O(\lceil m/w \rceil n)$ in the general case where we need $\lceil m/w \rceil$ length-$w$ bit-vectors in order to represent a length-$m$ bit-vector. This excluded the cost of preprocessing the $M(\lambda)$ vectors, which is $O(\lceil m/w \rceil \sigma + m)$. The space complexity is dominated by the $M(\lambda)$ vectors and is $O(\lceil m/w \rceil \sigma)$. The difference vectors require $O(\lceil m/w \rceil)$ space during the computation if we overwrite previously computed vectors as soon as they are no longer needed.

## 4 IndelNew Algorithm

In this section we will present IndelNew, our faster version for IndelMYE which at the same time was a modification of MYE to use indel distance instead of Levenshtein distance.

As we noted before, indel distance allows also the diagonal difference $D[i,j] - D[i-1,j-1] = 2$. Fig. 4 is helpful in observing how this complicates the compu-

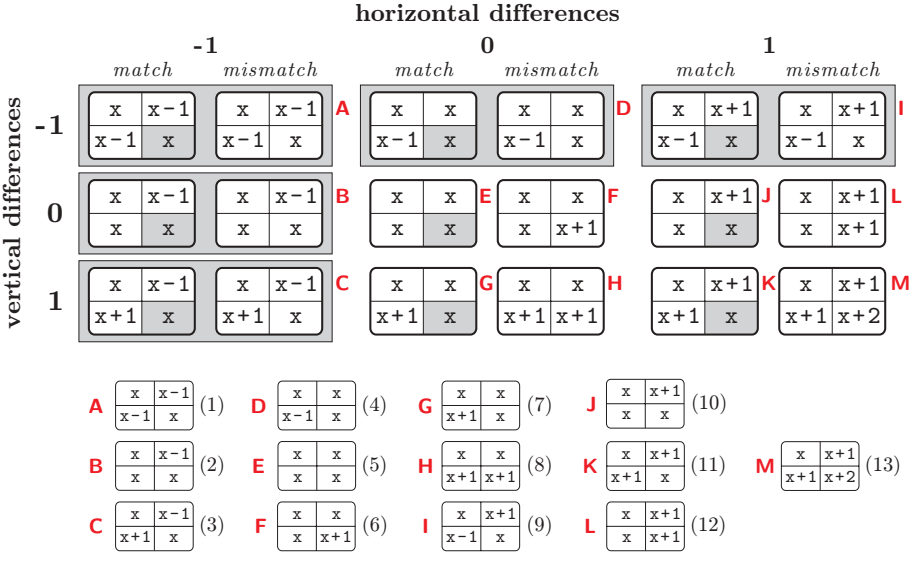



**Fig. 4.** The 13 possible cases when computing a $D$-cell

tation of the difference vectors. It shows the 13 different cases that can occur in a $2\times2$ submatrix $D[i-1..i, j-1..j]$ of $D$. The cases are composed by considering all 18 possible combinations between the left/uppermost vertical/horizontal differences ($D[i, j-1]-D[i-1, j-1]/D[i-1, j]-D[i-1, j-1]$) and a match/mismatch between the characters $p_i$ and $t_j$, some cases occur more that once so only 13 of them are unique.

We note that **M** is the only case where the diagonal difference is +2, and further that **M** is also the only case that is different between indel and Levenshtein distances: in all other cases the value $D[i, j]$ is the same regardless of whether substitutions are allowed or not. And since the diagonal, horizontal and vertical differences in the case **M** have only positive values, IndelNew can compute the 0/-1 difference vectors $Zd_j$, $Nh_j$, and $Nv_j$ exactly as MYE. In the case of Levenshtein distance, the value $D[i, j]$ would be $x + 1$ in case **M**, and hence the corresponding low/rightmost differences $D[i, j]-D[i, j-1]$ and $D[i, j]-D[i-1, j]$ would be zero. This enables MYE to handle the case **M** implicitly, as it computes only the -1/+1 difference vectors. But IndelNew needs to explicitly deal with the case **M** when computing the +1 difference vectors $Ph_j$ and $Pv_j$, unless these vectors are computed implicitly/indirectly. The latter approach was employed in IndelMYE algorithm [3] by using vertical and horizontal zero difference vectors $Zv_j$ and $Zh_j$, where $Zv_j[i] = 1$ iff $D[i, j] - D[i-1, j] = 0$, and $Zh_j[i] = 1$ iff $D[i, j]-D[i, j-1] = 0$. Then, solutions where found for computing $Zv_j$ and $Zh_j$, and the positive difference vectors were then computed simply as $Ph_j = \sim (Zh_j \mid Nh_j)$ and $Pv_j = \sim (Zv_j \mid Nv_j)$. For IndelNew we propose the following more efficient solution for computing $Ph_j$ and $Pv_j$ directly. The discussion assumes that $0 < i \leq m$ and $0 < j \leq n$.

**MYE-PreprocessingPhase**
1. **For** $\lambda \in \Sigma$ **Do** $M(\lambda) \leftarrow 0^m$
2. **For** $i \in 1 \ldots m$ **Do** $M(p_i) \leftarrow M(p_i) \mid 0^{m-i}10^{i-1}$
3. $Pv_0 \leftarrow 1^m,\ Nv_0 \leftarrow 0^m,\ currDist \leftarrow m$

**MYE-UpdatingPhase**
1. $Zd_j \leftarrow (((M(t_j)\ \&\ Pv_{j-1}) + Pv_{j-1})\ \hat{}\ Pv_{j-1})\ \mid M(t_j) \mid Nv_{j-1}$
2. $Nh_j \leftarrow Pv_{j-1}\ \&\ Zd_j$
3. $Ph_j \leftarrow Nv_{j-1} \mid \sim(Pv_{j-1} \mid Zd_j)$
4. $Nv_j \leftarrow (Ph_j << 1)\ \&\ Zd_j$
5. $Pv_j \leftarrow (Nh_j << 1) \mid \sim((Ph_j << 1) \mid Zd_j)$
6. **If** $Ph_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
7. **If** $Nh_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
8. **If** $currDist \leq k$ **Then** Report a match at position $j$

**Fig. 5.** MYE algorithm. Variable $currDist$ keeps track of the value $D[m, j]$. The algorithm representations could be optimized to reuse the value $Ph_j << 1$ so that it is computed only once

**Computing $Ph_j$.** We may observe from Fig. 4 that $Ph_j[i] = 1$ in the six cases **A, D, I, F, L,** and **M**. Cases **A, D,** and **I** arise from the negative vertical difference in column $j-1$, *i.e.* $Nv_{j-1}[i] = 1$. Cases **F** and **L** arise from a zero vertical difference in column $j-1$, i.e. $Nv_{j-1}[i] = 1$ and $Pv_{j-1}[i] = 0$, together with a positive diagonal difference, *i.e.* $Zd_j[i] = 0$. Hence the formula

$$Nv_{j-1} \mid (\sim Nv_{j-1}\ \&\ \sim Pv_{j-1}\ \&\ \sim Zd_j) = Nv_{j-1} \mid \sim(Pv_{j-1} \mid Zd_j)$$

covers the first five cases for the complete vectors, and this is enough for MYE under Levenshtein distance. Case **M** arises from having a positive difference in column $j-1$, a positive horizontal difference in row $i-1$, and a non-zero diagonal difference. This translates into the formula $Pv_{j-1}\ \&\ (Ph_j << 1)\ \&\ \sim Zd_j$, which contains a slightly problematic self-reference to $Ph_j$. We solve it as follows.

The self-reference states that case **M** can be true on row $i$ only if one of the other five cases has happened above $i$. Let $X$ be an auxiliary length-$m$ bit-vector that covers the five cases, that is,

$$X = Nv_{j-1} \mid \sim(Pv_{j-1} \mid Zd_j).$$

Let $Y$ be another auxiliary bit-vector so that

$$Y = Pv_{j-1}\ \&\ \sim Zd_j.$$

Now each set bit $Ph_j[i] = 1$ can be assigned to a distinct region $Ph_j[a..b] = 1^{b-a+1}$ of consecutive set bits in such manner, that $1 \leq a \leq i \leq b \leq m$, $X[a] = 1$, $Y[a+1..b] = 1^{b-a}$ if $a < b$, and $Y[b+1] = 0$ if $b < m$. Moreover, the conditions $Y[a+1..b] = 1^{b-a}$ and $X[a] = 1$ are sufficient to imply that $Ph_j[a..b] = 1^{b-1+1}$. If we now shift the bit region $Y[a+1..b]$ one step right to overlap the positions

$a \ldots b-1$ and then perform an arithmetic addition $Y[a..b] + X[a..b]$, the result is that the bits $Y[a..b-1]$ will change from 1 to 0 and the bit $Y[b]$ from 0 to 1. These changed bits can be set to 1, and thus to be correct values for $Ph_j[a..b]$, by performing XOR. Hence we have the formula

$$Ph_j = (X + Y) \hat{} Y,$$

where $Y$ has already been shifted one step right. We further note that if $Nh_j = Pv_{j-1}$ & $Zd_j$ has already been computed, we may set $Y = Pv_{j-1}$ & $\sim Zd_j = Pv_{j-1} - Nh_j$ in the beginning.

**Computing $Pv_j$.** This step is diagonally symmetric with the case of $Ph_j$. After similar observations from Fig. 4 as before, the six relevant cases are seen to be **A, B, C, F, H**, and **M**, and the first five of these are covered by the formula $(Nh_j << 1) \mid \sim ((Ph_j << 1) \mid Zd_j)$. This time, case **M** has the formula $(Ph_j << 1)$ & $Pv_{j-1}$ & $\sim Zd_j$, which is straighforward to compute. As with the auxiliary variable $Y$, we may again use the fact that $Pv_{j-1}$ & $\sim Zd_j = Pv_{j-1} - Nh_j$. Then the complete formula for $Pv_j$ becomes

$$Pv_j = (Nh_j << 1) \mid \sim ((Ph_j << 1) \mid Zd_j) \mid ((Ph_j << 1) \,\&\, (Pv_{j-1} - Nh_j)).$$

Fig. 7 shows the complete algorithm IndelNew for computing the difference vectors $Zd_j$, $Nh_j$, $Ph_j$, $Nv_j$, and $Pv_j$ at text position $j$ under indel distance. Obviously IndelNew has the same asymptotical time and space complexities as IndelMYE. Fig. 6 shows the complete algorithm IndelMYE as presented in [3]. IndelNew algorithm is able to compute the positive vectors directly. IndelMYE main drawback is the way the horizontal solution is computed. All in all, the total number of bit-operations is 26 for IndelMYE versus 21 for IndelNew, so we have a more efficient implementation for a bit-parallel indel algorithm.

**IndelMYE-UpdatingPhase**

1. $D' \leftarrow (((K_{T_j} \& Pv) + Pv) \hat{} Pv) \mid K_{T_j} \mid Nv$
2. $X \leftarrow (Pv \,\&\, (\sim D')) >> 1$
3. $Y \leftarrow (Zv \,\&\, D') \mid ((Pv \,\&\, (\sim D')) \,\&\, 0^{m-1}1)$
4. $Zh' \leftarrow (X' + Y') \hat{} X'$
5. $Nh' \leftarrow Pv \,\&\, D'$
6. $Ph' \leftarrow \sim (Zh' \mid Nh')$
7. $Zv' \leftarrow (((Zh' << 1) \mid 0^{m-1}1) \,\&\, D') \mid ((Ph' << 1) \,\&\, Zv \,\&\, (\sim D'))$
8. $Nv' \leftarrow (Ph' << 1) \,\&\, D'$
9. $Pv' \leftarrow \sim (Zv' \mid Nv')$
10. **If** $Ph' \,\&\, 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
11. **If** $Nh' \,\&\, 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
12. **If** $currDist \leq k$ **Then** Report a match at position $j$

**Fig. 6.** IndelMYE algorithm as presented in [3]

**IndelNew-UpdatingPhase**

1. $Zd_j \leftarrow (((M(t_j)\ \&\ Pv_{j-1}) + Pv_{j-1}) \hat{}\ Pv_{j-1})\ \mid M(t_j) \mid Nv_{j-1}$
2. $Nh_j \leftarrow Pv_{j-1}\ \&\ Zd_j$
3. $X \leftarrow Nv_{j-1} \mid \sim (Pv_{j-1} \mid Zd_j)$
4. $Y \leftarrow (Pv_{j-1} - Nh_j) >> 1$
5. $Ph_j \leftarrow (X + Y) \hat{}\ Y$
6. $Nv_j \leftarrow (Ph_j << 1)\ \&\ Zd_j$
7. $Pv_j \leftarrow (Nh_j << 1) \mid \sim ((Ph_j << 1) \mid Zd_j)\ \mid ((Ph_j << 1)\ \&\ (Pv_{j-1} - Nh_j))$
8. **If** $Ph_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
9. **If** $Nh_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
10. **If** $currDist \leq k$ **Then** Report a match at position $j$

**Fig. 7.** IndelNew algorithm. The value $Pv_{j-1} - Nh_j$ could be reused

## 5 Experiments

We compare IndelNew against several other approximate string matching algorithms for indel distance. They are: IndelWM (our own implementation), IndelMYE (our own implementation), IndelBYN (a modification of the original code by Baeza-Yates and Navarro), and IndelUKK (our own implementation of the cutoff version of Ukkonen [8]). We also implemented a plain dynamic programming algorithm (without bit-parallelism) but it was too slow for the pattern lengths we used, therefore we removed it from the final test.

The computer used for testing was a 3.2Ghz AMD Athlon64 with 1.5 GB RAM running Windows XP. The computer word size was $w$=32. All code was compiled with MS Visual C++ 6.0 and optimization switched on. We tested on three different $\approx$ 20MB texts. The first was composed by repeating the yeast genome twice. The second was built from a sample of Wall Street Journal articles taken from the TREC collection. The third text was random with alphabet size $\sigma = 120$. The tested pattern lengths were $m = 8$, 16 and 32, and we tested over $k$=1 ... $m-2$. The patterns were selected randomly from the text, and each $(m, k)$ combination was timed by taking the average time over searching for 100 patterns.

Fig. 8 shows the results. It can be seen that IndelWM is competitive with low $k$, being always the best when $k$=1. The performance of IndelBYN depends highly on the effectiveness of its "cutoff" mechanism, which in turns depends on the alphabet size $\sigma$. With DNA its performance becomes poor quite quickly when $k$ grows (except when $m$=8 as then $Rd$ always fits into a single computer word. But IndelBYN is always the best when $k > 1$ with random text and moderately large alphabet size $\sigma = 120$. As expected due to its independence on $k$, IndelMYE/IndelNew has a very steady performance in all cases. But IndelNew was 24.5 percent faster that IndelMYE. Hence, IndelNew is the fastest in those cases where $k$ is moderately large.

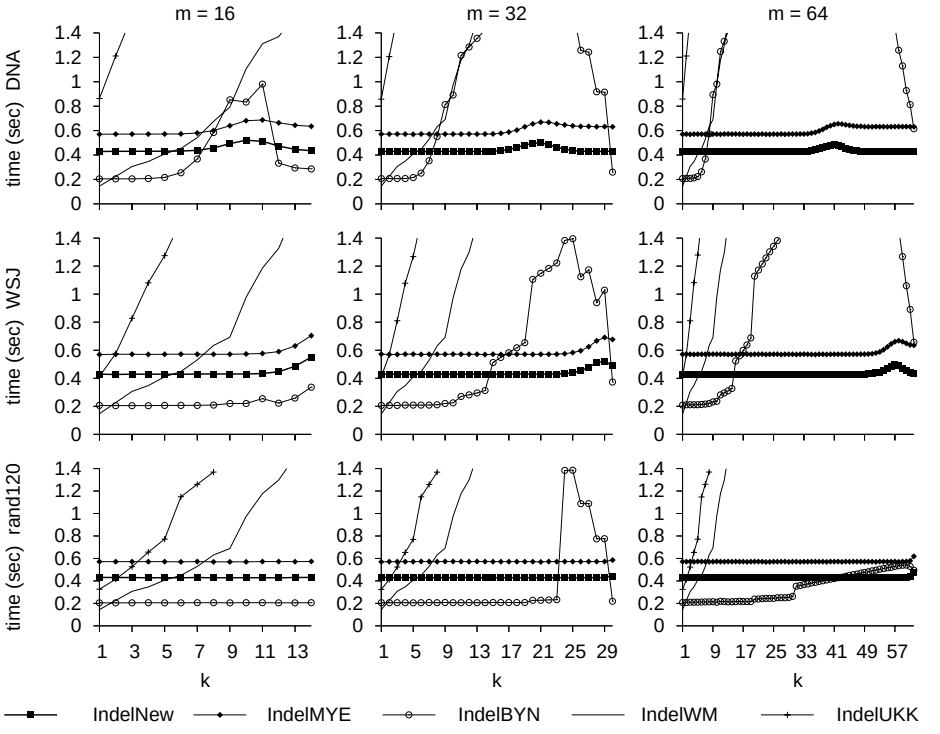



**Fig. 8.** The average time for searching for a pattern in a $\approx 40$ MB text. The first row is for DNA (a duplicated yeast genome), the second row for a sample of Wall Street Journal articles taken from TREC-collection, and the third row for random text with alphabet size $\sigma = 120$

## 6 Conclusions

We have presented a new algorithms based on bit-parallelism that solve the problem of approximate string matching problem with $k$ differences under indel edit distance measure, namely, IndelNew. IndelNew is a more thought version of the early IndelMYE version in [3]. In practice, we showed that the speedup gain by the new version was higher (24.5 percent) than the improvement in the number of bit-operations (about 19 percent – $26 \rightarrow 21$). IndelNew showed a very steady performance in all cases due to its independence on $k$. It is the fastest in those cases where $k$ is moderately large and the cutoff scheme of IndelBYN does not work well.

We plan to use some of the ideas presented in [2] to search several text segments in parallel by encoding several copies of the pattern (or its prefixes) into a single bit-vector. This is left as a future work.

## References

1. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
2. H. Hyyrö, K. Fredriksson and G. Navarro. Increased Bit-Parallelism for Approximate String Matching in *Proc. 3rd Workshop on Efficient and ExperimentalAlgorithms (WEA 2004)*, LNCS 3059, 285–298, 2004.
3. H. Hyyrö, Y. Pinzon and A. Shinohara. Fast Bit-Vector Algorithms for Approximate String Matching under Indel Distance in *Proc. 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, 380–384, 2005.
4. H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
5. V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones (original in Russian). *Russian Problemy Peredachi Informatsii 1*, 12–25, 1965.
6. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
7. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
8. Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
9. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.
10. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

# Dynamic Application Placement Under Service and Memory Constraints

Tracy Kimbrel, Malgorzata Steinder, Maxim Sviridenko, and Asser Tantawi

IBM T.J. Watson Research Center,
P.O. Box 218, Yorktown Heights, NY 10598
{kimbrel, steinder, sviri, tantawi}@us.ibm.com

**Abstract.** In this paper we consider an optimization problem which models the dynamic placement of applications on servers under two simultaneous resource requirements: one that is dependent on the loads placed on the applications and one that is independent. The demand (load) for applications changes over time and the goal is to satisfy all the demand while changing the solution (assignment of applications to servers) as little as possible. We describe the system environment where this problem arises, present a heuristic algorithm to solve it, and provide an experimental analysis comparing the algorithm to previously known algorithms. The experiments indicate that the new algorithm performs much better. Our algorithm is currently deployed in the IBM flagship product Websphere.

## 1 Introduction

With the proliferation of the Web and outsourcing of data services, computing service centers have increased in both size and complexity. Such centers provide a variety of services, for example web content hosting, e-commerce, web applications, and business applications. Managing such centers is challenging since a service provider must manage the quality of service provided to competing applications in the face of unpredictable load intensity and distribution among the various offered services and applications. Several management software packages which deal with these operational management issues have been introduced. These software systems provide functions including monitoring, demand estimation, load balancing, dynamic provisioning, service differentiation, optimized resource allocation, and dynamic application placement. The last function, namely dynamic application placement, is the subject of this paper.

Service requests are satisfied through the execution of one or more instances of each of a set of applications. Applications include access to static and dynamic web content, enterprise applications, and access to database servers. Applications may be provided by HTTP web servers, servlets, Enterprise Java Beans (EJB), or database queries. When the number of service requests for a particular application increases, the application placement management software deploys additional instances of the application in order to accommodate the increased

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 391–402, 2005.
© Springer-Verlag Berlin Heidelberg 2005

load. It is imperative to have an on-demand management environment allowing instances of applications to be dynamically deployed and removed. The problem is to dynamically change the number of application instances so as to satisfy the dynamic load while minimizing the overhead of starting and stopping application instances.

We characterize an application by two parameters, a load-independent requirement of resources required to run an application, and a load-dependent requirement which is a function of the external load or demand placed on the application. Examples of load-independent requirements are memory, communication channels, and storage. Examples of load-dependent requirements are current or projected request rate, CPU cycles, disk activity, and number of execution threads.

We also characterize a server by two parameters: a load-independent capacity which represents the amount of resources used to host applications on the server, and a load-dependent capacity which represents the available capacity to process requests for the applications' services.

In this paper we restrict ourselves to a single load-independent resource, say memory requirements, and a single load-dependent resource, say CPU cycles. These are the most constrained resources for most applications, and thus we are able to focus on these and use our solution in practice even though there are other resource requirements and constraints.

The paper is organized as follows. In Section 2 we describe the Websphere environment in which our algorithm is implemented. In Section 3 we describe related work. In Section 4 we present a mathematical formulation of the problem. We describe our heuristic algorithm in Section 5. Experimental results are given in Section 6. Further work is discussed and conclusions are given in Section 7.

## 2 System Description

Based on the experimental results presented in this paper, our algorithm has been incorporated into the IBM Websphere environment [10]. A Websphere component known as the placement controller receives dynamic information about the load-independent and load-dependent requirements of the various applications, and the load-independent and load-dependent capacities of the various servers. We used memory size and CPU cycles/sec as the representative load-independent and load-dependent parameters, respectively. The placement controller is aware of the configuration, i.e., the mapping of applications onto servers in a given Websphere cell. Upon need, or periodically, the placement controller executes our algorithm in order to determine the change in application placement configuration in response to changes in loads and characteristics of the applications and servers. Then the placement controller realizes the change, automatically or in a supervised mode, through the execution of scripts to start and stop applications servers.

The system includes an application workload predictor and an application profiler. The application workload predictor utilizes historical information re-

garding the offered load to produce a workload prediction for each application supported by the server farm. For instance, the workload prediction can be characterized by the arrival rate of requests to a given application. Similar to the application workload predictor, the application profiler produces a set of application resource requirements by estimating the amount of server resources required by a single request of each application. The application resource requirements includes, for example, the number of CPU cycles required to process a request.

The placement controller utilizes the workload prediction and the application resource requirements provided by the application workload predictor and the application profiler to compute predicted load-dependent resource requirements for each application. Considering the predicted resource requirements for each application, the given capacities of each of the server computing nodes in the server farm, and the current application placement, the placement controller uses the algorithm presented here to compute a new placement of applications.

## 3 Related Work

The problem of optimally placing replicas of objects on servers, constrained by object and server sizes as well as capacity to satisfy a fluctuating demand for objects, has appeared in a number of fields related to distributed computing. In managing video-on-demand systems, replicas of movies are placed on storage devices and streamed by video servers to a dynamic set of clients with a highly skewed movie selection distribution. The goal is to maximize the number of admitted video stream requests. Several movie placement and video stream migration policies have been studied. A disk load balancing criterion which combines a static component and a dynamic component is described in [9]. The static component decides the number of copies needed for each movie by first solving an apportionment problem and then solving the problem of heuristically assigning the copies onto storage groups to limit the number of assignment changes. The dynamic component solves a discrete class-constrained resource allocation problem for optimal load balancing, and then introduces an algorithm for dynamically shifting the load among servers (i.e. migrating existing video streams). A placement algorithm for balancing the load and storage in multimedia systems is described in [5]. The algorithm also minimizes the blocking probability of new requests.

In the area of parallel and grid computing, several object placement strategies (or, meta-scheduling strategies) have been investigated [8, 1]. Communication overhead among objects placed on various machines in a heterogeneous distributed computing environment plays an important role in the object placement strategy. A related problem is that of replica placement in adaptive content distribution networks [**?**, 1]. There the problem is to optimally replicate objects on nodes with finite storage capacities so that clients fetching objects traverse a minimum average number of nodes in such a network. The problem is shown to

be NP-complete and several heuristics have been studied, especially distributed algorithms.

Similar problems have been studied in theoretical optimization literature. The special case of our problem with uniform memory requirements was studied in [6, 7] where some approximation algorithms were suggested. Related optimization problems include bin packing, multiple knapsack and multi-dimensional knapsack problems [2].

## 4 Problem Formulation

We formalize our *dynamic application placement problem* as follows. We are given $m$ servers $1, \ldots, m$ with memory capacities $\Gamma_1, \ldots, \Gamma_m$ and service capacities (number of requests that can be served per unit time) $\Omega_1, \ldots, \Omega_m$. We are also given $n$ applications $1, \ldots, n$ with memory requirements $\gamma_1, \ldots, \gamma_n$. Application $j$ must serve some number of requests $\omega_{jt}$ in time interval $t$.

A feasible solution for the problem at time step $t$ is an assignment of applications' workloads to servers. Each application can be assigned to (replicated on) multiple servers. For every server $i$ that an application $j$ is assigned to, the solution must specify the number $\omega_{itj}$ of requests this server processes for this application. $\sum_i \omega_{ijt}$ must equal $\omega_{jt}$ for all applications $j$ and time steps $t$. For every server the memory and processing constraints must be respected. The sum of memory requirements of applications assigned to server $i$ cannot exceed its memory $\Gamma_i$ and $\sum_j \omega_{ijt}$, i.e. the total number of requests served by this server during the time step $t$, cannot exceed $\Omega_i$. Note that each assignment (copy) of an application to a server incurs the full memory cost, whereas the processing load is divided among the copies.

The objective is to find a solution at time step $t$ which is not very different from the solution at time step $t-1$. More formally, with every feasible solution we associate a bipartite graph $(A, S, E_t)$ where $A$ represents the set of applications, $S$ represents the set of servers, and $E_t$ is a set of edges $(j, i)$ such that application $j$ is assigned to (or has copy on) server $i$ at time step $t$. Our objective function is to minimize $|E_t \ominus E_{t-1}|$, i.e., the cardinality of the symmetric difference of the two edge sets. This is the number of application instances that must be shut down or loaded at time $t$.

### 4.1 Practical Assumptions

Since finding a feasible solution to the static problem (i.e., that of finding an assignment for a single time step) is NP-hard, we must make certain assumptions about the input. Intuitively, if the input is such that even finding a static solution is hard we cannot expect to find a good solution with respect to the dynamic objective function. Thus the problem instance must be "easy enough" that a relatively straightforward heuristic can find a feasible solution at each time step. In practical terms, this means there must be enough resources to easily satisfy the demand if we ignore the quality of the solution in the sense of the dynamic

objective function. In the worst case, we can fall back on the heuristic to find a feasible solution. In fact, our algorithm will degrade gradually to this extreme, but should perform much better in the common case.

## 5 Algorithm

We first describe an algorithm that builds a solution from scratch, i.e. under the assumption that $E_{t-1} = \emptyset$, either because this is the first step ($t = 1$) or because the solution from the previous step $t - 1$ is very bad for serving demands at step $t$. This heuristic will be also used later as a subroutine when we describe an incremental algorithm which optimizes the objective function as we move from step $t - 1$ to $t$. At the risk of slight confusion, we will refer to this heuristic as the initial placement heuristic even when it is used as part of the incremental construction.

### 5.1 Initial Placement

We order all servers by decreasing value of their densities $\Omega_i/\Gamma_i$, and order applications by decreasing densities $\omega_{jt}/\gamma_j$. We load the highest density application $j$ to the highest density server $i$ which has enough memory for that application.

If the available service capacity $\Omega_i$ of a server $i$ is larger then service requirement $\omega_{jt}$ of an application that we assign to the server, then we delete application $j$ from the list of unscheduled applications. We recompute the available memory and service capacities of the server $i$ by subtracting the amounts of resources consumed by application $j$ and insert server $i$ back to the list of servers according to its new density $\Omega_i/\Gamma_i$ with the updated values $\Omega_i$ and $\Gamma_i$.

If the available service capacity $\Omega_i$ of the server $i$ is exceeded by the demand $\omega_{jt}$, we still assign application $j$ to server $i$, but this application's demand served by this server is limited by the server's (remaining) service capacity. We remove the server from the list.

In the latter case that the service capacity on the server $i$ is exceeded by application $j$ assigned to it, let $\omega'_{jt}$ be the amount of demand of application $j$ assigned to this server and let $\omega''_{jt}$ be the remaining demand; note $\omega'_{jt}+\omega''_{jt} = \omega_{jt}$. Since the server $i$ cannot serve all demand of application $j$ we will need to load at least one more copy of it on another server, but we do not yet know which server. The density of the remaining demand is $\omega''_{jt}/\gamma_j$. We place the application back in the list with this value as its density in the sequence of remaining applications (in the appropriate place in the list ordered by densities). Then we move on to the next highest density application, and so on.

The intuition behind the rule is as follows. We should match applications which have many requests per unit of memory with servers which have high processing capacity per unit of memory. It is not wise to assign applications with high density to a low density server, since we would be likely to reach the processing capacity constraint and leave a lot of memory unused on that server. Similarly, if low density applications are loaded on high density servers,

we would be likely to reach the server's memory constraint without using much of the processing capacity.

Note that for every server the algorithm splits the demand of at most one application between this server and some other servers. Thus the total number of application-to-server mappings (edges in the bipartite graph) is at most $n+m-1$.

### 5.2 Incremental Placement

Although the initial placement algorithm is rather conservative in memory allocation, it could be very bad from the viewpoint of the dynamic objective function, which seeks a minimal incremental cost of unloading and loading applications between time steps. We now explain how we can combine the initial placement algorithm with a max flow computation to yield a heuristic for minimizing our objective function.

Given a feasible solution on the previous step $(A, S, E_{t-1})$ we first would like to check whether we can satisfy the new demands $\omega_{jt}$ by simply using the old assignment of applications to servers. We check this by solving a bipartite flow problem. I.e., we use the edge set $E_{t-1}$. Each node corresponding to application $j$ is a source of $\omega_{jt}$ units of flow. We test whether there is a flow satisfying these sources by routing flow to sinks corresponding to the servers, such that the flow into each sink corresponding to a server $i$ is limited by the server's service capacity $\Omega_i$.

If this flow is feasible we are done; the flow values on the edges give the assignments of applications' loads to servers. Otherwise, there is a residual demand for every application (possibly 0 for some) which remains unassigned to servers. Denote the residual demands by $\omega'_{jt}$. For every server there are a residual memory $\Gamma'_i$ and a service capacity $\Omega'_i$ that are not consumed by the assignment given by the flow. Notice that these demands and capacities induce a problem of the same form as the initial placement problem. We apply our greedy initial placement heuristic to this instance. If our heuristic finds a feasible solution to the residual instance, we can construct an overall solution as follows. The residual instance results in a new set of edges, i.e., application-to-server mappings (applications which must be loaded onto servers), which we simply add to the existing edges. The total cost of the new solution is the number of new edges used by the heuristic to route the residual demand. This should not be large since our heuristic is conservative in defining new edges.

If our heuristic fails to find a feasible solution, we delete an edge in the graph $(A, S, E_{t-1})$ and repeat the procedure. We continue in this fashion until a feasible solution is found. The total cost is the number of deleted edges in addition to the number of new edges. In the worst case, we eventually delete all edges in the graph and build the solution from scratch using our initial placement heuristic, which is possible by our assumption that the instance is "not too hard."

It remains to define which edge should be deleted. A good heuristic choice should be the edge which minimizes the ratio of the total demand routed through this edge (i.e., the flow on this edge) divided by the memory requirement of the corresponding application. The intuition for this is that we would like to delete

an edge which uses memory in the most inefficient way. Experimental analysis could find a better function to define a candidate for deletion.

## 6 Experimental Evaluation

### 6.1 Uniform Memory Requirements

In this section we evaluate our proposed algorithm for uniform memory requirements $\gamma_j = const$. We concentrate on its performance in terms of the number of placement changes it suggests and in terms of the execution time. The algorithm is compared against two other placement algorithms we considered for implementation.

The first algorithm, known as the *Noah's Bagels* algorithm, is a solution to a class-constrained Multiple Knapsack problem as defined in [6, 7]. This algorithm is applicable to the application placement problem only after making the assumption that all applications have the same memory requirement $d$, i.e., for every application $j$, $\gamma_j = d$. In consequence, the memory of each server can be expressed as a multiple $d$; normalizing by setting $d = 1$, server $i$ can host $\Gamma_i$ applications. The algorithm fills nodes in increasing order of $\Omega_i$. To fill a node $i$, it considers applications in increasing order of $\omega_j$, and finds the minimum $j$ such that $\sum_{k=j}^{j+\Gamma_i-1} \omega_k \geq \Omega_i$. Applications $j, \ldots, j+\Gamma_i-1$ are loaded on $i$, possibly splitting the demand of $j+\Gamma_i-1$; in this case, $j+\Gamma_i-1$ is put back in the list of applications, sorted according to its remaining unmet demand. This algorithm is known to solve a maximization version of our problem, maximizing $\sum_i \sum_j \omega_{ijt}$, when for every $i$, $\frac{\Omega_i}{\Gamma_i} = \text{const}$ and $\sum_i \Gamma_i \geq n+m-1$. If the above conditions are not satisfied, the algorithm is sub-optimal, but if we allow resource augmentation, i.e., if we increase each $\Gamma_i$ by 1, then this algorithm finds a solution with value lower bounded by the optimal value for the original instance.

Under our assumption on practical inputs, we can use this maximization algorithm to find a feasible solution to our problem (i.e., optimal under the multiple knapsack objective). Our implementation of the *Noah's Bagels* algorithm computes a new placement in each iteration from scratch, not taking the previous placement into account. It may produce a different placement, even if no placement changes are needed to satisfy the new demand. To avoid this unnecessary placement churn, in our implementation, a new placement is computed only if the previous one is unable to satisfy the current demand as witnessed by the existence or non-existence of a solution to the satisfying flow problem.

The second algorithm is a modification of the *Noah's Bagels* algorithm that we developed to make it take the previous placement into account. The algorithm adopts several heuristics to modify the previous placement to satisfy the new demand. It first accepts the old placement on nodes that are well utilized by the new demand. We say a node is well utilized if applications placed on it use the total CPU capacity and no more than the node's available memory, and the CPU load of no more than one application needs to be transfered to

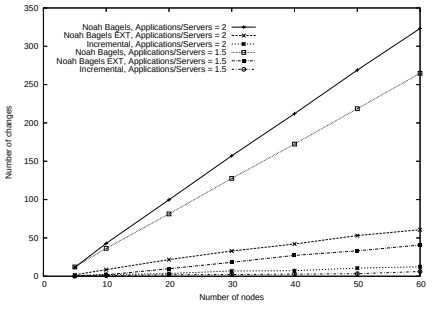
another node. When no more well utilized nodes exist, the algorithm fills unused memory on underutilized nodes using the original *Noah's Bagels* algorithm. If this step does not result in a well utilized node, an application with the smallest CPU demand is removed from the node and the process is repeated until the node is well utilized. When no more underutilized nodes remain, overutilized nodes are processed by removing applications from them, starting from ones with the highest CPU demand, until a node becomes well utilized. We call this new algorithm *Noah's Bagels Ext*. Like the original algorithm, the *Noah's Bagels Ext* algorithm assumes that all applications have the same memory requirements and it guarantees a placement satisfying the demand of all applications if for every $i$, $\frac{\Omega_i}{\Gamma_i} = \text{const}$ and $\sum_i \Gamma_i \geq n + m - 1$. The algorithm *Noah's Bagels Ext* was the previous candidate to be implemented in the placement controller and this why we compare our algorithm with it.

**Experiment Design.** In the experimental study we vary the following variables:

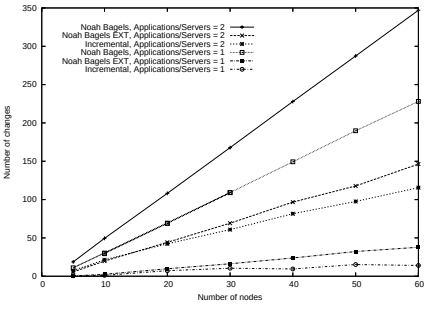
1. The number of servers. We start from 5 servers and increase their number to 60. This range of values covers the server-cluster sizes that are used in practice.
2. The number of applications that may be hosted on each server, which corresponds to their memory sizes. We set this value to 3 in all experiments, i.e. $\Gamma_i = 3$, $i = 1, \ldots, m$.
3. The ratio of the number of applications to the number of servers. We use values of 1, 1.5, and 2.
4. Memory requirements of applications. For this section, memory requirements are the same for all applications, i.e., $\gamma_j = 1$, $j = 1, \ldots, n$. In the next section we will consider the case in which different applications have different memory requirements.
5. CPU speeds of servers. In our study, the CPU speeds of all servers are the same.
6. Utilization factor, which is the ratio of the sum of CPU processing requirements of all applications to the sum of the CPU speeds of all servers. We use utilization factors of 0.8, 0.9, and 0.99.

Given the above parameters, we generate the test scenarios iteratively as follows. We set the initial placement to be empty. We repeatedly produce CPU demand for applications by independently generating a set of $n$ random numbers in the interval $[0, 1]$, $p_1, \ldots, p_n$, one for each application. Then we normalize these values by setting $p_j^* = \alpha p_j$, where $\alpha = \frac{1}{\sum_j p_j}$. Then we set the load-dependent demand of application $j$, $\omega_j = p_j^* \rho \sum_i \Omega_i$, where $\rho$ is the utilization factor. Given the new demand, we compute the new placement, taking the placement from the previous iteration as the current placement.

It should be mentioned that, in practice, demand in a given cycle is correlated with the demand in the previous cycle, whereas in our experiments the new demand is independent of the old demand. However, this correlation is very




(a) Utilization factor of 0.8




(b) Utilization factor of 0.99

**Fig. 1.** The number of placement changes

difficult to quantify. Our random method provides a pessimistic case on which to test the algorithms.

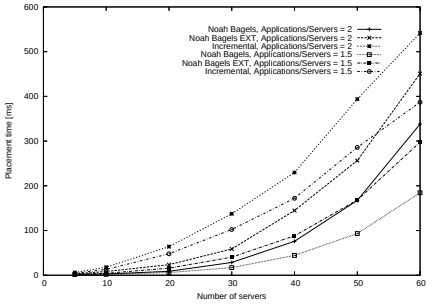
Each data point presented in the following sections was obtained as an average of 100 consecutive placement recomputations.

**Number of Placement Changes.** In this section we compare the three algorithms with respect to the number of changes to the previous placement required to satisfy the new demand. Figures 1(a) and 1(b) show this difference for various utilization factors and application-to-server ratios.

We can conclude that our incremental algorithm causes less perturbation of the placement than the previous algorithms. For reasonable utilization factors (e.g., 0.8 in Figure 1(a)), this difference is very significant.

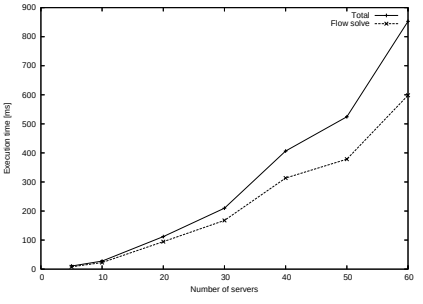
**Running Time.** In Figures 2(a) and 2(b), we show the execution time of a single placement recomputation. We observe that the incremental algorithm runs longer than the other two algorithms. In problem instances with a large number of applications and a high utilization factor, the running time performance of our algorithm is significantly worse than that of other algorithms. This is caused by the fact that in those hard problem instances, a placement satisfying the new demand might be obtained only by removing all or most of the current application instances, which requires the algorithm to execute a large number of iterations. However, even in these hard cases, the incremental algorithm allows the new placement for a 60-server cluster to be computed in 20 seconds. Note that starting a single application instance in our environment takes about 1 minute. A placement computation time that is a fraction of the application start-up time is acceptable in our application domain. Nevertheless, we investigate the computational time of the incremental algorithm further to understand the reasons for this lower performance.

Figure 3(a) presents the results obtained for utilization factor 0.9 and application to server ratio 2. It shows the total placement recomputation time and




(a) Utilization factor of 0.8 (b) Utilization factor of 0.99

**Fig. 2.** Execution time




(a) Total execution time (b) Number of invocations

**Fig. 3.** Contribution of flow-solve computation

the amount of time spent in flow-solve computation. Figure 3(b) presents the number of iterations and the number of flow-solve invocations per placement recomputation. We observe that solving the flow problem is the most significant contributor to the overall execution time. Even though the number of times the initial placement has to be invoked to compute the new placement is high in this scenario, compared to the number of times the flow problem has to be solved, the contribution of the actual placement computation remains low compared to that of the flow solver. It should be noted at this point that our flow-solve method uses the Simplex algorithm. Clearly, much more efficient techniques are available. We believe that, should one of the more efficient techniques be applied, the overall algorithm execution time would be significantly reduced.

### 6.2 Nonuniform Memory Requirements

Since in the case of non-uniform memory requirements $\gamma_j$ there are no other heuristics to solve our problem, we need to compare the value of the solution

provided by our algorithm either with the optimal value or with some lower bound on the optimal value. Finding a good lower bound seems a non-trivial task and remains an interesting open problem. The difficulty is that our objective function is essentially non-linear (concave) and it is hard (or impossible) to write a linear programming relaxation which does not have a large integrality gap. Finding good enumerative algorithms for our problem is also an interesting open question and again the main difficulty is to figure out how to construct a lower bound.

We compared the quality of solutions obtained by our algorithm with an optimal solution on instances of small size (5 servers and 14 applications). The optimal solution is found by complete enumeration with some additional tricks to cut down the search space.

The test instances are defined as follows: $m = 5$, $n = 14$. Server memory and capacity requirements are normalized and are equal to 1. Application memory requirements are chosen uniformly independently at random in the interval $[0, 2/3]$ and numbers of requests for each application are chosen uniformly independently at random in the interval $[0, 2\rho n/m]$, where $\rho$ is the density which was 0.7, 0.8 or 0.9 in these experiments.

In the case of small density $\rho = 0.7$ the heuristic found an optimal solution in almost all the cases and sometimes it used one more assignment. Note that if the optimal value is zero our heuristic always finds an optimal solution.

In the case of higher density $\rho = 0.8$ the behavior of our heuristic becomes worse with $1-2$ additional changes of assignment, and occasionally our heuristic wipes out the previous solution completely, making $11-15$ changes. In the case of $\rho = 0.9$ the behavior is even worse. The difference between optimal value (which is $1-2$) and approximate value is significant $(5-6)$ in approximately 30% cases.

We believe that such behavior is due to a combination of bad factors such as small instance size, non-uniform capacities that sometimes do not allow us to utilize servers' memory well, and high density which decreases the number of "good" solutions.

## 7 Conclusion

In this paper we defined a new optimization problem and presented a heuristic algorithm to solve it. In this problem we produce copies of applications (at some cost) and assign them to servers. This is not a well-studied concept in classical scheduling where jobs or applications are usually simply assigned to machines. (A related concept is "duplication" that is popular in scheduling with communication delays [3, 4].) Another interesting feature of our problem is the objective function, which is the number of placement changes from time step to the next time step. This type of objective functions was not studied before in the optimization literature, to the best of our knowledge.

We think that it would be interesting to provide some theoretical evidence that our (or any other) algorithm performs well on a suitably defined class of instances for which finding feasible solutions is easy.

Another open problem, mentioned in the previous section, is to develop efficiently computable lower bounds which would allow more extensive and rigorous heuristic testing.

## References

1. J. Kangasharju, J. Roberts, and K. W. Ross, Object replication strategies in content distribution networks, In 6th Int'l Workshop on Web Content Caching and Distribution (WCW), Boston, MA, 2001.
2. H. Kellerer, U. Pferschy and D. Pisinger, Knapsack Problems, Springer, 2004.
3. A. Munier and C. Hanen, Using duplication for scheduling unitary tasks on $m$ processors with unit communication delays, Theoret. Comput. Sci. 178 (1997), 119–127.
4. C. Papadimitriou and M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms. SIAM J. Comput. 19 (1990), 322–328.
5. D. N. Serpanos, L. Georgiadis, and T. Bouloutas, MMPacking: A load and storage balancing algorithm for distributed multimedia servers, IEEE Transactions on Circuits and Systems for Video Technology, 8(1):13 17, February 1998.
6. H. Shachnai and T. Tamir, On two class-constrained versions of the multiple knapsack problem, Algorithmica 29 (2001), 442–467.
7. H. Shachnai and T. Tamir, Noah Bagels - Some Combinatorial Aspects, International Conference on FUN with Algorithms (FUN), Isola d'Elba, June, 1998.
8. A. Turgeon, Q. Snell and M. Clement, Application placement using performance surfaces, in Proceedings of the Ninth International Symposium on High-Performance Distributed Computing, August 2000, Pittsburgh, PA, 229 - 236.
9. J. L. Wolf, P. S. Yu, and H. Shachinai, Disk load balancing for video-on-demand systems, ACM/Springer Multimedia Systems Journal, 5(6):358–370, 1997.
10. http://www.ibm.com/software/webservers/appserv/extend/

# Integrating Coordinated Checkpointing and Recovery Mechanisms into DSM Synchronization Barriers[1]

Azzedine Boukerche[1], Jeferson Koch[2], and Alba Cristina Magalhaes Alves de Melo[2]

[1] SITE – School of Information Technology and Engineering,
University of Ottawa, Canada
boukerch@site.uottawa.ca
[2] Department of Computer Science,
Campus Universitario - Asa Norte, Caixa Postal 4466,
University of Brasilia, Brasilia – DF,
CEP 70910-900, Brazil
{jeferson,albamm}@cic.unb.br

**Abstract.** Distributed Shared Memory (DSM) creates an abstraction of a physical shared memory that parallel programmers can access. Most recent software DSMs provide relaxed memory models that guarantee consistency only at synchronization operations. As the main goal of DSM systems is to provide support for long term computation intensive applications, checkpointing and recovery mechanisms are highly desirable. This article presents and evaluates the integration of a coordinated checkpointing mechanism to the barrier primitive that is usually provided with many DSM systems. Our results on some popular benchmarks and a real parallel application show that the overhead introduced during the failure-free execution is often small.

## 1 Introduction

In order to make shared memory programming possible in distributed architectures, we must create a shared memory abstraction that parallel processes can access. This abstraction is called Distributed Shared Memory (DSM). The first DSM systems tried to give parallel programmers the same guarantees they had when programming uniprocessors. It has been observed that providing such a strong memory consistency model creates a huge coherence overhead, slowing down the parallel application and bringing frequently the system into a thrashing state [15]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared memory behaviors that are different from the traditional uniprocessor one.

In the shared memory programming paradigm, synchronization operations must be used every time processes want to restrict the order in which memory operations should be performed. Using this fact, hybrid Memory Consistency Models guarantee that processors only have a consistent view of the shared memory at synchronization

[1] This work was partially supported by NSERC, Canada Foundation for Innovation and Canada Research Chair Programs.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 403 –414, 2005.
© Springer-Verlag Berlin Heidelberg 2005

time. This allows a great overlapping of basic read and write memory accesses that can potentially lead to considerable performance gains. By now, the most popular hybrid memory consistency models for DSM systems are Release Consistency (RC) [4] and Scope Consistency (ScC)[7].

While the memory consistency model defines when consistency should be ensured, coherence protocols define how it should be done. Most recent DSM systems use sophisticated coherence protocols that are often variations of the Lazy Release Consistency (LRC) protocol [9]. LRC is a multiple-writer, homeless protocol that uses techniques such as page twinning, page diffing and write notices to guarantee that pages at the shared memory segment are consistent at acquire time. In homeless protocols, fetching the up to date version of a page usually involves sending messages to many nodes whereas, in home-based protocols, it is sufficient to fetch the up to date version of a page from a single node, called home.

For any system running on a distributed platform that aims to be used in large scale, fault tolerance mechanisms must be considered. Usually, fault tolerance is achieved by periodically checkpointing each process that compose the system and, in the case of a failure, recovering the system to a previous consistent system state by activating the saved checkpoints.

Checkpointing can be done in a coordinated or in an uncoordinated way. Coordinated checkpointing is achieved by establishing a checkpointing session that captures a global consistent state of the execution and saves it to a stable storage. Usually, all DSM processes stop computing to take their checkpoints, in a coordinated way. Rollback/recovery is quite simple and is done by activating the last set of checkpoints.

In uncoordinated (or independent) checkpointing, there is no need to establish chekpointing sessions and all processes can take their checkpoints whenever they want. However, rollback/recovery in this case is unbounded and garbage collection is complex [3]. To overcome this problem, message logging is often associated with uncoordinated checkpointing.

Barrier synchronization is used in DSM systems whenever a global synchronization point needs to be established. Thus, taking checkpoints at barriers is a natural choice to implement coordinated checkpointing in DSM systems.

In this article, we propose and evaluate a coordinated checkpointing/recovery strategy that can be adapted to DSM systems that have barrier primitives. We also opted to take the actual checkpoints with an existing checkpoint library (ckpt [21]) in order to make our approach more portable and to concentrate efforts on the coordinated checkpointing strategy and at the recovery mechanism itself. This separation between the checkpointing mechanism and the checkpointing policy is the original part of our approach.

The remainder of this paper is organized as follows. Section 2 describes some characteristics of memory coherence protocols for DSM systems. Section 3 presents an overview of checkpointing schemes for DSM systems. The JIAJIA software DSM is described in section 4. Section 5 describes our checkpointing and recovery mechanisms. Some experimental results are discussed in Section 6. Finally, Section 7 concludes the paper and presents future work.

## 2 Memory Coherence Protocols for DSM Systems

There are two basic coherence protocols which are traditionally used to solve the cache coherence problem: write-invalidate and write-update protocols. In general, at consistency time, a write-invalidate protocol guarantees that there is only one up-to-date copy of the data in the system. If these data are further needed, they are fetched from the only node that has a valid copy. On the other hand, a write-update protocol allows many copies of the same data to be valid at consistency time and generally guarantees that updates will be done atomically and totally ordered. Although most of DSM systems use write-invalidate based protocols (TreadMarks[9], ATMK[1]), some recent DSM systems offer also write-update protocols (ADSM[14], Brazos[18]).

Another characteristic that is important in a cache coherence protocol for DSM systems is the number of simultaneous writers allowed. The most intuitive approach allows only one processor to write data at a given moment (single-writer protocol). However, as the unity of consistency is often a page, false sharing can occur when two or more processors want to access independent variables that belong to the same page. To reduce the effects of false sharing, many DSM systems use multiple-writer protocols, allowing many processors to have write access to the same page simultaneously and then merging the multiple versions of the page when a consistent view is required. Multiple-writers protocols are generally implemented for LRC and Scope Consistent DSM systems as follows. If a write fault occurs inside a critical section, the original page is copied to a twin before write access is granted. When the lock is released, the pages are compared to their twins and the differences between them (diffs) are generated. At acquire time, the lock manager sends to the acquiring process an acquire message containing the identifications of the pages that are no longer valid (write notices). These pages are invalidated before the application continues its execution.

There are two basic approaches used to manage the information needed to execute coherence protocols in page-based DSM systems: home-based and homeless. In home-based systems, each page is assigned to a node (home-node) that concentrates all modifications made to the page. Every time an up-to-date version of the page is needed, it is sufficient to contact the home node in order to fetch the page. In the homeless approach, each processor that modifies a page maintains such modifications locally. In order to obtain an up-to-date version of the page, a node must collect the modifications that are distributed all over the system. Modifications are kept by each node and garbage collection is required.

Home-based protocols do have some advantages. First, each access fault requires only communication with the home-node. Second, since modifications are eagerly applied at the home-node, there is no need to keep additional control structures such as twin pages or diffs after the home node is updated. However, as modifications are eagerly sent to the home node, such protocols generally require additional messages. Also, on an access fault, homeless protocols fetch only the modifications made to the page (diffs) while home-based protocols fetch the whole page [5].

## 3 Checkpointing Mechanisms for DSM Systems

In DSM systems, research in rollback/recovery is concentrated in adapting either coordinated or uncoordinated checkpointing strategies from message-passing systems to the DSM environment.

In [8], a coordinated checkpointing mechanism is proposed that keeps track of the DSM processes interactions by building a communication tree in order to reduce the number of processes involved in a checkpointing session. Incremental checkpoints are taken only for this subset of processes and stored to disk.

A coordinated checkpointing strategy for TreadMarks [9] is presented in [10]. Coherence-related information contained in write-notices are used to decide which data really need to be saved in the checkpoint.

An uncoordinated checkpointing strategy for home-based release consistent DSM systems is presented in [19]. Logs of DSM operations are maintained in remote memories for further use in the recovery mechanism.

A logging strategy for ADSM[14] is proposed in [11]. Coherence-related information is logged according to the coherence protocol which is in use. The protocols considered are multiple-writer/write-invalidate and single-writer/write-update. Coordinateed and uncoordinated checkpointing can be used. The use of coordinated checkpointing with logging can be justified since, in this case, the size of the logs can be reduced.

In [23], the impact of locks and barriers in sequential consistent DSM systems is analyzed. Kernel-level support is used to keep track of data dependencies in either coordinated and uncoordinated checkpointing.

In all these systems, coherence related information is used to decide which information will be included in the checkpoint. Also, most of these works deployed their own checkpoint libraries. Modifications made to the operating system kernel must be often taken into account by the checkpoint library. Modifications into the kernel itself are made in [23].

Like [8] and [10], we opted to use coordinated checkpointing. However, unlike them, we integrated an existing checkpointing library to our checkpointing mechanism.

Also, we did not use coherence-related information to reduce the size of the checkpoint file since our approach is designed to work in any DSM system that provides barriers as synchronization mechanisms.

## 4 The JIAJIA Software DSM System

JIAJIA is a software DSM system proposed by Hu, Shi and Tang [5]. It implements the Scope Consistency memory model with a write-invalidate multiple-writer home-based protocol.

In JIAJIA, the shared memory is distributed among the nodes in a NUMA-architecture basis. Each shared page has a home node. A page is always present in its home node and it is also copied to remote nodes on an access fault. There is a fixed number of remote pages that can be placed at the memory of a remote node. When this part of memory is full, a replacement algorithm is executed.

Scope Consistency is a memory model that requires that consistency must be guaranteed when a process acquires a lock or when it reaches a synchronization barrier. In the first case, consistency is maintained in a per-lock basis, i.e., only the shared variables that were modified on the critical section guarded by lock l are guaranteed to be updated when a process acquires lock l.

On a synchronization barrier, however, consistency is globally maintained and all processes are guaranteed to see all past modifications to the shared data [6].

In order to implement Scope Consistency, JIAJIA statically assigns each lock to a lock manager. The functions that implement lock acquire, lock release and synchronization barrier in JIAJIA are jia_lock, jia_unlock and jia_barrier, respectively [17].

On a release access, the releaser sends all modifications performed inside the critical section to the home node of each modified page. The home node applies all modifications to its own copy and sends an acknowledgment back to the releaser. When all acknowledgements arrive, the releaser sends a message containing the numbers of the pages modified inside the critical section (write notices) to the lock manager [6].

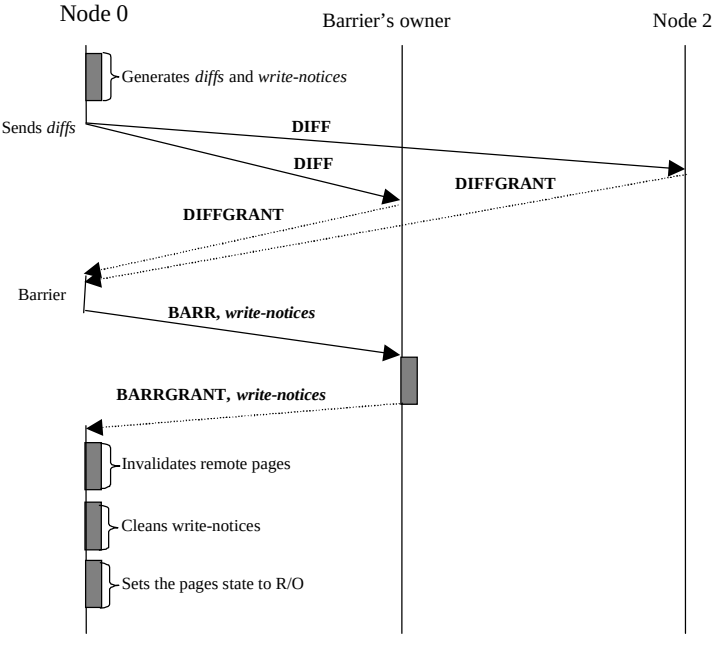



**Fig. 1.** Barrier synchronization at the JIAJIA software DSM system

On an acquire access, the acquirer sends an ACQ message to the lock manager. When the lock manager decides that the lock can be granted to the acquirer, it responds with a lock granting message that contains all write notices associated with that lock. Upon receiving this message, the acquirer invalidates all pages that have write notices associated, since their contents are no longer valid.

On a barrier access, the arriving process generates the diffs of all pages that were modified since the last barrier access. Then, it sends the diffs to the respective home nodes. The home nodes receive the diffs, apply the modifications, and send an acknowledgement to the arriving process. Then, the arriving process sends a BARR message containing the write notices of all modified pages to the owner of the barrier.

When all processes arrive at the barrier, the owner of the barrier sends back a BARRGRANT message to each process, containing the write notices of all pages modified since the last barrier. Upon receiving this message, the processor invalidates the pages contained in the write notices and continues the execution.

Figure 1 illustrates the barrier operation in JIAJIA. For simplicity, the whole process was only represented for node 0.

## 5 Proposed Coordinated Checkpointing/Recovery Mechanisms

### 5.1 Design Choices

Some assumptions were made when designing our mechanisms. First, the communication network and the stable storage used to store the checkpoints is assumed to be fault-free. Second, only transient faults will be treated. Permanent faults are not supported. Third, processes fail in a fail-stop mode. More complex failures such as byzantine failures are not considered. Forth, failures that occur during the recovery process are not supported.

The first decision made in the design of our checkpoint/recovery mechanism is whether to use coordinated or uncoordinated checkpointing. Although coordinated checkpointing can introduce non-negligible overheads in failure-free executions, the overheads that can be introduced by uncoordinated strategies due to complex garbage collection schemes and log management are often very high. Besides, uncoordinated checkpointing makes the recovery process more complex. Thus, we opted to design a coordinated checkpointing scheme and, to overcome the possible problem of considerable overheads introduced to failure-free executions, special care was taken not to increase the number of messages exchanged by the nodes due to checkpointing.

The second design choice concerned the checkpointing mechanism itself. We opted to use an existing checkpointing library since we wanted to concentrate our efforts on the coordinated checkpointing strategy. Also, we wanted to decouple the checkpoint strategy from the checkpointing procedure in order to make it easier to use different libraries for different operating systems, while maintaining the same checkpointing strategy.

In order to choose an appropriate checkpointing library, we evaluated *Libckpt* [16], *Libckp* [20] and *ckpt* [21]. The following characteristics were analyzed. First, solutions that required modifications in the operating system kernel were discarded, since these modifications introduce portability problems. Libraries that ran always in user-level with no kernel modifications were preferred. Second, open source libraries were preferred. Third and more important, the recovery mechanism provided by the library should work for general Unix processes and, specially, for Unix processes using JIAJIA primitives.

Of those, only *ckpt* presented these three characteristics that is the reason why it was chosen. However, *ckpt* does have some limitations [21]. First, only integral checkpointing is done, i. e., incremental checkpointing is not implemented. Second, checkpoint files are always written to disk and that precludes the use of remote memories as stable storage. Third, the following resources are not saved to the checkpoint file [21]: open files, network connections, Unix inter-process communication (IPC) mechanisms, thread information and the process status.

### 5.2 Design of the Checkpointing Mechanism

Our coordinated checkpointing mechanism is integrated to the barrier synchronization primitive and requires no additional inter-node communication to take checkpoints. In DSM systems that implement relaxed memory models such as LRC and ScC, the barrier is the only execution point that captures a consistent global state of the execution. Thus, it is the natural choice for the integration of a coordinated chekpointing mechanism. In this case, there is no need for the checkpointng mechanism to establish a consistent global state since it is done naturally by the barrier primitive.

The primitive *jia_barrier* illustrated in figure 1 was modified to include our checkpointing strategy. When a process receives a BARRGRANT message, it knows that all the other processes have already reached the barrier and that the global consistent state is attained. At this moment, the following actions are taken:

a) the process disables interruptions to guarantee that no messages will be treated while the checkpointing is underway;
b) integral checkpointing is made to the local disk using the command *checkpointHere*, provided by the *ckpt* library. If there is already a checkpoint for this process on the disk, it is replaced by this last one; and
c) interruptions are enabled.

After that, the *jia_barrier* proceeds as in the original implementation (figure 1): remote pages are invalidated, write-notices are cleaned and the pages state is set to read/only.

The programmer has the choice to enable/disable the checkpointing mechanism whenever he/she wants. The primitives *jia_config(CKPTREC,ON)* and *jia_config(CKPTREC,OFF)* are provided and they activate/deactivate the checkpointing mechanism at the subsequent barrier primitives. The primitive *jia_barrier_ckpt* is also provided to force chekpointing without considering the value of the variable CKPTREC.

### 5.3 Design of the Rollback/Recovery Mechanism

The rollback/recovery mechanism allows the execution to be continued from the last saved checkpoint. The whole rollback/recovery mechanism is illustrated in figure 2, where P0 is the node from where the DSM application was initially launched.

The program *jia_restart <application>* is offered to provide this facility. When the command *jia_restart* is executed at processor P0, the following executable files are copied to the remote machines: *jia_restart*, *<application>* and *restart*. The library *librestart.so* and some configuration files are also copied at this point. After that, the remote execution command *rexec "jia_restart <application>"* is issued to all nodes that were running the DSM application when the failure occurred.

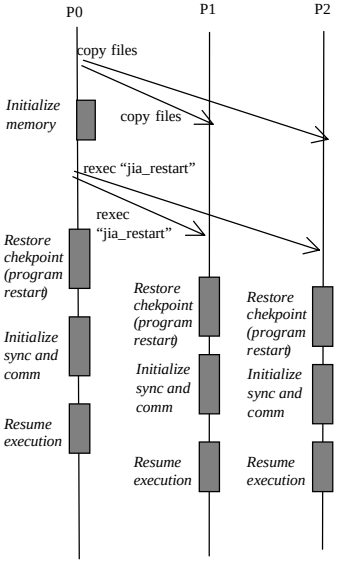



**Fig. 2.** Proposed rollback/recovery protocol

When this command starts execution on the remote nodes, it first executes the procedure *initmem* that does the mapping of the DSM area to the memory assigned to the process. After that, the checkpoint file is read from the local disk and the process state is built from this file by the *ckpt* library (program *restart*). Since the *ckpt* library did not save some data that are needed for a JIAJIA process to restart correctly, additional procedures (*initsyn* and *initcomm*) were included at this point.

The procedure *initsyn* is needed to allocate memory for the data structures that represent the locks and barriers. Finally, the procedure *initcomm* allocates and reactivates the sockets that were used by the process, which were not saved by *ckpt*, as explained in section 5.1. Since the restored JIAJIA processes resume execution from the last saved barrier, there is no need for synchronization at the end of this protocol.

## 6 Experimental Results

The mechanisms described in section 5 were implemented in C and integrated to JIAJIA.

To evaluate our mechanism, we ran our experiments on a dedicated cluster of 8 Pentium II 550 MHz, with 160 MB RAM connected by a 100Mbps Ethernet switch. The JIAJIA software DSM system v.2.1 ran on top of Debian Linux 2.2.5.

Our results were obtained with 4 popular parallel benchmarks and a real parallel application. The following benchmarks were used: EP from the NAS parallel benchmark [2], TSP from TreadMarks benchmarks [12], MM, which is a matrix multiplication program that uses the inner product algorithm and MMA, which is also an inner-product matrix multiplication but each value is calculated with additions. A real parallel application that locally aligns long DNA sequences (Genome) was also evaluated. This application uses a variant of the algorithm proposed by [22] and has time complexity $O(n^2)$ where $n$ is the size of the sequences. More details about this application can be found in [13]. The sizes of the problem and the synchronization primitives used by these applications are shown in table 1.

**Table 1.** Characteristics of the applications

| Program | Problem size | Synchronization |
|---|---|---|
| EP | $2^{28}$ | Locks and Barriers |
| TSP | 19 | Locks and Barriers |
| MM1408 | 1408x1408 | Barriers |
| MMA1408 | 1408x1408 | Barriers |
| Genome398 | 398x398 | Locks and Barriers |
| Genome782 | 782x782 | Locks and Barriers |

The execution times and speedups of these applications are shown in figure 3. It must be noted that the Genome application parameters were set in order to reproduce a highly communication-intensive scenario with bad speedups.

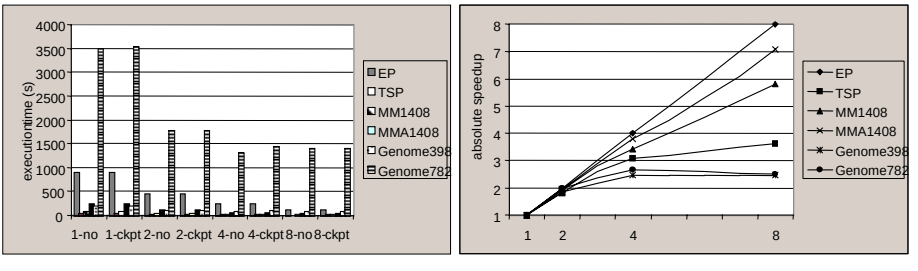



**Fig. 3.** Execution times (sec) and speedups for the applications

Figure 4 shows the average size of the checkpoint file for these applications.

For the *ckpt* library there seems to be a lower bound on the checkpoint file size that relies around 7MB (figure 4). For this reason, the sizes of the checkpoint files for EP and TSP do not decrease as we increase the number of nodes. For MM and MMA, the size of the checkpoint file for the one-node execution is 31MB. As long as we

increase the number of nodes, the part of the matrix to be calculated is smaller and so are the checkpoint files.

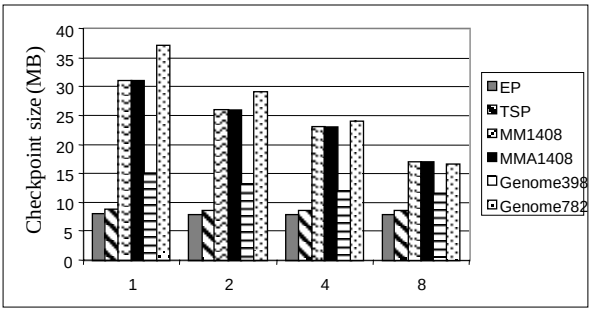



**Fig. 4.** Average checkpoint sizes for the applications

It must be noted that the size of the checkpoint file at node 0 is bigger than in the other nodes. For instance, with 8 processors, the size of the checkpoint file for MM1408 is 23.1MB and 14.3MB, at node 0 and at the other nodes, respectively. This can be explained since node 0 is considered the master node and contains data structures that are exclusive to it.

Figure 5 shows the overhead introduced by our checkpointing strategy in failure-free executions. The highest overhead obtained was 19.9%, achieved when executing MM with 8 processors. For the four benchmarks analyzed, the overhead introduced by our mechanism increases as long as the number of nodes is increased. This can be explained since, with more nodes, the execution time is smaller but the size of the checkpointing file does not decrease in the same rate (figure 4) and 7MB seems to be a lower bound for the checkpoint file size. Thus, the impact of the overhead on the total execution time increases when more nodes are added.

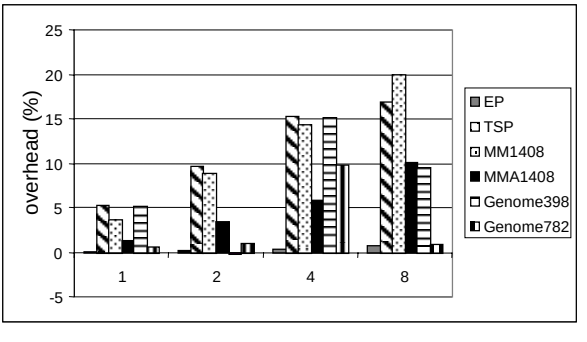



**Fig. 5.** Failure-free overheads for the 6 applications

The genome applications (genome398 and genome782) were of particular interest since they presented really bad speedups (figure 3) and this could lead to a negative

impact in our checkpointing mechanism. For these two applications, the highest overheads (9.7% and 15.1%, respectively) were obtained with 4 nodes. With 8 nodes, the failure-free overhead presented by these applications dropped to 0.8% and 9.4%, respectively.

Nevertheless, even in this bad scenario, the overhead values obtained by our checkpointing mechanism with the genome applications were close to the ones obtained with three benchmarks (TSP, MM and MMA).

To test our recovery mechanism, we turned the power off while the genome782 application was running, waited for 5 minutes and then turned the power on. After rebooting, all the nodes restarted execution from the last saved checkpoint and the results produced were correct. The whole recovery process took less than 2 minutes.

## 7 Conclusions and Future Work

This article presented the design and evaluation of checkpointing and rollback/recovery mechanisms that can be adapted to DSM systems that use barriers as synchronization mechanisms. The main characteristic of our checkpointing mechanism is that it does coordinated checkpointing at barrier time, without adding messages to the DSM system. We chose to use an existing checkpointing library (ckpt) in order to separate the procedure from the mechanisms and to focus on the checkpointing and recovery mechanisms themselves.

The experimental results obtained in an eight machine cluster with 4 popular benchmarks and a bioinformatics application presented reasonable failure-free overheads, ranging from 0.6% to 19.9% for 8-node executions. The size of the checkpoint file is also reasonable, ranging from 7MB to 17MB with 8 processors.

As future work, we will use our mechanism in long run real parallel applications. Also, we are investigating how to integrate an incremental checkpointing strategy to our mechanism.

## References

1. Amza C., Cox A., Dwarkakas S., Zwaenenpoel W.: Software DSM Protocols that Adapt between Single Writer and Multiple Writer, Proc. of HPCA'97, p. 261-271, 1997.
2. Bailey D. et al.: The NAS Parallel Benchmarks, TR 103863-NASA, July, 1993.
3. Elnozahy M., Alvisi L. Wang L.: A Survey of Rollback/recovery Protocols in Message-Passing Systems, TR CMU-CS-96-181, 1996.
4. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proc.ISCA*, May, 1990, p15-24.
5. Hu W., Shi W., Tang Z.: JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In Proc. of HPCN'99, pp. 463-472.
6. Iftode L.: Home-Based Shared Virtual Memory, PhD Thesis, 1998, Princeton University.
7. Iftode L. et al..: Scope Consistency: Bridging the Gap Between Release Consistency and Entry Consistency, Proc.ACM SPAA'96, p 277-287.
8. Janakiraman, G., Tamir, Y., Coordinated Checkpointing-Rollback Error Recovery for DSM Multicomputers, proc of 13th Symposium on Reliable Distributed Systems, 1994.

9. Keleher P. et al..: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proc.USENIX, 1994, p. 115-132.
10. Kongmunvattana A., Tanchatchawal S., Tzeng N.: Coherence-Based Coordinated Checkpointing for Software Distributed Shared Memory Systems, Proc. ICDCS, April, 2000, p. 556-563.
11. Kongmunvattana A., Tzeng N, Logging and Recovery in Adaptive Software Distributed Shared memory Systems, Proc. of the 18th Symp. on Reliable Distributed Systems, 1999.
12. 12.H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Quantifying the performance differences between pvm and Treadmarks," Journal of Parallel and Distributed Computation, Vol. 43, pp. 65-- 78, Jun. 1997.
13. R. Melo et al. "Comparing Two Long DNA Sequences Using a DSM System", Proceedings of Euro-Par 2003, p. 517-524, Klagenfurt, 2003.
14. Monnerat L. , Bianchinni R.: Efficiently Adapting to Sharing Patterns in Software DSMs, Proc. HPCA'98, February, 1998.
15. Mosberger D.: Memory Consistency Models, Operating Systems Review, p. 18-26, 1993.
16. Plank, J. S., Beck, M, Kingsley, G. and Li, K., Libckpt: transparent Checkpointing under Linux, USENIX Winter 1995 Technical Conference, January, 1995.
17. Shi W.: Improving the Performance of DSM Systems, PhD Thesis, CAS, November, 1999.
18. Speight E., Bennett J, Reducing Coherence-Related Communication in Software Distributed Shared Memory Systems, TR ECE-TR-98-03, Rice University,1998.
19. Sultan, F. Nguyen, T., Iftode, L., Scalable Fault Tolerant Distributed Shared Memory, proc of Int. Conf. On High Performance Networking and Computing, 2000.
20. Wang, Y. Chung, P. and Fuchs, W., Tight Upper Bound on Useful Distributed Systems Checkpoints, Technical Report CRHC-95-16, University of Urbana-Champaign, USA, 1995.
21. Zandy, V., CKPT: A Checkpoint Library under Unix, http://www.cs.wisc.edu/~zandy/ckpt.
22. Smith, T. F, Waterman, M. S., Identification of common molecular sub-sequences, Journal of Molecular Biology, 147 (1) 195-197 – 1981.
23. Badrinath, R. and Morin, C., Locks and Barriers in Checkpointing and Recovery, Proceedings of the IEEE/ACM CCGrid 2004, Chicago, USA, April, 2004.

# Synchronization Fault Cryptanalysis for Breaking A5/1*

Marcin Gomułkiewicz[1], Mirosław Kutyłowski[1], Heinrich Theodor Vierhaus[2], and Paweł Wlaź[3]

[1] Wrocław University of Technology
{gomulkie, mirekk}@im.pwr.wroc.pl
[2] Brandenburg University of Technology, Cottbus
htv@informatik.tu-cottbus.de
[3] Lublin University of Technology
p.wlaz@pollub.pl

**Abstract.** A5/1 pseudo-random bit generator, known from GSM networks, potentially might be used for different purposes, such as secret hiding during cryptographic hardware testing, stream encryption in piconets and others. The main advantages of A5/1 are low cost and a fixed output ratio.

We show that a hardware implementation of A5/1 and similar constructions must be quite careful. It faces a danger of a new kind of attack, which significantly reduces possible keyspace, allowing full recovery of A5/1 internal registers' content. We use "fault analysis" strategy: we disturb the A5/1 encrypting device (namely, clocking of the LFSR registers) so it produces an incorrect keystream, and through error analysis we deduce the state of the internal registers. If a secret material is used to initialize the generator, like in GSM, this may enable recovering the secret. The attack is based on unique properties of the clocking scheme used by A5/1, which is the basic security component of this construction.

The computations that have to be performed in our attack are about 100 times faster than in the cases of the previous fault-less cryptanalysis methods.

**Keywords:** fault cryptanalysis, A5/1, GSM, LFSR.

## 1 Introduction

In this paper we consider the security of A5/1 algorithm. The algorithm is a pseudo-random bit generator used e.g. by GSM networks for keystream generation. It is extremely simple in design: it consists of three LFSR registers, which output is XOR-ed. The most important feature of this algorithm is its' LFSRs' clocking mechanism.

The cryptographic strength of A5/1 algorithm comes from a non-linear clocking rule. All LFSR-based designs, which do not have a non-linear component, can be broken easily with simple linear algebra. A lot of attention was devoted to the question how

* Partially supported by DAAD project PROKRYPTO and KBN project no. 4 T11D 005 24.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 415–427, 2005.
© Springer-Verlag Berlin Heidelberg 2005

to combine LFSRs – which are easy to build and fast – in order to get a cryptographically strong generator. A5/1 is an extremely simple solution to this question, another one is a shrinking generator [8]. A5/1 has an advantage over the shrinking generator, it produces an output bit at each step, while the shrinking generator requires a buffer for collecting the bits before outputting them.

**Potential Applications.** A5/1 is at the moment probably the most widely used stream cipher. It has been designed so that its' cheap and efficient hardware implementation is possible, so the idea to use A5/1 for purposes other that those it was originally created for might be tempting. This is additionally motivated by the fact that using existing and tested ciphers instead of designing new ones is advised from a security's viewpoint. Such tasks could include for example purposes related to secure testing of integrated circuits that store sensitive data. Every chip that is currently produced contains an appropriate test circuitry, used generally right after production and before final packaging. However, sometimes integrated circuits have to be checked thoroughly and frequently – this occurs for many crucial security components implemented in hardware, a good example are secure signature creation devices for which a hardware fault must be detected as soon as possible. If such a chip contains secret data, possibility to perform checks by an unauthorized personnel poses a security threat, especially in presence of some modern side-channel cryptanalysis techniques, such as timing attacks, DPA or fault analysis. To avoid problems, the test output should be scrambled. Due to technical reasons, algorithms such as A5/1 are very well suited for becoming scramblers – the data from the testing circuit comes through a kind of serial interface and can be combined with the random bits of the scrambler before sending them out.

**A5/1 Security.** Several attacks on encryption algorithms A5/1 and A5/2 (a weaker algorithm that may be used instead of A5/1 by GSM phones) have been proposed, see [4, 5, 10]; the most recent work [2] is based on some fundamental cryptographic flaws found in GSM protocol, and describes a way to recover A5/2 secret session key using only a couple of milliseconds of encrypted communication within a second on a home computer. Authors suggest that although similar method can be used against stronger A5/1, that attack would be rather cumbersome in practice: one of possible setups is that if one had 8 seconds of encrypted communication and 5000 PCs had completed preprocessing in 1 year's time filling up 176 200GB disks, another 1000 PCs could perform an attack in real-time. However, while such an effort is rather unrealistic for individuals, it is well within the range of a middle sized company or university, not to mention wealthy and highly motivated attackers.

**New Result.** The main goal of this paper is to inspect threats of A5/1 device security that come from the design of the shift registers' clocking. We show that disturbing a clocking sequence yields an alternative, quite efficient attack on A5/1. For instance, it allows to reconstruct the contents of A5/1 internal registers with a moderate computational effort in a certain fault model. This attack may be seen as a special case of attacks on A5/1 predicted in [4], however as far as we know it is the only attack that use fault-analysis strategy, and at the same time it is probably the most efficient

one. The attack is based on stopping a LFSR register from clocking at a given moment. Our paper shows that the countermeasures against hardware faults of this kind are necessity.

**Paper Organization.** We start with an overview of A5/1 stream cipher and briefly describe the idea of fault analysis. Then we introduce the idea of resynchronization, and describe the way it can be used to guess some part of A5/1 internal state, and thus mount an effective attack; we discuss some difficulties and their possible solutions. At the end we discuss technical feasibility of attacks on physical devices and necessary precautions.

## 2 Preliminaries

### 2.1 The A5/1 Stream Cipher

The A5/1 algorithm is known for several years: although never officially published, its construction has been found via reverse-engineering [7]. The algorithm, as used by GSM, has three linear shift registers R1, R2 and R3 with 19, 22 and 23 bit cells respectively. The session key has 64 bits. The data transferred is divided into so-called *frames*, each 228-bit long: 114 bits of incoming data and 114 bits of outgoing data. The frame number is 22 bit long. For each frame the keystream generation is divided into three phases. In the first phase, registers are zeroed and then clocked regularly 64 times, while their leftmost (see Fig. 1) bits are XOR-ed with consecutive bits of the session key. Similarly in another 22 similar steps 22 bits of the *frame number* are inserted (actually, the frame number is a fixed bit permutation of *TDMA frame number*, cycled in $2^{22}$ long periods.) In the second phase registers do not clock regularly – which of the registers clock depend on bits on positions 8 , 10, 10 in (respectively) R1, R2, R3, called *clocking windows*. The registers clock according to a majority rule: in the next step the only registers (2 or 3) that clock are the ones which "decision" bits from clocking windows (black cells on Fig. 1) are equal to the majority of these bits (majority of the three bits, $a$, $b$, $c$ can be expressed as $ab + ac + bc$.) The second phase lasts for 100 steps and its' output is discarded. Finally, during the third phase the registers work as in the second phase, but their output bits are XOR-ed; the results are the key bits that are XOR-ed with the plaintext. This phase lasts for 228 steps and then the frame is over. For the next frame, the whole procedure restarts with a new frame number.

### 2.2 Fault Cryptanalysis

Fault analysis is a fairly recent cryptanalysis tool; it was introduced by Boneh *et. al* [6]. The idea is to cause distortions during operation of a cryptographic device so that it produces a faulty output; comparing faulty and correct output can yield significant information about the secrets contained within the device. Among others, a spectacular attacks of this kind have been proposed against improperly implemented RSA scheme [13], and against symmetric algorithms [3, 9].

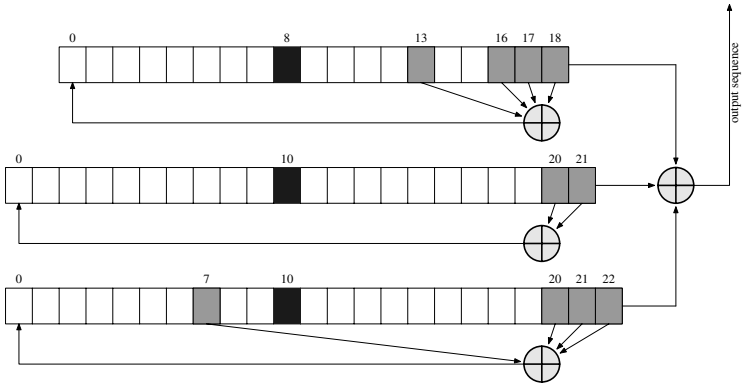



**Fig. 1.** A5/1 registers. Black bits are the majority bits

## 3 Resynchronization Attack

We consider an attack in which the adversary holds the encryption device with a secret key inside. The adversary's goal is to find the state of the linear shift registers at some moment of the computation. Having this state one can apply known computational methods [5] to trace back the states of the shift registers to the moment when the secret key is involved in the computation and derive the secret key.

### 3.1 Clocking Fault

We assume that an adversary, say Bob, holds an A5/1 encryption device and can use it freely. Since the input and output data are known for Bob, he gets the pseudorandom sequence generated by the device. Bob can insert faults into device's operation. For the sake of simplicity in this paper we discuss only one scenario: we assume that in an arbitrary moment Bob can block the shift that would be performed by one of the shift registers (say R1.) Hence, if R1 does not clock at this moment, this has no effect on the state of shift registers and therefore has no effect on the output. In the opposite case the computation gets disturbed: in the next step majority function is computed from different bits and changes in clocking may propagate. Typically, the shift registers clock in a different way and, consequently, different pseudorandom bits are generated.

Obviously, if after blocking R1 for one step no change in the output sequence occurs, then with high probability R1 does not clock at this moment in a correct computation. So by checking whether the output changes Bob gets some information on the bits that determine clocking. By performing many such trials one may hope to gather enough information to reconstruct the states of the shift registers. The problem is that putting these information together seems to be hard as the number of possible cases grows rapidly. Our goal is to find a method that allows making conclusions about the contents of the shift registers possibly less frequently, but when succeeded, only a small number of candidates for relatively large blocks of bits exist (strategy similar to the one presented in [4].)

### 3.2 Resynchronization

Assume that at step $t'$ clocking of a register was prohibited. We say that shift registers are *synchronized* at step $t$ of a computation, if during steps $t'$ through $t$ each of the shift registers clocks exactly the same number of times as in a fault free computation with the same initial state. Of course, clocking patterns need not to be the same, only the total number of moves for each shift register counts.

A key observation is that after a clocking fault **resynchronization** is possible and occurs in quite specific situations. For instance, consider the following configuration:

| | | |
|---|---|---|
| R1: | 1 0 0 0 0 ... 0 0 0 | $\leftarrow n-1$ zeroes |
| R2: | 0 0 0 0 0 ... 0 0 0 | $\leftarrow$ $n$ zeroes |
| R3: | 0 0 0 0 0 ... 0 0 0 | $\leftarrow$ $n$ zeroes |

Assume that a fault occurs at the first step in R1. It is easy to see that after $n$ steps the registers get resynchronized. Of course, the above example (due to the construction of R1) is valid only for $n < 10$.

Now let us introduce some notation. The block of bits starting in the clocking windows that provides resynchronization immediately after $k$ steps is called **resynchronization pattern** or $k$-**resynchronization pattern**, **RSP**$k$ for short; each resynchronization pattern consist of three blocks of bits, one per shift register (R1, R2 and R3.) For instance, the example above considered for $n = 2$ yields an RSP2 of the form $(10,00,00)$.

Any pattern can be checked offline for resynchronization property, so all resynchronization patterns of small length can be found via an exhaustive search. For instance, there are only two RSP3: $(011,111,111)$ and $(100,000,000)$, which in fact we have already seen. The things become more interesting for RSP4. There are the following resynchronization patterns of length 4:

$(011,1101,110)$, $(011,110,1101)$ , $(100,0010,001)$, $(100,001,0010)$
$(011,1101,1101)$, $(100,0010,0010)$,
$(0111,1111,1111)$, $(1000,0000,0000)$.

Essentially, we have here only three different cases, we can obtain each other pattern by replacing the contents of R2 and R3, or by flipping each bit.

The things are really exciting for larger lengths. Through an exhaustive search we have identified all resynchronization patters of lengths 5 through 9. The number of these patters equals: 30 for RSP5, 112 for RSP6, 480 for RSP7, 2068 for RSP8, and 8992 for RSP9. The number of the resynchronization patterns fits very well cryptanalysis needs: it is not too small (which would make finding resynchronization difficult) and not too big (large number of possible patterns would make hard to guess the pattern that has actually occurred.)

### 3.3 Outline of Cryptanalysis

The attack is based on the following observations:

– if we observe resynchronization after $n$ steps, then it is plausible to assume that one of patterns RSP$k$ occurred, for $k = n$ or $k$ slightly bigger than $n$,

– the clocking fault does not change the contents of the shift registers, only their timing is affected; as a consequence, before a resynchronization occurs, the output is built from the same bits, but the moments when particular bits are used by the output XOR-gate differ. The important thing is that these changes can be derived from resynchronization pattern.

The attack consists of the following phases:

*Phase 1:* run the device without faults and with faults injected at different moments; look for a situation where resynchronization occurs after some (about 5 to 9) steps and collect the corresponding output streams. For a given resynchronization length, there is a number of possible RSPs, typically couple thousands; we execute the remaining phases for each of these patterns.

*Phase 2:* compute clocking pattern that follows from the resynchronization pattern assumed – in this way derive clocking in both faulty and non-faulty execution for the period between the moment of injecting the fault and the moment of resynchronization. Given the clocking pattern, construct a system of linear equations describing the output in this period. For this purpose take a number of variables describing some number of rightmost bits in each of the shift registers. Then express the output bits in terms of these variables (each expression is a (mod 2) sum of three variables.) This is possible, since the clocking pattern is known. Note that we have two sets of equations – one corresponding to a faulty execution and one for the error-free one; moreover, the faulty computation gives us **different** equations with **the same** variables. For example, for RSP7 there are typically 13 equations of 18 variables. Solve this systems of linear equations. If the system has no solution then stop considering this resynchronization pattern. Otherwise, we can express a number of bits on the right side of shift registers at the moment when a fault occurs by but a few of these bits.

*Phase 3:* In this phase we have many bits of registers known (about 44 bits in the case of RSP9; about 21 in the case of RSP5.) To find the rest of them we will gradually guess the values of unknown bits needed for the clocking mechanism, make one move of the system and construct a linear equation with current rightmost bits of registers and the output bit. All equations are expressed in terms of bits of registers at the moment when clocking fault is caused.

Some of the guesses will contradict the system of linear equations, other will lead to full rank linear system with 64 unknowns – the solution should be then verified by comparing to the original keystream.

### 3.4 Some Details of the Attack

**Resynchronization Probability.** After injecting a clocking fault resynchronization occurs after 5 to 9 steps with probability of about 1.5% (0.0148462...). If we may afford to look for resynchronization occurring less frequently (meaning more experiments with the device) we should rather concentrate on longer patterns, say 8 or 9 steps long.

**Linear Equations for Phase 2.** First let us consider a toy example of an RSP3. Consider one of RSP3s mentioned before – $(011,111,111)$. Consider the state of registers before injecting a fault and denote the bits stored by R1, R2 and R3 by, respectively, $a_0,\ldots,a_{18}$, $b_0,\ldots,b_{21}$ and $c_0,\ldots,c_{22}$ (see Fig. 1.) Let the output observed in the fault free computation be $x_1,x_2,x_3,\ldots$ and in the faulty computation – $y_1,y_2,y_3,\ldots$. Since we assume resynchronization after 3 steps, $x_3 = y_3$. By inspecting the clocking sequence in both computations one can derive the following equalities for the output values:

$$\begin{cases} a_{17}+b_{20}+c_{21}=x_1 \\ a_{16}+b_{19}+c_{20}=x_2 \\ a_{16}+b_{18}+c_{19}=x_3 \end{cases} \qquad \begin{cases} a_{18}+b_{20}+c_{21}=y_1 \\ a_{17}+b_{19}+c_{20}=y_2 \\ a_{16}+b_{18}+c_{19}=y_3 \end{cases}$$

Since $x_3 = y_3$, this yields a system of 5 equations, which are in this case independent. So we may express 5 variables occurring in this system through expressions with 4 remaining variables. For instance:

$$\begin{cases} a_{16}=b_{20}+c_{20}+x_1+x_2+y_2 \\ a_{17}=b_{20}+c_{21}+x_1 \\ a_{18}=b_{20}+c_{21}+y_1 \\ b_{18}=b_{20}+c_{19}+c_{20}+x_1+x_2+x_3+y_2 \\ b_{19}=b_{20}+x_1+y_2 \end{cases}$$

In this example no values of $x_i,y_i$ can contradict the system, which could only happen, if the rank was lower than the number of equations. However, for instance for RSP5 $(100,00110,0011)$ the system considered consists of 9 equations and 13 variables, rank of its left side is 7 and it has a solution if and only if

$$\begin{cases} x_1+x_2+y_1+y_2=0 \\ x_1+x_3+y_1+y_3=0 \end{cases}$$

So this RSP5 will be excluded for 75% cases of the values of $x_i,y_i$.

In general, systems of equations defined in Phase 3 are underdefined, but the rank of the system matrix is quite high; more thorough statistics can be found in the Appendix A.1 [12]. If we consider only resynchronization after 5, 6, 7, 8 and 9 steps, then there are 11682 RSPs (with different lengths.) We may also consider *fixed length* RSPs by filling in additional bits so that all RSPs have the same length. Then we get 1992622 RSPs of length 27. The average case is that resynchronization occurs after 6.70807 steps, (the variable length) RSP contains 16.421 bits, the system of linear equations contains 17.1326 variables and its rank is 10.5344. So once we choose an RSP, on average we express 33.5536 bits of the shift registers by guessing some 6.5982 bits. Since the difference between the number of equations and the rank is 1.88174, for an average system, in about 73% cases RSP guess is not consistent with the bits $x_i,y_i$, so it can be quickly filtered out.

**Uncertainty About Resynchronization Pattern Length.** Unfortunately, we cannot be sure about the exact number of steps before resynchronization occurs: even if the

output bits from a given step onwards are the same, it may happen that resynchronization occurs a few steps later, and the same keystream bits before resynchronization came out by accident. If resynchronization occurs after step $t$, then with probability approximately $\frac{1}{2}$ the outputs are the same already after step $t-1$ (we would have probability exactly $\frac{1}{2}$ after replacing each shift register by an independent random generator.) However, finding exact probability that output of an RSP$n$ will synchronize after step $n-1, n-2, \ldots$ is nontrivial due to some subtleties of resynchronization. For instance consider an RSP4 $(011,1101,1101)$. Denote the contents of the registers exactly as in the case of an RSP3. Then the output sequence starting from the moment of injecting a fault equals

$$a_{18}+b_{20}+c_{21}, \quad a_{18}+b_{19}+c_{20}, \quad a_{17}+b_{18}+c_{19}, \quad a_{16}+b_{17}+c_{18}\ ,$$

while the correct system outputs

$$a_{17}+b_{20}+c_{21}, \quad a_{17}+b_{19}+c_{20}, \quad a_{16}+b_{18}+c_{19}, \quad a_{16}+b_{17}+c_{18}\ .$$

So the output for this RSP4 will synchronize after step 4 if and only if

$$a_{16}+b_{18}+c_{19} \neq a_{17}+b_{18}+c_{19}$$

or just $a_{16} \neq a_{17}$ for which probability is about $\frac{1}{2}$. We observe the same output already after step 3 if

$$\begin{cases} a_{16}+b_{18}+c_{19} = a_{17}+b_{18}+c_{19} \\ a_{17}+b_{19}+c_{20} \neq a_{18}+b_{19}+c_{20}\ , \end{cases}$$

that is, if $a_{16} = a_{17}, a_{17} \neq a_{18}$. This occurs with probability roughly $\frac{1}{4}$. For observing output resynchronization after step 2 conditions are

$$\begin{cases} a_{16}+b_{18}+c_{19} = a_{17}+b_{18}+c_{19} \\ a_{17}+b_{19}+c_{20} = a_{18}+b_{19}+c_{20} \\ a_{17}+b_{20}+c_{21} \neq a_{18}+b_{20}+c_{21}\ , \end{cases}$$

that is, if $a_{16} = a_{17}$, $a_{17} = a_{18}$, $a_{17} \neq a_{18}$, which can never happen. So the remaining probability is allocated to the event that the outputs get the same already after step 1 after the moment of fault injection.

Exactly the same effects can be observed for the following RSP4: $(100,0010,0010)$, $(011,1101,110)$, $(011,110,1101)$, $(100,0010,001)$, $(100,001,0010)$. For two other RSP4, namely $(0111,1111,1111)$ and $(1000,0000,0000)$, the probabilities of resynchronization of the output after step 4, 3, 2 and 1 are, respectively, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ and $\frac{1}{8}$. Thus, if we consider fixed length RSP4s (all consisting of 12 bits), then we get two RSPs for which the outputs can re-synchronize after step 2, and 20 RSPs for which it is impossible. So probability that output of random registers with RSP4 resynchronizes after 2nd step is $\frac{1}{8} \cdot \frac{1}{11} = 0.011(36)$. Experimental results are very close to these values.

**Table 1.** Which RSP is responsible for resynchronization of outputs

| observed | | actual RSP length | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| RSP | prob. | $n$ | $n+1$ | $n+2$ | $n+3$ | $n+4$ | $n+5$ | $n+6$ | $> n+6$ |
| $n=1$ | 1.93% | 0% | 80.84% | 5.05% | 6.63% | 3.42% | 1.79% | ... 2.27% | ... |
| $n=2$ | 1.72% | 91.29% | 5.70% | 0.36% | 1.18% | ..... 1.47% | | ..... | |
| $n=3$ | 0.39% | 49.41% | 34.02% | 4.05% | 5.16% | 3.13% | 1.81% | ... 2.42% | ... |
| $n=4$ | 0.43% | 62.07% | 23.64% | 3.16% | 4.72% | 2.72% | 1.58% | ... 2.11% | ... |
| $n=5$ | 0.34% | 60.61% | 23.33% | 3.76% | 5.11% | 3.03% | 1.78% | ... 2.38% | ... |
| $n=6$ | 0.28% | 56.47% | 25.53% | 4.32% | 5.63% | 3.38% | 2.00% | 1.16% | 1.51% |
| $n=7$ | 0.25% | 56.84% | 24.91% | 4.43% | 5.68% | 3.39% | 2.01% | 1.18% | 1.56% |
| $n=8$ | 0.22% | 56.06% | 25.12% | 4.62% | 5.81% | 3.50% | 2.07% | 1.20% | 1.62% |
| $n=9$ | 0.20% | 55.60% | 25.21% | 4.77% | 5.85% | 3.55% | 2.10% | 1.22% | 1.70% |

# 4 Cryptanalysis Implementation

## 4.1 Some Notes on Implementation

**Phase 1.** We have to guess which RSP really occurred: we know we should not rely on outputs' similarities only. For that reason, we should rather exclude situations where outputs seem to synchronize after 9 steps – investigating Table 1 shows that the chances that some of RSP$k$, $k \in \{5,6,7,8,9\}$ occurred are rather slim (about 55%.) Therefore we shall concentrate on outputs that seem to synchronize after 5 to 8 steps. Of course if we have observed resynchronization after, say, 7 steps, we need not to consider RSP5s and RSP6s. Each RSP guess gives us possible values of some bits within the registers – typically 12 to 27. We may think of these as of trivial equations (*variable* = *value*.) Obviously, these equations are independent.

**Phase 2.** This phase basically sieves the possible guesses from Phase 1 out, and adds some equations given by the guess. Phase 2 gives significant boost to our calculations: not only we are able to quickly falsify about 73% of the wrong guesses from the previous phase, but also, since we are considering two different outputs at the same time, given by two different clockings, we get twice as much equations. Some of them are unfortunately dependent; luckily, the rank of an appropriate matrix is still quite high. As it has been pointed before, because of the moderate number of RSPs it is possible to precompute the equations given by every single RSP – this significantly speeds up the search and also makes the whole search easier to implement on parallel computers.

**Phase 3.** In Phase 3 we try to fill the bits on the left side of the pattern guessed in Phase 1 (i.e. we add trivial equations describing some bits close to the left end of the registers) and afterwards we construct more equations, just as in Phase 2, concerning the bits that are at the rightmost positions when the newly guessed bits enter the clocking window. Of course now only one equation for each step is given. Once non-contradictory set of 64 independent equations is found, we solve it and check if the solution is consistent with output given by an original device. A rough estimation of the number of cases

shows that about $2^{34.23}$ systems of linear equations with 64 unknowns need to be solved in an average case. More details regarding complexity of our approach can be found in Appendix A.2, see [12].

It is also possible to perform Phase 3 in a different, simplified way: guess gradually bits in interesting positions and check it against the known output. This approach can be used when for some reason one does not want to perform lots of matrix operations necessary to solve linear equations' system.

Let us remark that in this attack, compared to the the previous ones [4, 10], the number of cases to be considered has been reduced about 100 times. This is possible since we take for cryptanalysis only very particular sequences. On the other hand, their frequency is close to 1.5%. This is advantageous, because generally cost of running / simulating A5/1 is negligible comparing to cryptanalysis' cost.

### 4.2 Test Implementation

We have tested our cryptanalysis procedure on a home computer, namely AMD Athlon XP 1800+ based PC running Debian GNU/Linux.

In the precomputational phase we have found and stored all RSPs and corresponding systems of equations together with their solutions and conditions under which the system has solutions. The space required for the data is less than 1MB and the computations were performed on the same home PC.

Our proof-of-concept implementation finds the whole 64-bit-long contents of the three registers given two outputs that resynchronize after 5 to 8 steps. This first implementation is not (yet) optimized for code efficiency. We believe that a lot of fine tuning in algorithm design is possible to speed it up considerably. In fact, many directions seem to be promising; the problem is to choose the most efficient tricks.

When assumed RSP is of length 5, after Phases 1 and 2 we are left with about one thousand (partially filled) candidates, checking each of them takes few minutes on average. For (assumed) RSP of length 9 Phases 1 and 2 yield about one million candidates, but hundreds of them may be checked in a second.

It is also worth noting that the computation very easily scales up to a larger number of CPUs: different processors can simply check different candidates in parallel.

## 5 Technical Feasibility

One may try to implement fault attack using test capabilities of integrated circuits (IC). It seems to be a plausible idea, since testing should enable to run single components, that is, disable the other components. However, while a "raw" chip or die typically has extra test pins accessible via needles, they are not included in the packaging of the IC and cannot be accessed from the circuit board. The package of an IC must be opened for such purpose, which requires a highly sophisticated semiconductor test lab's equipment.

Manipulation of functional chips based on intrusive technologies is a different subject. Internal structures of an IC can be influenced by needles mechanically and electrically, by electron and ion beams electrically, and optically by lasers. In today's deep

sub-micron structures, using needles is no longer feasible except for specific pad areas. However, a focused charged electron or ion beam can very well influence specific wires of an IC or even a single memory cell. Re-scattered electrons from a charged lines can be observed on a scanning electron microscope and can exhibit whether such a line is at "low" or at "high." Again the advances in semiconductor technology limit such mechanisms. With up to seven layers of metal interconnects stacked upon each other in state-of-the-art digital CMOS technologies [1], the higher levels will shield the lower level effectively. Removal of higher layers and successive active operation of lower levels for observation is close-to impossible. Furthermore, shielding of signal lines from observation is quite well possible by physical IC design. Because of that, if the chip have been designed without some special precautions, our attack is of little or no concern.

**Trapdoors.** With currently available IC technology possibilities of external manipulations of clocking seem to be limited. However, the manufacturers may introduce some trapdoors through a certain IC design – one can put a certain line on a higher level in order to make it vulnerable to intrusive technologies. In such a scenario it would be necessary to possess an advanced test lab to perform the synchronization attack. This situation might be comfortable for security agencies – the number of manufacturers with the access to sophisticated semiconductor technology is small and they can be relatively easily monitored. Controlling all / majority of parties with appropriate technological tools would successfully limit the potential attackers, and at the same time allow the secret services to use the synchronization attack.

## 6 Conclusions

We have shown that if one can stop clocking in a chosen register for a single step, and run A5/1 for the same initial contents, then with a reasonable number of experiments one can find a case in which the contents of the registers can be fully reconstructed.

So the main point is that during security evaluation of hardware devices implementing A5/1 one has to prove infeasibility of resynchronization attack against the A5/1 component with physical equipment available today.

**Further Attacks.** One can consider different scenarios in which essentially the same idea can be applied. Consider the case in which we can change a single bit at a random position in one of the registers. Attacks of this kind should be considered very carefully, since such faults might be likely in appropriate physical conditions. The details of retrieving the contents of the shift registers are different in this case (perhaps more confusing) but the general attack idea based on resynchronization remains the same.

If one could assume the possibility of changing the state of consecutive bits of chosen register to 1, then another attack can be applied. In this case no knowledge of correct sequence is needed, see [11].

**Source of Weaknesses.** The problem with the A5/1 algorithm is that the keystream is generated from quite weak components, even if their outputs are combined in a clever way. So a distortion of a single component has limited consequences and the effect

may cancel out in certain circumstances – which is exactly the opposite to an avalanche effect.

It turns out that taking the bits for clocking in the middle (which is reasonable from the point of view of the previous attacks) becomes advantageous for synchronization attack. Namely, the following design aspects help to mount the attack:

- for a relatively long resynchronization period no bit of the resynchronization pattern reaches the output position, while
- all the bits of resynchronization pattern are already in the registers when the fault occurs.

However, the main feature is that one can consider separately the bits on each side of the resynchronization pattern. This reduces complexity of the attack against brute force by an order of magnitude.

**Countermeasures.** The change of the registers' length does not reduce the gain obtained in our attack, it only influences the number of the remaining bits that have to be found.

Another way to defend against synchronization attacks would be redesigning the way in which the shift registers cooperate. Certainly, considering more than one clocking window in each register would make the attack much harder – the resynchronization pattern would consist of many blocks of bits. Consequently, we would have to choose shorter patterns and in this way gain less information on the remaining bits. However, we cannot be sure that such a modification does not bring new dangers. Additionally, one can use feedback bits from all three registers for each of the shift registers. In such a case resynchronization would be unlikely.

## References

1. A. Allan, D. Edenfield, W. H. Joyner, A. B. Kahng, M. Roger, Y. Zorian, *2001 Technology Roadmap for Semiconductors*, IEEE Computers, Jan. 2002, pp. 42–53
2. Elad Barkan, Eli Biham, Nathan Keller, *Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication*, Crypto'2003, LNCS 2729, Springer, 2003 , pp. 600–616
3. Eli Biham, Adi Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, Crypto'97, LNCS 1294, Springer, 1997, pp. 513–525
4. Eli Biham, Orr Dunkelman, *Cryptanalysis of the A5/1 GSM stream cipher*, INDOCRYPT'2000, LNCS 1977, Springer, 2000, pp. 43–51
5. Alex Biryukov, Adi Shamir, David Wagner, *Real time cryptanalysis of A5/1 on a PC*, FSE'2000, LNCS 1978, Springer, 2001, pp. 1–18
6. Dan Boneh, Richard A. DeMillo, Richard J. Lipton, *On the importance of checking cryptographic protocols for faults*, Eurocrypt'97, LNCS 1233, Springer 1997, pp. 37–51
7. Marc Briceno, Ian Goldberg, David Wagner, *A pedagogical implementation of A5/1 and A5/2 "voice privacy" encryption algorithms*, `http://cryptome.org/gsm-a512.htm`, 1999
8. Don Coppersmith, Hugo Krawczyk, Yishay Mansour, *The Shrinking Generator*, Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology, LNCS 773, Springer, 1993, pp. 22–39
9. Pierre Dusart, Gilles Letourneux, Olivier Vivolo, *Differential Fault Analysis on A.E.S.*, ACNS'2003, LNCS 2846 Springer 2003, pp. 293–306

10. Jovan Dj. Golič, *Cryptanalysis of Alleged A5 Stream Cipher*, Eurocrypt'97, LNCS 1233, Springer, 1997, pp. 239–255
11. Marcin Gomułkiewicz, Mirosław Kutyłowski, Paweł Wlaź, *Fault Cryptanalysis for Breaking A5/1*, to appear in Tatra Mountains Mathematical Publications
12. Marcin Gomułkiewicz, Mirosław Kutyłowski, Heinrich Theodor Vierhaus, Paweł Wlaź, *Synchronization Fault Cryptanalysis for Breaking A5/1. Appendix*, `http://kutylowski.im.pwr.wroc.pl/a5/` or `http://mat.pol.lublin.pl/a5/`, 2005
13. Marc Joyce, Arjen K. Lenstra, Jean-Jacques Quisquater, *Chinese Remaindering Based Cryptosystems in the Presence of Faults*, J. of Cryptology 12(4), 1999, pp. 241–245

# An Efficient Algorithm for $\delta$-Approximate Matching with $\alpha$-Bounded Gaps in Musical Sequences

Domenico Cantone, Salvatore Cristofaro, and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica,
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone, cristofaro, faro}@dmi.unict.it

**Abstract.** We present a new efficient algorithm for the $\delta$-approximate matching problem with $\alpha$-bounded gaps. The $\delta$-approximate matching problem, recently introduced in connection with applications in music retrieval, generalizes the exact string matching problem by relaxing the notion of matching. Here we consider the case in which matchings may contain bounded gaps.

An extensive comparison with the only (to our knowledge) other solution existing in the literature for the same problem, due to Crochemore *et al.*, indicates that our algorithm is more efficient, especially in the cases of large alphabets and long patterns. In addition, our algorithm computes the total number of approximate matchings for each position of the text, requiring only $\mathcal{O}(m\alpha)$-space, where $m$ is the length of the pattern.

**Keywords:** approximate string matching, experimental algorithms, musical information retrieval.

## 1 Introduction

Given a text $T$ and a pattern $P$ over some alphabet $\Sigma$, the *string matching problem* consists in finding *all* occurrences of $P$ in $T$. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

Recently, the classical string matching problem has been generalized to various notions of approximate matching, particularly useful in specific fields such as molecular biology [9], musical applications [3], or image processing [8].

In this paper we focus on a variant of the approximate string matching problem, namely the *$\delta$-approximate string matching problem with $\alpha$-bounded gaps*. Such a problem, which will be given a precise definition later, arises in many questions concerning musical information retrieval and musical analysis. This is especially true in the context of monophonic music, in which one wants to retrieve a given melody from a complex musical score. We mention here that

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 428–439, 2005.
© Springer-Verlag Berlin Heidelberg 2005

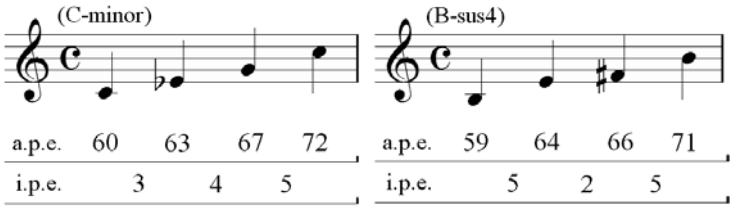



**Fig. 1.** Representation of the C-minor and B-sus4 chords in the absolute pitch encoding (a.p.e.) and in the interval pitch encoding (i.p.e.)

a significant amount of research has been devoted to adapt solutions for exact string matching to $\delta$-approximate matching (see for instance [1], [5], [6], [2]). In this respect, Boyer-Moore-type algorithms are of particular interest, since they are very fast in pratice.

### 1.1 Approximate Matching in Musical Sequences

Musical sequences can be schematically viewed as sequences of integer numbers, representing either the notes in the chromatic or diatonic notation (absolute pitch encoding), or the intervals, in number of semitones, between consecutive notes (interval pitch encoding); see the examples in Fig. 1. The second representation is generally of greater interest for applications in tonal music, since absolute pitch encoding disregards tonal qualities of pitches. Note durations and note accents can also be encoded in numeric form, giving rise to richer alphabets whose symbols can really be regarded as sets of parameters. This is the reason why alphabets used for music representation are generally quite large.

$\delta$-approximate string matching algorithms are very effective to search for all similar but not necessarily identical occurrences of given melodies in musical scores. We recall that in the $\delta$-approximate matching problem two integer strings of the same length match if the corresponding integers differ by at most a fixed bound $\delta$. For instance, the chords C-minor and B-sus4 match if a tolerance of $\delta = 1$ is allowed in the absolute pitch encoding (where C-minor$= (60, 63, 67, 72)$ and B-sus4$= (59, 64, 66, 71)$), whereas if we use the interval pitch encoding, a tolerance of $\delta = 2$ is required to get a match (in this case we have C-minor$= (3, 4, 5)$ and B-sus4$= (5, 2, 5)$); see Fig. 1. Notice that for $\delta = 0$, the $\delta$-approximate string matching problem reduces to the exact string matching problem.

Intuitively, we say that a melody (or *pattern*) has a $\delta$-approximate occurrence with $\alpha$-bounded gaps within a given musical score (or *text*), if the melody has a $\delta$-approximate matching with a subsequence of the musical score, in which it is allowed to skip up to a fixed number $\alpha$ of symbols (the *gap*) between any two consecutive positions. In the present context, two symbols have an approximate matching if the absolute value of their difference is bounded by a fixed number $\delta$.

In classical music compositions, and in particular in compositions for *Piano Solo*, it is quite common to find musical pieces based on a sweet ground melody, whose notes are interspaced by rapidly executed arpeggios. Fig. 2 shows two bars

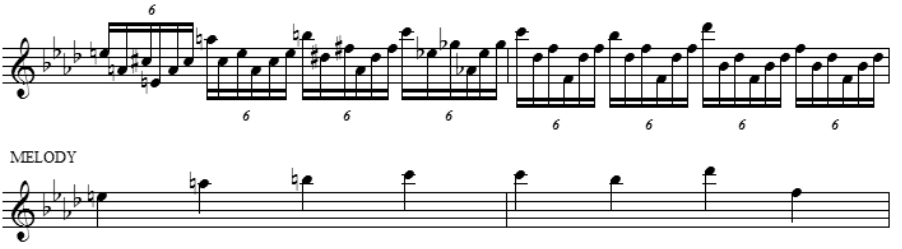



**Fig. 2.** Two bars of the study Op. 25 N. 1 of F. Chopin (first score). The second score represents the melody

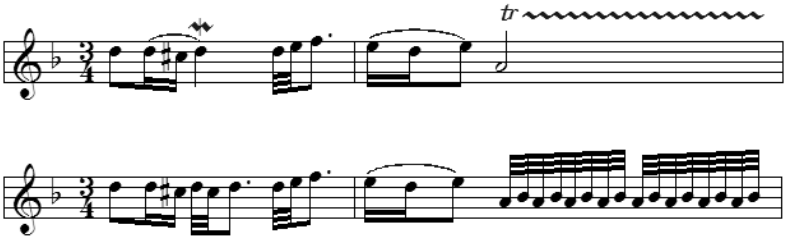



**Fig. 3.** An excerpt of a piece of J.S. Bach (first score). The second score shows how the musical ornaments must be played. Two musical ornaments are present: a *mordent*, attached to the $4^{th}$ note, and a *trill*, attached to the $11^{th}$ note

of the study *Op. 25 N. 1 for Piano Solo* by F. Chopin illustrating such a point. The notes of the melody are the first of each group of six notes (sextuplet). If we use the standard MIDI representation of the pitches, then the melody corresponds to the sequence of integer numbers $P = [76, 81, 83, 84, 84, 83, 86, 77]$. Then, if a gap bound of $\alpha = 5$ is allowed, an exact occurrence of the melody can be found through the piece.

The above musical technicality is not by any means the only one for which approximate string matching with bounded gaps turns out to be very useful. Other examples are given by *musical ornaments*, which are common practice in classical music, and especially in the music of the baroque period. Musical ornaments are groups of notes, played at a very fast tempo, which generally are "attached" to the notes of a given melody, in order to emphasize or adorn certain dynamical passages. Some of the most common musical ornaments are the *acciaccatura*, the *appoggiatura*, the *mordent*, the

Fig. 3 shows an excerpt of a *Minuet* by J.S. Bach, which makes use of musical ornaments. We provide both the actual score, in which ornaments are represented as special symbols marked above notes or as groups of small notes, and the corresponding score showing how these notations translate into real musical execution. Note that in Fig. 3 the mordent corresponds to a group of three notes, whereas the trill corresponds to a group of 16 notes. In general, to take care of

musical ornaments in $\delta$-matching problem with gaps, one needs gap values in the range between 4 and 16.

### 1.2 Paper's Organization

The paper is organized as follows. In Section 2 we introduce some basic notions and give a formal definition of the $\delta$-approximate matching problem with $\alpha$-bounded gaps. An algorithm based on the dynamic programming approach for the approximate matching problem of our interest is reviewed in Section 3. Then, in Section 4, we present a new efficient algorithm for the same problem. Experimental data obtained by running under various conditions both our algorithm and the one based on the dynamic programming approach are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

## 2 Basic Definitions and Properties

Before entering into details, we need a bit of notation and terminology. A string $P$ is represented as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that $P$ has length $m$ and write $\text{length}(P) = m$. In particular, for $m = 0$ we obtain the empty string. By $P[i]$ we denote the $(i+1)$-st character of $P$, for $0 \leq i < \text{length}(P)$. Likewise, by $P[i\,..\,j]$ we denote the substring of $P$ contained between the $(i+1)$-st and the $(j+1)$-st characters of $P$, for $0 \leq i \leq j < \text{length}(P)$. The substrings of the form $P[0\,..\,j]$ (also denoted by $P_j$), with $0 \leq j < \text{length}(P)$, are the nonempty *prefixes* of $P$.

Let $\Sigma$ be an alphabet of integer numbers and let $\delta \geq 0$ be an integer. Two symbols $a$ and $b$ of $\Sigma$ are said to be *$\delta$-approximate* (or we say that $a$ and $b$ *$\delta$-match*), in which case we write $a =_\delta b$, if $|a-b| \leq \delta$. Two strings $P$ and $Q$ over the alphabet $\Sigma$ are said to be $\delta$-approximate (or we say that $P$ and $Q$ $\delta$-match), in which case we write $P \stackrel{\delta}{=} Q$, if

$$\text{length}(P) = \text{length}(Q), \quad \text{and } P[i] =_\delta Q[i], \quad \text{for } i = 0, ..., \text{length}(P) - 1\,.$$

Given a text $T$ of length $n$ and a pattern $P$ of length $m$, a *$\delta$-occurrence with $\alpha$-bounded gaps of $P$ in $T$ at position $i$* is an increasing sequence of indices $(i_0, i_1, \ldots, i_{m-1})$ such that (i) $0 \leq i_0$ and $i_{m-1} = i \leq n-1$, (ii) $i_{h+1} - i_h \leq \alpha+1$, for $h = 0, 1, \ldots m-2$, and (iii) $P[j] =_\delta T[i_j]$, for $j = 0, 1, \ldots m-1$. We write $P \trianglelefteq^i_{\delta,\,\alpha} T$ to mean that $P$ has a $\delta$-occurrence with $\alpha$-bounded gaps in $T$ at position $i$ (in fact, when the bounds $\delta$ and $\alpha$ are well understood from the context, we will simply write $P \trianglelefteq^i T$).

The *$\delta$-approximate string matching problem with $\alpha$-bounded gaps* admits the following variants: (a) find all $\delta$-occurrences with $\alpha$-bounded gaps of $P$ in $T$; (b) find all positions $i$ in $T$ such that $P \trianglelefteq^i T$; (c) for each position $i$ in $T$, find the number of distinct $\delta$-occurrences of $P$ with $\alpha$-bounded gaps at position $i$.

In Section 4 we will describe an efficient $\mathcal{O}(mn)$-time solution for the variants (b) and (c) above which uses only $\mathcal{O}(m\alpha)$ extra space. Variant (a) can then be solved by running an $\mathcal{O}(m^2\alpha)$-time and -space local search at each position $i$ such that $P \trianglelefteq^i T$.

The following very elementary fact will be used later.

**Lemma 1.** *Let $T$ and $P$ be a text of length $n$ and a pattern of length $m$, respectively. Also, let $\delta, \alpha \geq 0$. Then, for each $0 \leq i < n$ and $0 \leq k < m$, we have that $P_k \trianglelefteq^i_{\delta,\,\alpha} T$ if and only if $P[k] =_\delta T[i]$ and either $k = 0$, or $P_{k-1} \trianglelefteq^{i-h}_{\delta,\,\alpha} T$, for some $h$ such that $1 \leq h \leq \alpha + 1$.* ∎

## 3 A Dynamic Programming Algorithm: $\delta$-Bounded-Gaps

The $\delta$-approximate matching problem with $\alpha$-bounded gaps has first been addressed by Crochemore *et al.* in [7], where an algorithm based on the dynamic programming approach—named $\delta$-Bounded-Gaps—has been proposed. To our knowledge, this is still the only solution present in literature for the $\delta$-approximate matching problem with $\alpha$-bounded gaps. In our review, we follow the presentation given later in [4], which also considers several new versions of the approximate matching problem with gaps.

Given as usual a text $T$ of length $n$, a pattern $P$ of length $m$, and two integers $\delta, \alpha \geq 0$, the algorithm $\delta$-Bounded-Gaps runs in $\mathcal{O}(mn)$-time and -space, at least in the case in which one is interested in finding all $\delta$-occurrences with $\alpha$-bounded gaps of $P$ in $T$ (variant (a)). Space requirements can be reduced to $\mathcal{O}(n)$, if only positions $i$ in $T$ such that $P \trianglelefteq^i T$ need to be computed (variant (b)). To solve also variant (c) with $\delta$-Bounded-Gaps, one needs to first solve variant (a) and then trace back and count all approximate matchings with gaps at each position of the text $T$.

The algorithm $\delta$-Bounded-Gaps is presented as an incremental procedure, based on the dynamic programming approach. More specifically, it starts by first computing all the $\delta$-occurrences with $\alpha$-bounded gaps in $T$ of the prefix of $P$ of length 1, i.e. $P_0$. Then, during the $i$-th iteration, it looks for all the $\delta$-occurrences with $\alpha$-bounded gaps in $T$ of the prefix $P_{i-1}$. At the end of the last iteration, the $\delta$-occurrences of the whole pattern $P$ have been computed.

To give a more formal description of the algorithm, let us put:

$$\mathit{LastOccur}_j(P_i) = \max\left(\{0 \leq k \leq j : P_i \trianglelefteq^k T \text{ and } j - k \leq \alpha\} \cup \{-1\}\right) .$$

Notice that if $\mathit{LastOccur}_j(P_i) = -1$, then $P_i \not\trianglelefteq^k T$ for $k = j - \alpha, j - \alpha + 1, \ldots, j$. Otherwise, $\mathit{LastOccur}_j(P_i)$ is the largest value $k \in \{j - \alpha, j - \alpha + 1, \ldots, j\}$ such that $P_i \trianglelefteq^k T$.

Using a *trace-back* procedure, as described in [4], the values $\mathit{LastOccur}_j(P_i)$ can be used to retrieve the approximate matchings at a given position in time $\mathcal{O}(m\alpha)$.

The values $\mathit{LastOccur}_j(P_i)$ can be computed incrementally, for $0 \leq i < m$ and $0 \leq j < n$. More specifically, the algorithm $\delta$-Bounded-Gaps fills a matrix $D$

```
δ-Bounded-Gaps (P, T, δ, α)
 1.   n = length(T)
 2.   m = length(P)
 3.   for i = 0 to m − 1 do
 4.        D[i, 0] = −1
 5.   if P[0] =_δ T[0] then
 6.        D[0, 0] = 0
 7.   for i = 0 to m − 2 do
 8.        for j = 1 to n − 1 do
 9.             D[i, j] = −1
10.             if ((P[i] =_δ T[j]) and (i = 0 or D[i − 1, j − 1] ≥ 0)) then
11.                  D[i, j] = j
12.             else if D[i, j − 1] ≥ j − α then
13.                       D[i, j] = D[i, j − 1]
14.   for j = m − 1 to n − 1 do
15.        if P[m − 1] =_δ T[j] and D[m − 2, j − 1] ≥ 0 then
16.             output(j)
```

**Fig. 4.** The algorithm $\delta$-Bounded-Gaps for the $\delta$-approximate matching problem with $\alpha$-bounded gaps

of dimension $m \times n$, where $D[i, j]$ corresponds to $LastOccur_j(P_i)$, according to the following recursive relation:

$$D[i,j] = \begin{cases} j & \text{if } T[j] =_\delta P[i] \text{ and} \\ & \text{- } i = 0\,, \text{ or} \\ & \text{- } i, j \geq 1 \text{ and } D[i-1, j-1] \geq 0 \\ D[i, j-1] & \text{if } j \geq 1\,,\ D[i, j-1] \geq j - \alpha\,, \text{ and} \\ & \text{- } T[j] \neq_\delta P[i]\,, \text{ or} \\ & \text{- } T[j] =_\delta P[i]\,,\ i \geq 1\,, \text{ and } D[i-1, j-1] < 0 \\ -1 & \text{otherwise} \end{cases}$$

where $0 \leq i < m$ and $0 \leq j < n$.

Fig. 4 presents the pseudo-code of the algorithm $\delta$-Bounded-Gaps. Its running time is easily seen to be $\mathcal{O}(mn)$. Also, $\mathcal{O}(mn)$-space is needed to store the matrix $D$. However, if one is only interested in the positions $i$ of $T$ at which $P \trianglelefteq^i T$, space requirements reduce to $\mathcal{O}(n)$, since the computation of each row depends only on the values stored in the previous row.

## 4 A New Efficient Algorithm: $(\delta, \alpha)$-Sequential-Sampling

In this section we present a new efficient algorithm for the $\delta$-approximate matching problem with $\alpha$-bounded gaps, named $(\delta, \alpha)$-Sequential-Sampling. Our

algorithm is characterized by an $\mathcal{O}(mn)$-time and an $\mathcal{O}(m\alpha)$-space complexity, where $m$ and $n$ are the length of the pattern and text, respectively. Notice that in practical applications $m\alpha$ is much smaller than $n$. In addition, our algorithm solves variant (c) (and therefore also variant (b)) of the approximate matching problem with gaps, as stated in Section 2, namely it computes the number of distinct $\delta$-occurrences of the pattern with $\alpha$-bounded gaps at each position of the text. If one is also interested in retrieving the actual approximate matching occurrences at position $i$ of a text $T$, a possibility would be to compute the submatrix $D[k, j]$, for $\max(0, (m-1)\cdot(\alpha+1)) \leq k \leq i$ and $0 \leq j \leq m-1$, where, as before, $m$ is the length of the pattern, and then trace back through all possible approximate matchings. The submatrix $D[k, j]$ can be computed in time and space $\mathcal{O}(m^2\alpha)$ by the algorithm $\delta$-Bounded-Gaps.

Our algorithm follows a different computation ordering than the one followed by the algorithm $\delta$-Bounded-Gaps; in fact, it computes the occurrences of all prefixes of the pattern in continuously increasing prefixes of the text, rather than computing all occurrences in the whole text of continuously increasing prefixes of the pattern, as the algorithm $\delta$-Bounded-Gaps does. That is, for a text $T$ of length $n$, pattern $P$ of length $m$, and nonnegative integers $\delta, \alpha$, during its first iteration the algorithm $(\delta, \alpha)$-Sequential-Sampling computes the (number of) occurrences of all prefixes $P_k$ of $P$ such that $P_k \trianglelefteq^0 T$. Then, during the $i$-th iteration, it computes (the number of) all occurrences of prefixes $P_k$ of $P$ such that $P_k \trianglelefteq^{i-1} T$, using information gathered during previous iterations.

To be more precise, let $\mathcal{S}_i$ denote the collection of all pairs $(j, k)$ such that $P_k \trianglelefteq^j T$, for $0 \leq i \leq n$, $0 \leq j < i$, and $0 \leq k < m$. Notice that $\mathcal{S}_0 = \varnothing$. If we put $\mathcal{S} = \mathcal{S}_n$, then the problem of finding the positions $i$ in $T$ such that $P \trianglelefteq^i T$ translates to the problem of finding all values $i$ such that $(i, m-1) \in \mathcal{S}$.

To begin with, notice that Lemma 1 justifies the following recursive definition of the set $\mathcal{S}_{i+1}$ in terms of $\mathcal{S}_i$, for $i < n$:

$$\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{(i,k) : P[k] =_\delta T[i] \text{ and } (k = 0 \text{ or } (i-h, k-1) \in \mathcal{S}_i, \text{ for some } h \in \{1, \ldots, \alpha+1\})\}.$$

This relation, coupled with the initial condition $\mathcal{S}_0 = \varnothing$, allows one to compute the set $\mathcal{S}$ in an iterative fashion, as shown in Fig. 5. The time complexity of the resulting algorithm—named Slow-$(\delta, \alpha)$-Sequential-Sampling—is $\mathcal{O}(nm\alpha)$. Notice that given the set $\mathcal{S}$, one can easily compute the actual $\delta$-occurrences with $\alpha$-gaps of $P$ in $T$.

From a practical point of view, the set $\mathcal{S}$ in the algorithm Slow-$(\delta, \alpha)$-Sequential-Sampling could be represented by its characteristic $n \times m$ matrix $\mathcal{M}$, where $\mathcal{M}[i, k]$ is 1 or 0, provided that the pair $(i, k)$ belongs or does not belong to $\mathcal{S}$, for $0 \leq i < n$ and $0 \leq k < m$.

```
Slow-(δ, α)-Sequential-Sampling (T, P, δ, α)
 1.  n = length(T)
 2.  m = length(P)
 3.  S = ∅
 4.  for i = 0 to n − 1 do
 5.      for k = m − 1 downto 1 do
 6.          if P[k] =_δ T[i] and (i − h, k − 1) ∈ S, for some h ∈ {1, ..., α + 1} then
 7.              S = S ∪ {(i, k)}
 8.      if P[0] =_δ T[i] then
 9.          S = S ∪ {(i, 0)}
10.  for i = 0 to n − 1 do
11.      if (i, m − 1) ∈ S then
12.          output(i)
```

**Fig. 5.** The algorithm Slow-$(\delta, \alpha)$-Sequential-Sampling for the $\delta$-approximate matching problem with $\alpha$-bounded gaps

Since during the $i$-th iteration of the **for**-loop at line 4 of the algorithm Slow-$(\delta, \alpha)$-Sequential-Sampling at most $\alpha + 1$ rows of $\mathcal{M}$ need to be scanned (more precisely the ones having index $j \in \{\max(0, i-\alpha-1), i-1\}$), it would be enough to store only $\alpha+1$ rows of $\mathcal{M}$ at each step of the computation, plus another one as working area.

In addition, by maintaining an extra array $\mathcal{C}$ of length $m$ such that during the $i$-th iteration of the **for**-loop at line 4 the following invariant holds:

$$\mathcal{C}[k] = \sum_{j=\max(0,i-\alpha-1)}^{i-1} \mathcal{M}[j,k]\,, \qquad \text{for } 0 \le k < m\,,$$

the test of the conditional instruction at line 6 can be performed in constant time, rather than in $\mathcal{O}(\alpha)$-time.

Such observations allow to reduce the space requirement to $\mathcal{O}(m\alpha)$ and the running time to $\mathcal{O}(mn)$.

In fact, we can do a little bit more than that. Rather than maintaining in $\mathcal{M}[j,k]$ the Boolean value of the test $(j,k) \in \mathcal{S}$, it is more convenient to let $\mathcal{M}[j,k]$ count the number of *distinct* $\delta$-occurrences with $\alpha$-gaps of $P_k$ at position $j$ of $T$. With this change, when the $i$-th iteration of the **for**-loop at line 4 of algorithm Slow-$(\delta, \alpha)$-Sequential-Sampling starts, the item $\mathcal{C}[k]$ will contain the total number of *distinct* $\delta$-occurrences with $\alpha$-bounded gaps of $P_k$ at positions $\max(0, i - \alpha - 1)$ through $i - 1$, provided that the above invariant holds. Such values can then be used to maintain the invariant itself.

Plainly, at the end of the computation one can retrieve in constant time the number of approximate matchings at each position of the text.

The resulting algorithm—named $(\delta, \alpha)$-Sequential-Sampling—is presented in detail in Fig. 6. Its overall running time is $\mathcal{O}(mn)$ and its space requirement is $\mathcal{O}(m\alpha)$.

```
(δ, α)-Sequential-Sampling (T, P, δ, α)
 1.   n = length(T)
 2.   m = length(P)
 3.   for i = 0 to α + 1 do
 4.       for j = 0 to m − 2 do
 5.           M[i, j] = 0
 6.   for i = 0 to m − 2 do C[i] = 0
 7.   for i = 0 to n − 1 do
 8.       j = i mod(α + 2)
 9.       for k = 0 to m − 2 do
10.           C[k] = C[k] − M[j, k]
11.           M[j, k] = 0;
12.       if P[m − 1] =δ T[i] and C[m − 2] > 0 then
13.           output(i)
14.       for k = m − 2 downto 1 do
15.           if P[k] =δ T[i] and C[k − 1] > 0 then
16.               M[j, k] = C[k − 1]
17.               C[k] = C[k] + C[k − 1]
18.       if P[0] =δ T[i] then
19.           M[j, 0] = 1
20.           C[0] = C[0] + 1
```

**Fig. 6.** The $(\delta, \alpha)$-Sequential-Sampling algorithm for the $\delta$-approximate matching problem with $\alpha$-bounded gaps

## 5 Experimental Results

In this section we report experimental data relative to an extensive comparison of the running times of our algorithm $(\delta, \alpha)$-Sequential-Sampling, presented in Section 4, and the algorithm $\delta$-Bounded-Gaps, described in Section 3.

Both algorithms have been implemented in the C programming language and were used to search for the same patterns in large fixed text sequences on a PC with a Pentium IV processor at 2.66GHz. In particular, they have been tested on three Rand$\sigma$ problems, for $\sigma = 60, 120, 180$ and on a real music text buffer.

In particular, each Rand$\sigma$ problem consisted in searching a set of 250 random patterns of length 10, 20, 40, 60, 80, 100, 120, and 140 in a 5Mb random text sequence over a common alphabet of size $\sigma$. For each Rand$\sigma$ problem, the approximation bound $\delta$ and the gap bound $\alpha$ have been set to 1, 2, 4 and to 4, 8, respectively.

Concerning the tests on the real music text buffer, these have been performed on a 4.8Mb file obtained by combining a set of classical pieces, in MIDI format, by C. Debussy. The resulting text buffer has been translated in the pitch interval encoding with an alphabet of 101 symbols. For each $m = 10, 20, 40, 60, 80, 100, 120, 140$, we have randomly selected in the file 250 substrings of length $m$ which subsequently have been searched for in the same file.

All running times in the tables have been expressed in tenths of second and, for each length of the pattern, the best result achieved has been bold-faced. Moreover, $(\delta, \alpha)$-S-S denotes our algorithm $(\delta, \alpha)$-Sequential-Sampling, whereas $\delta$-B-G denotes the algorithm $\delta$-Bounded-Gaps by Crochemore *et al.*

Experimental results with $\sigma = 60$

| $\delta = 1, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
|---|---|---|---|---|---|---|---|---|
| $(\delta, \alpha)$-S-S | **5.042** | **8.203** | **14.26** | **20.41** | **26.38** | **32.36** | **38.43** | **44.47** |
| $\delta$-B-G | 5.724 | 10.75 | 21.13 | 32.03 | 42.80 | 53.84 | 64.87 | 75.77 |
| $\delta = 1, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.199** | **8.511** | **14.56** | **20.69** | **26.57** | **32.53** | **38.80** | **44.85** |
| $\delta$-B-G | 5.660 | 10.52 | 21.28 | 31.69 | 42.55 | 53.19 | 64.20 | 74.36 |
| $\delta = 2, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.336** | **8.580** | **14.73** | **20.86** | **26.72** | **32.73** | **38.84** | **44.85** |
| $\delta$-B-G | 5.832 | 11.00 | 22.67 | 33.34 | 44.80 | 55.99 | 67.29 | 78.56 |
| $\delta = 2, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | 5.914 | **9.395** | **15.41** | **21.54** | **27.46** | **33.48** | **39.68** | **45.68** |
| $\delta$-B-G | **5.827** | 10.98 | 22.67 | 33.36 | 44.79 | 56.03 | 67.12 | 78.44 |
| $\delta = 4, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | 6.347 | **9.717** | **15.75** | **21.87** | **27.84** | **33.94** | **40.01** | **46.03** |
| $\delta$-B-G | **6.258** | 11.76 | 24.02 | 35.79 | 47.41 | 59.89 | 71.23 | 83.55 |
| $\delta = 4, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | 8.135 | 12.67 | **19.14** | **25.36** | **31.20** | **37.34** | **43.54** | **49.67** |
| $\delta$-B-G | **6.259** | **11.99** | 24.18 | 35.75 | 47.78 | 59.82 | 71.33 | 83.35 |

Experimental results with $\sigma = 120$

| $\delta = 1, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
|---|---|---|---|---|---|---|---|---|
| $(\delta, \alpha)$-S-S | **4.916** | **8.044** | **14.10** | **20.26** | **26.17** | **32.21** | **38.25** | **44.37** |
| $\delta$-B-G | 5.529 | 10.37 | 20.48 | 30.49 | 41.06 | 51.37 | 61.80 | 71.87 |
| $\delta = 1, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **4.931** | **8.129** | **14.24** | **20.33** | **26.20** | **32.15** | **38.48** | **44.47** |
| $\delta$-B-G | 5.406 | 10.16 | 21.18 | 30.81 | 41.74 | 51.55 | 62.58 | 72.42 |
| $\delta = 2, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.006** | **8.143** | **14.19** | **20.34** | **26.35** | **32.31** | **38.39** | **44.44** |
| $\delta$-B-G | 5.861 | 10.96 | 21.24 | 31.59 | 42.06 | 52.53 | 63.12 | 73.72 |
| $\delta = 2, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.100** | **8.372** | **14.40** | **20.54** | **26.50** | **32.41** | **38.68** | **44.73** |
| $\delta$-B-G | 5.772 | 10.83 | 21.34 | 31.86 | 42.62 | 53.41 | 63.82 | 74.57 |
| $\delta = 4, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.242** | **8.481** | **14.50** | **20.67** | **26.67** | **32.67** | **38.75** | **44.76** |
| $\delta$-B-G | 5.788 | 10.92 | 22.48 | 33.18 | 44.47 | 55.49 | 66.51 | 77.72 |
| $\delta = 4, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.696** | **9.093** | **15.15** | **21.32** | **27.18** | **33.23** | **39.40** | **45.51** |
| $\delta$-B-G | 5.960 | 11.14 | 21.51 | 32.00 | 42.93 | 53.87 | 64.97 | 75.95 |

Experimental results with $\sigma = 180$

| $\delta = 1, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
|---|---|---|---|---|---|---|---|---|
| $(\delta, \alpha)$-S-S | **4.911** | **8.019** | **14.06** | **20.24** | **26.17** | **32.16** | **38.24** | **44.34** |
| $\delta$-B-G | 5.406 | 10.16 | 21.05 | 30.82 | 41.77 | 51.52 | 62.43 | 72.31 |
| $\delta = 1, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **4.886** | **8.055** | **14.11** | **20.22** | **26.08** | **32.00** | **38.31** | **44.35** |
| $\delta$-B-G | 5.913 | 11.07 | 21.36 | 31.65 | 41.96 | 52.30 | 62.51 | 72.95 |
| $\delta = 2, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **4.941** | **8.057** | **14.09** | **20.26** | **26.16** | **32.17** | **38.21** | **44.29** |
| $\delta$-B-G | 5.605 | 10.47 | 20.72 | 31.07 | 41.68 | 52.04 | 62.69 | 73.13 |
| $\delta = 2, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **4.954** | **8.164** | **14.19** | **20.33** | **26.18** | **32.13** | **38.42** | **44.46** |
| $\delta$-B-G | 5.635 | 10.58 | 21.38 | 31.61 | 42.75 | 53.41 | 64.64 | 75.68 |
| $\delta = 4, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.045** | **8.230** | **14.44** | **20.51** | **26.40** | **32.45** | **38.47** | **44.53** |
| $\delta$-B-G | 5.786 | 10.84 | 21.18 | 31.76 | 42.37 | 53.54 | 65.46 | 76.52 |
| $\delta = 4, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.201** | **8.476** | **14.55** | **20.72** | **26.56** | **32.57** | **38.81** | **44.84** |
| $\delta$-B-G | 5.724 | 10.74 | 21.89 | 32.40 | 43.40 | 54.07 | 65.01 | 75.85 |

Experimental results on a Real Music problem with $\sigma = 101$

| $\delta = 1, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
|---|---|---|---|---|---|---|---|---|
| $(\delta, \alpha)$-S-S | **4.707** | **7.846** | **13.28** | **19.06** | **24.50** | **30.30** | **35.68** | **41.32** |
| $\delta$-B-G | 5.138 | 9.639 | 19.62 | 29.14 | 38.90 | 48.60 | 58.61 | 68.15 |
| $\delta = 1, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **4.885** | **8.404** | **13.57** | **19.62** | **24.84** | **30.70** | **35.92** | **41.61** |
| $\delta$-B-G | 5.121 | 9.579 | 19.80 | 29.15 | 39.10 | 48.58 | 58.74 | 68.20 |
| $\delta = 2, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.063** | **8.779** | **13.57** | **19.92** | **25.13** | **31.12** | **36.12** | **42.13** |
| $\delta$-B-G | 5.246 | 9.823 | 20.39 | 29.96 | 40.48 | 49.99 | 60.77 | 70.26 |
| $\delta = 2, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.306** | **8.531** | **14.11** | **20.50** | **25.60** | **31.66** | **36.85** | **42.60** |
| $\delta$-B-G | 5.351 | 9.985 | 20.04 | 30.25 | 40.23 | 50.07 | 60.42 | 70.47 |
| $\delta = 4, \alpha = 4$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | **5.610** | **9.077** | **14.59** | **21.10** | **26.31** | **32.26** | **37.41** | **43.32** |
| $\delta$-B-G | 5.677 | 10.49 | 21.26 | 31.68 | 42.39 | 52.75 | 63.52 | 73.72 |
| $\delta = 4, \alpha = 8$ | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 |
| $(\delta, \alpha)$-S-S | 5.864 | **9.838** | **15.56** | **23.04** | **27.65** | **34.13** | **39.12** | **45.27** |
| $\delta$-B-G | **5.581** | 10.33 | 21.28 | 31.66 | 42.42 | 52.92 | 63.75 | 73.94 |

Experimental results show that most of the times our newly presented algorithm is faster than the one by Crochemore *et al.* Its superiority is more noticeable as the size of the pattern increases.

## 6 Conclusions

We have presented a new efficient $\mathcal{O}(mn)$-time algorithm for the $\delta$-approximate string matching problem with $\alpha$-bounded gaps. Extensive experimentation has shown that in most of the cases our algorithm is faster than the one by Crochemore *et al.*, which to our knowledge is the only solution present in literature for the same matching problem. The performances of our algorithm become more remarkable as the size of the pattern increases. In addition, our algorithm uses only $\mathcal{O}(m\alpha)$-space, rather than $\mathcal{O}(n)$-space, and it also computes the number of all distinct approximate matchings of the pattern at each position of the text.

We plan to reach a further speed-up of our algorithm by appropriate tuning. We also intend to generalize it to other variants of the approximate matching problem with gaps.

## References

1. E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Perth, WA, Australia, 1999.
2. D. Cantone, S. Cristofaro, and S. Faro. Efficient algorithms for the $\delta$-approximate string matching problem in musical sequences. pages 69–82, Czech Technical University, Prague, Czech Republic, 2004. Proc. of the Prague Stringology Conference '04.
3. T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.
4. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps, 2002.
5. M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In M. Balík and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference'01*, pages 26–36, Prague, Czech Republic, 2001. Annual Report DC–2001–06.
6. M. Crochemore, C. S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three heuristics for $\delta$-matching: $\delta$-BM algorithms. In A. Apostolico and M. Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, number 2373 in Lecture Notes in Computer Science, pages 178–189, Fukuoka, Japan, 2002. Springer-Verlag, Berlin.
7. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 75–86, Hunter Valley, Australia, 2000.
8. J. Karhumäki, W. Plandowski, and W. Rytter. Pattern-matching problems for two-dimensional images described by finite automata. *Nordic J. Comput.*, 7(1):1–13, 2000.
9. S. Karlin, M. Morris, G. Ghandour, and M. Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science*, 85:841–845, 1988.

# The Necessity of Timekeeping in Adversarial Queueing

Maik Weinard

Institut für Informatik,
Johann Wolfgang Goethe–Universität Frankfurt am Main,
Robert-Mayer-Straße 11-15
60054 Frankfurt am Main, Germany
weinard@thi.informatik.uni-frankfurt.de

**Abstract.** We study queueing strategies in the adversarial queueing model. Rather than discussing individual prominent queueing strategies we tackle the issue on a general level and analyze classes of queueing strategies. We introduce the class of queueing strategies that base their preferences on knowledge of the entire graph, the path of the packet and its progress. This restriction only rules out time keeping information like a packet's age or its current waiting time.

We show that all strategies without time stamping have exponential queue sizes, suggesting that time keeping is necessary to obtain subexponential performance bounds. We further introduce a new method to prove stability for strategies without time stamping and show how it can be used to completely characterize a large class of strategies as to their 1-stability and universal stability.

## 1 Introduction

We study the problem of contention resolution for packet routing in networks. A network is represented as a graph with vertices representing the access points of the network (routers) and edges representing the established connections between routers. Users will insert data – organized in packets of roughly same size – into the access points of the network. Each packet has a destination and routing policies assign a simple path from its source to its destination.

This paper focuses on queueing strategies. Queueing strategies are used to decide which packet may proceed whenever more than one packet intends to traverse an edge. Throughout this paper we will concentrate on *greedy* strategies. These are strategies that allow one packet to cross an edge $e$ whenever there is a packet ready to cross edge $e$.

We analyze queueing strategies in a distributed online model, i.e., decisions need to be made on the fly, independent of future input, with local information only, as a router is realistically neither aware of packets that will be inserted into the network in the future nor of packets that are currently stored in other nodes of the network.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 440–451, 2005.
© Springer-Verlag Berlin Heidelberg 2005

We work within the model of adversarial queueing theory [6], a worst-case setup which allows to establish performance guarantees. Packets are inserted by an adversary at arbitrary times and with arbitrary assigned paths. Of course the strength of the adversary needs to be restricted, since no queueing strategy can cope with an adversary, who continuously inserts more packets which seek to cross a specific edge $e$ than edge $e$ can handle.

**Definition 1.** *An adversary is an $(r, b)$-adversary, if for every time interval $I$ and every edge $e$ at most $r \cdot |I| + b$ packets with edge $e$ in their path are inserted into the network during $I$. We call $r$ the rate and $b$ the burstiness.*

Hence for $r \leq 1$ an adversary cannot just simply overload an edge by new insertions. While for some queueing strategies like First-In-First-Out (FIFO) there exist graphs for which an upper bound for the total traffic in a network cannot be guaranteed even for arbitrarily small $r > 0$ [5, 8], others like Nearest-To-Source (NTS) have bounded total traffic in every graph even for $r = 1$ [7].

**Definition 2.** *1. A queueing strategy is $r$-stable, if for every graph $G$ and every $b \in \mathbb{N}$ there exists a bound $c_{G,r,b}$ such that for every sequence of insertions by an $(r, b)$-adversary into $G$ the total number of packets in $G$ never exceeds $c_{G,r,b}$.*

*2. A queueing strategy is universally stable, if it is $r$-stable for every $r < 1$.*

Apart from the number of packets in the system, one is also interested in transportation times, i.e., the time between the insertion of a packet into the network and its arrival at the final node of its path. For $r < 1$ there is a straight forward connection between the number of packets in the system and transportation times [2]: if in a given network and against a given adversary the maximal size of a queue is bounded, so are the number of packets in the entire network and the transportation times. Hence for $r < 1$ we concentrate on analyzing the queue sizes of strategies.

It is crucial to see that universal stability is a necessary but by no means sufficient condition for a strategy in order to be "useful" in the $r < 1$ setup. A transportation time superpolynomial in the number of vertices of the graph is inacceptable if one has networks like the internet in mind and the fact, that this is a constant for a fixed network, offers little comfort.

A randomized protocol with polynomial queue size was introduced in [2]. This protocol however can only be derandomized in a centralized manner: total knowledge of all insertions is necessary. In [3] another randomized strategy, following similar high level ideas, with polynomial delay is introduced and derandomized in a distributed manner: each router can do the necessary computations only with the knowledge of the packets inserted into the network via this router. So also deterministic queueing strategies with polynomial queue size do exist.

The question remains as to whether there exist *simple* queueing strategies that achieve the same goal. Longest-In-System (LIS) is the only *prominent* strategy for which the upper bound of $2^{O(d)}$ could only be matched by a lower bound of $\Omega(d)$, where $d$ is the diameter of the graph. It is known [1], that on directed

acyclic graphs the queue size for LIS is indeed $O(d)$, whereas a proof for the general case requires new techniques [4]. Also in [4] it is shown, that the transportation time under LIS can be exponential in the path length of a packet. (These paths however turn out to be short in comparison to the diameter.) This result is then extended to a class of *vulnerable* queueing strategies.

We also seek results about entire classes of queueing strategies. We associate a queueing strategy $S$ with a priority function that maps the *state of a packet* and available knowledge of the network to a priority. Among packets competing for the same edge, $S$ then prefers a packet of highest priority. Classes arise by specifying which parameters a strategy bases its priority on. Our goal is a study of the large class of *strategies without time stamping*.

**Definition 3.** *We say that a queueing strategy $S_f$ operates* without time stamping *if it assigns priorities $f(G, P, a)$, where $G$ is the graph of the network, $P$ is the path of the packet and $a$ is the number of edges already traversed. We call strategies that operate without time stamping WTS-strategies for short.*

Prominent WTS-strategies include Nearest-To-Source (NTS), Farthest-From-Source (FFS), Nearest-To-Go (NTG), and Farthest-To-Go (FTG) with priority functions $f_{NTS}(G, P, a) = -a$, $f_{FFS}(G, P, a) = a$, $f_{NTG}(G, P, a) = a - |P|$ and $f_{FTG}(G, P, a) = |P| - a$ respectively.

In Section 3 (Theorem 1) we show that each WTS-strategy has queue size $2^{\Omega(\sqrt{n})}$, where $n$ is the number of vertices. Moreover the diameter $d$ turns out to coincide asymptotically with $\sqrt{n}$ and hence the queue size is $2^{\Omega(d)}$.

This result suggests that time keeping is crucial to obtain good performance bounds, as the only *reasonable* quantities that WTS-strategies ignore, are times, such as the age of a packet – as used in LIS – or the current waiting time of a packet – as used in FIFO.

In [9] the term *eternal packet* is introduced for a packet that gets stuck in a network indefinitely. For greedy strategies this effect only arises at the critical arrival rate $r = 1$. Observe that there are strategies that avoid eternal packets at $r = 1$ but are instable for every $r < 1$ like FIFO, while others are even 1-stable but fail to avoid eternal packets at $r = 1$. In fact no strategy can avoid eternal packets *and* be 1-stable [9].

All WTS-strategies produce eternal packets at $r = 1$. For burstiness $b \geq 1$ this observation can be easily verified with a one edge network: in step one insert two packets that seek to traverse the edge and in every later step one more packet. As a $WTS$-strategy is incapable of distinguishing between any of these packets, one can be stuck forever.

In Section 4 we introduce the technique of push-around-cycles that can be used to prove 1-stability for WTS-strategies. Using this technique we provide a complete classification of 1-stable distance-based strategies: these strategies base their decision on the number of edges a packet has already crossed and the length of its path. It turns out that in this class of strategies 1-stability and universal stability coincide. Conclusions and open problems are discussed in Section 5.

## 2 Notation and Conventions

Throughout this paper we let $G$ denote the graph of the network, $\mathcal{P}_G$ the set of all simple paths in $G$ and $n, m, d$ the number of vertices, number of edges and the diameter of the graph in question. We assume that the network operates in consecutive steps, where each step breaks down into three substeps:

1. Insertion: New packets with assigned paths are inserted by the adversary.
2. Transportation: Packets move along edges.
3. Clean Up: Packets that have crossed the last edge of their path are removed from the system.

Hence a packet that is inserted into the system may proceed in the same step. In each step each edge can be crossed by at most one packet. We use $Q_e(t)$ to denote the set of packets ready to cross edge $e$ in step $t$. We occasionally use multiple edges; however, if desired, multiple edges can be eliminated by introducing extra nodes. Whenever two packets of same priority reside in the same queue we assume worst case tie resolution.

## 3 Queue Size of WTS-Strategies

We now see that WTS-strategies cannot avoid exponential queue size and hence exponential transportation time.

**Theorem 1.** *There is a family $G_k$ of graphs with $n = 2k^2 + 6$ nodes and diameter $d = 4k$ so that every WTS-strategy requires queues of size $2^{\Theta(k)}$ for $r > 0.5$ and $b > \frac{2rk}{2r-1}$.*

*Proof.* We describe $G_k = (V_k, E_k)$. $G_k$ basically consists of $k^2$ copies of the gadget $G_{ij}$ (see Figure 1). $G_k$ has $2k^2 + 6$ vertices and diameter $4k$.

Let $R_i$ be the set of paths from $X$ to $Z$ traversing only the gadgets of row $i$. Let $C_j$ be the set of paths from $X'$ to $Z'$ traversing only the gadgets of column $j$. We will only work with these row and column paths.

For each of the $x$-edges $x_{ij}^l$ we determine a *dominant path*: i.e., a path in $R_i \cup C_j$ that uses $x_{ij}^l$ and has a maximum priority in the queue of $x_{ij}^l$. Hence in the queue of $x_{ij}^l$ a packet on the dominant path has priority

$$\max\left\{\max_{P\in R_i}\{f(G_k, P, 2j) | P \text{ uses } x_{ij}^l\}, \max_{P\in C_j}\{f(G_k, P, 2i) | P \text{ uses } x_{ij}^l\}\right\}.$$

Observe that a packet on a path in $R_i$ [$C_j$] has traversed $2j$ edges [$2i$ edges] when reaching $x_{ij}^l$. If a dominant path of $x_{ij}^l$ is in $R_i$ we say that edge $x_{ij}^l$ is *row dominated*, otherwise it is *column dominated*. Furthermore we say that gadget $G_{i,j}$ is row [column] dominated if at least $k + 1$ of its $x$-edges are row [column] dominated. Hence each gadget is either row or column dominated.

We now focus on a row in which at least half of the gadgets are column dominated or a column in which at least half of the gadgets are row dominated.

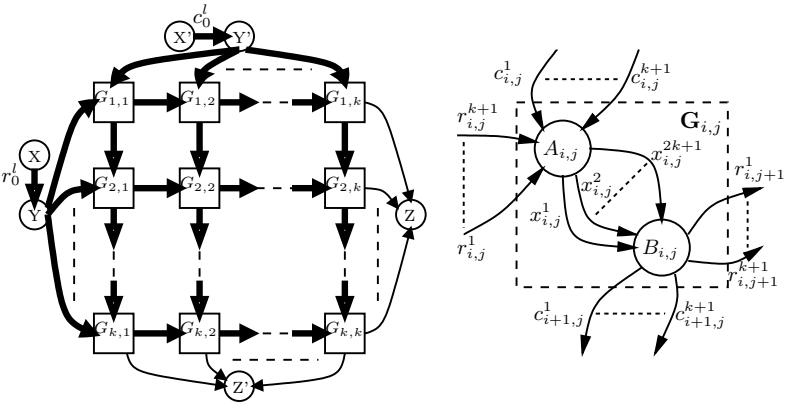



**Fig. 1.** Graph $G_k$ contains $k^2$ gadgets $G_{i,j}$ and extra nodes $X, Y, Z, X', Y'$ and $Z'$ arranged and connected as indicated on the left. A thick arrow represents a set of $k+1$ multiple edges. Each gadget consists of two internal nodes $A_{i,j}$, where all incoming edges end, and $B_{i,j}$, where all outgoing edges originate, connected by $2k+1$ directed edges named $x_{i,j}^1, \ldots, x_{i,j}^{2k+1}$. The incoming row edges (coming from $G_{i,j-1}$ resp. $Y$) are labeled $r_{i,j}^l$ for $1 \leq l \leq k+1$. The incoming column edges (coming from $G_{i-1,j}$ resp. $Y'$) are labeled $c_{i,j}^l$ for $1 \leq l \leq k+1$. The row edges [column edges] leaving $G_{i,k}$ [$G_{k,j}$] and entering $Z$ [$Z'$] are named $r_{i,k+1}$ [$c_{k+1,j}$]. Finally we call the $k+1$ edges connecting $X$ and $Y$ [$X'$ and $Y'$] $r_0^l$ [$c_0^l$] for $1 \leq l \leq k+1$

Observe that at least one such row or column must exist. W.l.o.g. asssume that there are $q \geq \frac{k}{2}$ column dominated gadgets in row $i_0$ and that in column dominated gadgets $G_{i_0,j}$ edges $x_{i_0,j}^1, \ldots, x_{i_0,j}^{k+1}$ are column dominated. The following algorithm carefully chooses paths from $R_{i_0}$ that we will use to insert packets.

1. Initially let $L$ consist of the edge $r_{i_0,k+1}$, the edges $r_0^l$ and $r_{i_0,j}^l$ as well as $x_{i_0,j}^l$ for $1 \leq j \leq k$ and $1 \leq l \leq k+1$. We call edges in $L$ *legal* and call a path legal, if it only uses legal edges. Set $z := q$.
2. For $j$ from $k$ to 1 in descending order repeat:
   (a) Choose $e \in \{r_{i_0,j}^l | 1 \leq l \leq k+1\} \cap L$ arbitrarily (a legal entrance to $G_{i_0,j}$).
   (b) IF $G_{i_0,j}$ is column dominated:
      i. Let $S_z$ be a legal path that uses $e$ and assigns a minimum priority (restricted to legal paths using $e$) in $Q_e$. (I.e., $f(G_k, S_z, 2j-1)$ is minimal). Remove the edges $S_z$ traverses before $e$ from $L$.
      ii. Let $e' \in \{x_{i_0,j}^l | 1 \leq l \leq k+1\}$ be a legal edge $S_z$ does not use.
      iii. Let $D_z$ be the dominant path of $e'$. Set $z := z-1$.
   (c) ELSE: Choose $e' \in \{x_{i_0,j}^l | 1 \leq l \leq k+1\} \cap L$ arbitrarily.
   (d) Remove all edges $r_{i_0,j}^l \neq e$ and $x_{i_0,j}^l \neq e'$ with $1 \leq l \leq k+1$ from $L$.
3. Choose a legal path $S_0$ arbitrarily.

Observe that the choices of $e$ and $e'$ are always well defined, since at start there are for each $j$ at least $k+1$ legal $r_{i_0,j}^l$ and $x_{i_0,j}^l$ edges. At most $k-1$ of them are removed in steps 2(b)i before $e$ and $e'$ are picked.

We are now ready to start our insertion of packets. The insertion scheme proceeds in $q$ phases that correspond to the column dominated gadgets. Let these be $G_{i_0,j_1}, G_{i_0,j_2}, \ldots G_{i_0,j_q}$. To start off the process we exploit burstiness and launch $b$ packets along path $S_0$ in one step. We call this set of packets $X_0$.

**Phase t:** Phase $t$ starts, when the first packet of $X_{t-1}$ is ready to cross $r^l_{i_0,j_t}$ and lasts for $|X_{t-1}|$ steps. During phase $t$ we insert packets along $S_t$ and $D_t$ in parallel at rate $r$. Note that $S_t$ and $D_t$ are edge disjoint.

In the queue of $r^l_{i_0,j_t}$ the packets from $S_t$ collide with the packets of $X_{t-1}$ for the first time. They have a priority not greater than any packet in $X_{t-1}$ as the paths, the packets of $X_{t-1}$ are travelling on, were all legal when $S_t$ was picked minimally. Hence none of the $r|X_{t-1}|$ packets from $S_t$ is able to traverse $G_{i_0,j_t}$ in phase $t$.

After at most $2k$ steps of phase $t$ the first packet on $D_t$ arrives in $G_{i_0,j_t}$ and from then on the $x^l_{i_0,j_t}$ edge, the packets from $X_{t-1}$ intend to use, is occupied $r \cdot (|X_{t-1}| - 2k)$ steps by blocking packets. As $D_t$ is a dominant path of the edge at least $r \cdot (|X_{t-1}| - 2k)$ packets from $X_{t-1}$ do not traverse $G_{i_0,j_t}$ in phase $t$.

Let $X_t$ be the union of these remaining $X_{t-1}$ packets and the newly inserted packets from $S_t$. Then $|X_t| \geq 2r|X_{t-1}| - 2rk$ holds. Observe that for $r > \frac{1}{2}$ and sufficiently large $b$ (i.e., $|X_0| = b > \frac{2rk}{2r-1}$) the size of $X_t$ has increased by a multiplicative factor. As $q = \Theta(k) = \Theta(d)$, the last set $X_q$ has size $2^{\Theta(d)}$. □

## 4 Stability of WTS-Strategies

In this section we provide a new method for proving 1-stability of WTS strategies, that can be used to unify the proofs for NTS and FTG as well as for proving the 1-stability of entire classes of strategies. Besides providing a sufficient criterion for 1-stability of WTS-strategies we also characterize universally stable distance-based strategies.

Crucial for this approach is the concept of push-around-cycles. A push-around-cycle is intuitively speaking a sequence of paths that intersect in a cyclic manner so that the priorities allow to push packets around this cycle indefinitely. Assume WTS-strategy $S_f$ is used. We show that whenever the number of packets in a network $G$ can be driven beyond any bound (instability) then $G$ contains a push-around-cycle with respect to $f$. By contraposition we may then conclude 1-stability whenever a queueing strategy prevents push-around-cycles in every single graph. We introduce the concept of a path prefix.

**Definition 4.** *Let $P = (e_1, e_2, \ldots, e_z)$ be a path. Then the* path prefix $[P, a]$ *for $a \leq z$ consists of the edges $(e_1, e_2, \ldots, e_a)$.*

Let $Q^P_e(t)$ denote the set of packets in $Q_e(t)$ travelling along path $P$ and $Q^P(t)$ be the set of packets travelling along path $P$ at time $t$. (Hence $Q^P(t) = \bigcup_{i=1}^{z} Q^P_{e_i}(t)$.) Furthermore set $Q^{[P,a]}(t) = \bigcup_{i=1}^{a} Q^P_{e_i}(t)$ as the set of packets in path prefix $[P, a]$.

WTS-strategies define priorities between paths $P$ and $P'$ meeting in a common edge $e$: a packet reaching edge $e$ on path $P$ has the right of way over a

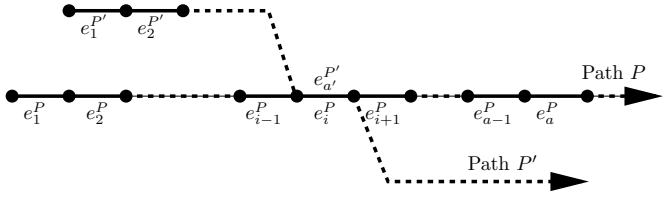



**Fig. 2.** An illustration for Lemma 1. The common edge $e_{a'}^{P'} = e_i^P$ is distinguished

packet reaching $e$ on path $P'$ if $f(G, P, j) > f(G, P', i)$ where $e$ is the $j$-th (resp. $i$-th) edge of $P$ (resp. $P'$). The Lemma 1 shows that for every heavily loaded path prefix $[P, a]$ there is another heavily loaded path prefix $[P', a']$ that has priority over $P$ in an edge $e$. Later we apply this method repeatedly to find a necessary condition for instability at every $r \leq 1$. Figure 2 illustrates Lemma 1.

**Lemma 1.** *Assume WTS-strategy $S_f$ (with priority function $f$) is used on a graph $G$. Moreover assume that after a sequence of insertions for $t$ steps by an $(1, b)$ adversary there is a path prefix $[P, a]$ such that $|Q^{[P,a]}(t)| \geq c$, where $c$ is a constant larger than $b$. Then there exists a path prefix $[P', a']$ such that*

- *$P'$ has the right of way over $P$ in a common edge $e = e_i^P = e_{a'}^{P'}$ and*
- *the path prefix of $P'$ ending in $e$ has at some moment in time before time $t$ carried at least $\frac{c}{2 \cdot 3^d \cdot |\mathcal{P}_\mathcal{G}|} - \frac{b}{|\mathcal{P}_\mathcal{G}|}$ packets.*

*Proof.* Assume that $|Q^{[P,a]}(t)| \geq c$ holds. Then choose $t_0$ minimal such that at time $t_0$ there exists $i \leq a$ such that

$$|Q_{e_i}^P(t_0)| > \frac{c}{3^{a-i+1}}. \tag{1}$$

Such a $t_0 \leq t$ is well defined, since otherwise we have $|Q_{e_j}^P(t)| \leq \frac{c}{3^{a-j+1}}$ for all $j \leq a$ and consequently

$$\begin{aligned} |Q^{[P,a]}(t)| &= \sum_{j=1}^{a} |Q_{e_j}^P(t)| \leq \sum_{j=1}^{a} \frac{c}{3^{a-j+1}} \\ &= c \cdot 3^{-a} \cdot \sum_{j=1}^{a} 3^{j-1} = c \cdot 3^{-a} \frac{3^a - 1}{2} < \frac{c}{2}, \text{ a contradiction.} \end{aligned}$$

Choose $i$ to be a minimal index satisfying inequality (1). Furthermore pick $t_1$ maximally with $t_1 < t_0$ such that $Q_{e_i}^P(t_1) = \emptyset$. $t_1$ exists, since all queues are assumed to be empty in the beginning. Exploiting $t_1 < t_0$ and the minimality of $t_0$ we get $|Q^{[P,i-1]}(t_1)| = \sum_{j=1}^{i-1} |Q_{e_j}^P(t_1)| \leq \sum_{j=1}^{i-1} \frac{c}{3^{a-j+1}} \leq \frac{c}{2 \cdot 3^{a-i+1}}$.

Since $|Q_{e_i}^P(t_0)| > \frac{c}{3^{a-i+1}}$, at most half of the packets in $Q_{e_i}^P(t_0)$ were already in the system at time $t_1$. We concentrate on the time interval $J = (t_1, t_0]$ and

assume that $y$ packets are inserted into path $P$ during $J$. Let $x$ denote the number of packets on path $P$ that traverse $e_i$ during $J$. We then conclude

$$y - x \geq \frac{c}{2 \cdot 3^{a-i+1}}. \tag{2}$$

Since $Q_{e_i}^P(t)$ is nonempty during $J$, one packet traverses $e_i$ in every step of $J$, and thus $t_0 - t_1 - 1 = |J|$ packets traverse $e_i$. Hence $(t_0 - t_1 - 1 - x)$ packets from paths that have priority over $P$ in $e_i$ traverse edge $e_i$ during $J$. Due to the restriction on the adversary at most $(t_0 - t_1 - 1 + b)$ packets with $e_i$ are inserted during $J$ and at most $(t_0 - t_1 - 1 + b - y)$ of them travel on paths other than $P$. We may hence conclude – using (2) –, that at least

$$(t_0 - t_1 - 1 - x) - (t_0 - t_1 - 1 + b - y) = y - x - b \geq \frac{c}{2 \cdot 3^{a-i+1}} - b \geq \frac{c}{2 \cdot 3^d} - b$$

packets on paths with priority over $P$ in $e_i$ are in the system at time $t_1$ and that they are somewhere before or on edge $e_i$ on their respective paths.

Finally we observe that at least one path with priority over $P$ in $e_i$ must carry at least $\frac{c}{2 \cdot 3^d \cdot |\mathcal{P}_G|} - \frac{b}{|\mathcal{P}_G|}$ of these packets before or on $e_i$ and hence if $P'$ is this path and $a'$ is picked so that $e_{a'}^{P'} = e_i$ we have verified the claim. □

A repeated application of Lemma 1 leads to the concept of push-around-cycles.

**Definition 5.** *A push-around-cycle with respect to a WTS-strategy with priority function $f$ and a network $G$ is a sequence of paths $P_1, P_2, \ldots, P_r \in \mathcal{P}_G$ in $G$ with two distinguished edges $e_{x_i}^{P_i}$ and $e_{y_i}^{P_i}$ for every path with the following properties:*

- $\forall_{1 \leq i \leq r}\ x_i \leq y_i$*: edge* $e_{x_i}^{P_i}$ *precedes edge* $e_{y_i}^{P_i}$ *on* $P_i$*,*
- $\forall_{1 \leq i \leq r-1}\ e_{y_i}^{P_i} = e_{x_{i+1}}^{P_{i+1}}$ *and* $e_{y_r}^{P_r} = e_{x_1}^{P_1}$*: The second distinguished edge of* $P_i$ *is the first distinguished edge of* $P_{i+1}$ *and the second distinguished edge of the last path is the first distinguished edge of the first path.*
- $\forall_{1 \leq i \leq r-1}\ f(G, P_i, y_i) \geq f(G, P_{i+1}, x_{i+1})$ *and* $f(G, P_r, y_r) \geq f(G, P_1, x_1)$*: path* $P_i$ *has the right of way over* $P_{i+1}$ *with respect to their common distinguished edge. Moreover* $P_r$ *has the right of way over* $P_1$ *with respect to their common edge.*

Figure 3 illustrates the concept of a push-around-cycle. We are now ready to state the main result of this section.

**Theorem 2.** *If a WTS-strategy $S_f$ with priority function $f$ is instable at $r = 1$ on a graph $G$, then there exists a push-around-cycle among the paths of $G$ with respect to $S_f$.*

*Proof.* We define the following recurence in order to use Lemma 1: $A(0) = b$ and $A(i) = \frac{A(i+1)}{2 \cdot 3^d \cdot |\mathcal{P}_G|} - \frac{b}{|\mathcal{P}_G|}$. As we assume instability, there is a sequence of insertions that causes $G$ to accomodate $|\mathcal{P}_G| \cdot A(d \cdot |\mathcal{P}_G|)$ packets. Hence one path accomodates at least $A(d \cdot |\mathcal{P}_G|)$ packets. Using this entire path $P_0$ and its entire length $a_0 := |P_0|$ as a first path prefix $[P_0, a_0]$ we apply Lemma 1

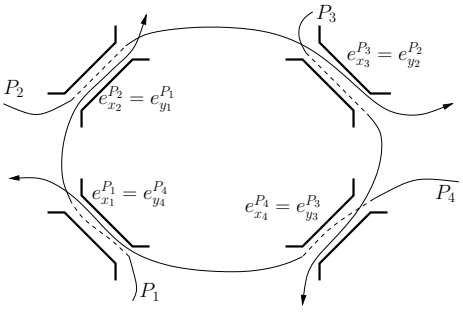



**Fig. 3.** An illustration for a push-around-cycle consisting of four paths and their four distinguished edges. A dashed line indicates that the respective path must yield priority

iteratively: if we have a path prefix $[P_i, a_i]$ that can be forced to accomodate $A(d \cdot |\mathcal{P}_G| - i)$ packets, Lemma 1 provides a path $P_{i+1}$ that intersects $[P_i, a_i]$ and has priority at that intersection. If we pick $a_{i+1}$ according to Lemma 1, we know that the path prefix $[P_{i+1}, a_{i+1}]$ can be forced to contain $A(d \cdot |\mathcal{P}_G| - (i+1))$ packets.

As we have picked $A(d \cdot |\mathcal{P}_G|)$ large enough, we can iterate the application of Lemma 1 $d \cdot |\mathcal{P}_G|$ times, which is an upper bound on the number of path prefixes in $G$. Hence a cycle must be closed in the process and we have our theorem. □

By contraposition we get immediately that every WTS-strategy, that does not allow push-around-cycles in a network $G$, is 1-stable in $G$. The following corollary contains Nearest-To-Source and Farthest-To-Go as special cases.

**Corollary 1.** *Assume a WTS-strategy $S_f$ with priority function $f$ is given. If $f$ is strictly decreasing along all paths $P$ in all graphs $G$, i.e., $f(G, P, i) > f(G, P, i+1)$, then $S_f$ is 1-stable.*

*Proof.* Assume $G$ contains a push-around-cycle with respect to $S_f$, let $P_1, \ldots, P_r$, $x_1, \ldots, x_r$ and $y_1, \ldots, y_r$ be defined in accordance to Definition 5. We then have $f(G, P_1, x_1) > f(G, P_1, y_1) \geq f(G, P_2, x_2) > \ldots > f(G, P_r, y_r) \geq f(G, P_1, x_1)$ as a contradiction. Hence by Theorem 2 the strategy is stable at $r = 1$. □

*Remark 1.* Assume that we have a WTS-strategy that does not depend on the number of edges a packet has traversed so far. If the strategy assigns different priorities to different paths in every $G$, then this strategy is 1-stable.

We say that a queueing strategy is distance-based, if its priority function only depends on the the number $x$ of traversed edges and the length $y$ of the packet's path. We provide a complete classification of 1-stable distance-based strategies.

**Theorem 3.** *Let $f$ be the priority function of a distance-based queueing strategy $S_f$. Then $S_f$ is 1-stable if and only if*

$$\forall (x, y)\ 1 \leq x < y\ :\ f(x, y) < f(x-1, y). \tag{3}$$

*It is not even universally stable otherwise.*

*Proof.* The 1-stability given (3) follows immediatly from Corollary 1, since $f$ is strictly decreasing along all paths. To complete the proof we need to carefully embed and adapt the *baseball graph* from [2] and come up with an elaborate insertion scheme.

Our proof exploits that $S_f$ is only stable if it guarantees stability for every possible choice among packets of same priority. Hence if two packets with identical $x$ and $y$ values collide, we may pick the one to be prefered in a worst case manner. We consider the network of Figure 4.

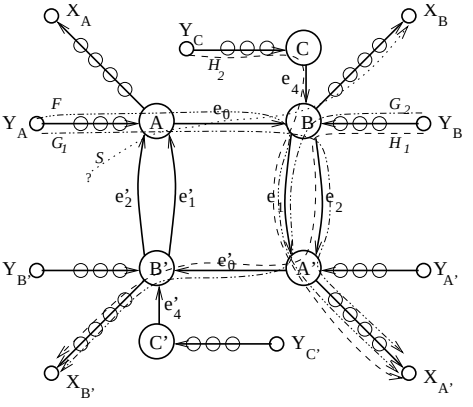



**Fig. 4. The graph used to prove instability of $\mathbf{A_f}$.** We have *central* nodes $A, B, C$ and $A', B', C'$. Each central node has an *entrance path* starting at a node named $Y_A, \ldots, Y_{C'}$. Each of these entrance paths consists of $x-1$ nodes and $x-1$ edges. Hence for $x = 1$ these entrance paths disappear and the $Y$-node is identical with the corresponding central node. Also $A, B, A'$ and $B'$ have an *exit path* ending in a node named $X_A, \ldots, X_{B'}$. Each exit path has length $y - x - 1$. So for $y = x + 1$ these paths vanish and the $X$-nodes are identical with their respective central node

We assume that there exists a time $t$ such that if injection of new packets is completely stopped after step $t$ , there is still – for each of the next $s$ steps – a packet crossing $e_0$. These packets (we call them set $S$) will have crossed precisely $x$ edges before $e_0$, have destination $X_B$ and a total path of length $y$.

We intend to construct an injection process (Phases 1,2 and 3), such that there is a set of more than $s$ packets waiting to cross edge $e'_0$, if injection is stopped after step $t' > t$. These packets will have crossed precisely $x$ edges before $e'_0$, have destination $X_{B'}$ and a total path of length $y$. This implies instability, since the process can then be repeated arbitrarily.

In the caption of Figure 4 we have introduced the notion of central node, entrance and exit path. All the packets that we are inserting in phases 1,2 and 3 have a path of length $y$. In the first central node of their paths they have crossed $x-1$ edges, in the second central node they have crossed $x$ edges. The necessary insertions are listed in the following table along with references to the relevant observations.

| Phase | Duration | Inserted Sets & Paths | Size | Observations |
|---|---|---|---|---|
| 1 | $s$ | $F : (Y_A \xrightarrow{*} A \xrightarrow{e_0} B \xrightarrow{e_1} A' \xrightarrow{*} X_{A'})$ | $rs$ | (1),(2),(3),(4) |
| 2 | $rs$ | $G_1 : (Y_A \xrightarrow{*} A \xrightarrow{e_0} B \xrightarrow{e_2} A' \xrightarrow{*} X_{A'})$ | $r^2s$ | (2),(4),(5) |
| | | $G_2 : (Y_B \xrightarrow{*} B \xrightarrow{e_1} A' \xrightarrow{e_0'} B' \xrightarrow{*} X_{B'})$ | $r^2s$ | (3),(4),(6) |
| 3 | $r^2s$ | $H_1 : (Y_B \xrightarrow{*} B \xrightarrow{e_2} A \xrightarrow{e_0'} B' \xrightarrow{*} X_{B'})$ | $r^3s$ | (5) |
| | | $H_2 : (Y_C \xrightarrow{*} C \xrightarrow{e_4} B \xrightarrow{e_1} A' \xrightarrow{*} X_{A'})$ | $r^3s$ | (6) |

**(1)** In phase 1 the set $S$ of $s$ packets we assume to cross $e_0$ complete their journey. When $F$ reaches $A$, it will be blocked by $S$, since $f(x, y) \geq f(x-1, y)$ holds. Observe that $x-1$ steps pass before $F$'s first packet reaches $A$. So in fact not all of the $rs$ packets of $F$ can get to $A$ before phase 1 ends, i.e., the last one has just been inserted and needs another $x-1$ steps to get to $A$. For $s$ large enough however this effect causes no problems.

After phase 1 $S$ has vanished. (The last packets of $S$ are actually still on their way from $B$ to $X_B$, but they do not interfere with anything we are about to do and we will consider them gone. The same argument in the next steps allows us to regard every packet as *out of the way* once it's on its exit path.)

**(2)** Set $G_1$ collides with $F$ in node $A$. Their path length and advance in the path at this point are identical, hence we may choose that $F$ is advanced over $e_0$.

**(3)** Set $G_2$ collides with $F$ in node $B$. Since $F$ has traversed $x$ edges and $G_2$ has traversed $x-1$ edges, $F$ will be prefered.

**(4)** At the end of phase 2 $F$ has vanished and both $G_1$ and $G_2$ are still in the first central node of their paths.

**(5)** $G_1$ completes its journey in phase 3. In node $B$ the set $G_1$ will block $H_1$.

**(6)** In the first $x$ steps of Phase 3 $x$ packets of $G_2$ cross $e_1$ and are lost for our purpose. After these $x$ steps the stream of $H_2$ packets has reached $B$. In $B$ packets from $H_2$ will be prefered over $G_2$-packets. Of course, as $r < 1$ holds, the stream of $H_2$ packets occupies $e_1$ for $r \cdot (r^2s - x)$ of the remaining $r^2s - x$ steps of Phase 3. Hence $(1-r) \cdot (r^2s - x)$ more $G_2$ packets slip through and are lost.

We define the end of phase 3 as the time $t'$. Observe that $r(r^2s - x)$ packets from $G_2$ and the entire $H_1$ still need to cross $e_0'$. Their combined size is $|S'| = 2r^3s - rx$. So by having $S$ and $r$ sufficiently large we can guarantee, that $|S'| > |S|$ holds and we have successfully increased the number of packets in the system. To start out the process we have to use a sufficiently large burstiness. □

## 5 Conclusion and Open Problems

We have introduced the class of WTS-strategies and obtained general results about this class itself and its subclass of distance-based strategies. Most importantly we have ruled out the existence of WTS-strategies with subexponential queue size indicating, that some form of timekeeping is necessary to achieve polynomial queue size. Furthermore we have introduced the concept of push-around-

cycles and used it to classify the 1-stable and universally stable distance-based strategies.

In order to be of practical use, queueing strategies need to be as simple as possible. So the question remains whether queueing strategies simpler than the one introduced in [3] and with polynomial queue size exist. Such a candidate is Longest-In-System, as LIS is obviously one of the simplest queueing strategies that does use timekeeping.

## Acknowledgements

I would like to thank Gregor Gramlich, Matthias Poloczek and Georg Schnitger, as well as an unknown referee for many helpfull comments.

## References

1. Adler, Micah and Rosen, Adi, Tight Bounds for the Performance of Longest in System on DAGs, *Proc. of the 19th Symposium on Theoretical Aspects of Computer Science, 2002*, pp. 88-99
2. Andrews, M., Awerbuch, B., Fernández, A., Leighton, T., and Liu, Z., Universal-Stability Results and Performance Bounds for Greedy Contention-Resolution Protocols, *Journal of the ACM, Vol. 48, No 1, January 2001*, pp. 39-69
3. Andrews, M., Fernández, A., Goel, A., Zhang, L., Source Routing and Scheduling in Packet Networks, *Proc. of the 42nd Symposium on Foundations of Computer Science, 2001*, pp. 168-177
4. Andrews, M., Zhang, L., The Effects of Temporary Sessions on Network Performance, *SIAM Journal of Computation, Vol. 33, No 3*, pp. 659-673
5. Bhattacharjee, R. and Goel, A., Instability of FIFO at arbitrarily low rates in the adversarial queueing model, *Proc. of the 44th Symposium on Foundations of Computer Science, 2003*, pp. 160-167
6. Borodin, A., Kleinberg, J., Raghavan, P., Sudan, M., and Williamson, D. P. Adversarial queueing theory, *Journal of the ACM, Vol. 48, No 1, January 2001*, pp. 13-38
7. Gamarnik, David, Stability od Adaptive and Non-Adaptive Packet Routing Policies in Adversarial Queueing Networks, *SIAM Journal on Computing, Vol. 32, No 2, 2003*, pp. 371-385
8. Koukopoulos, D., Mavronicolas, M., Spirakis, P., FIFO is Unstable at Arbitrarily Low Rates (Even in Planar Networks), *Electronic Colloq. on Computational Complexity, 2003*
9. Rosén, Adi and Tsirkin, Michael S., On Delivery Times in Packet Networks under Adversarial Traffic, *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures, 2004*, pp. 1-10

# BDDs in a Branch and Cut Framework*

Bernd Becker[1], Markus Behle[2], Friedrich Eisenbrand[2], and Ralf Wimmer[1]

[1] Albert-Ludwigs-Universität, Georges-Köhler-Allee 51,
79110 Freiburg im Breisgau, Germany
{becker,wimmer}@informatik.uni-freiburg.de
[2] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
{behle,eisen}@mpi-sb.mpg.de

**Abstract.** Branch & Cut is today's state-of-the-art method to solve $0/1$-integer linear programs. Important for the success of this method is the generation of strong valid inequalities, which tighten the linear programming relaxation of $0/1$-IPs and thus allow for early pruning of parts of the search tree.

In this paper we present a novel approach to generate valid inequalities for $0/1$-IPs which is based on *Binary Decision Diagrams* (BDDs). BDDs are a data-structure which represents $0/1$-vectors as paths of a certain acyclic graph. They have been successfully applied in computational logic, hardware verification and synthesis.

We implemented our BDD cutting plane generator in a branch-and-cut framework and tested it on several instances of the *MAX-ONES* problem and randomly generated $0/1$-IPs. Our computational results show that we have developed competitive code for these problems, on which state-of-the-art MIP-solvers fall short.

## 1 Introduction

Many industrial optimization problems can be formulated as an integer program. Formally, an integer program deals with the maximization of a linear objective function $c(1)x(1) + \cdots + c(n)x(n)$, where the variables $x(1), \ldots, x(n)$ have to be integers and have to satisfy $m$ given linear inequalities $a_{i1}x(1) + \cdots a_{in}x(n) \leq b_i$ for $1 \leq i \leq m$. A special case of integer programming is *0/1 integer programming* ($0/1$-IP), which arises if the variables are additionally restricted to attain values in $\{0, 1\}$. It is a particularly important special case, since most combinatorial optimization problems are modeled with decision variables and thus are $0/1$-IPs.

The most successful method for $0/1$-IP, which is applied by all competitive commercial codes is *branch-and-cut*. This variant of branch-and-bound relies on the fact that the *linear relaxation* of a given $0/1$-IP can be efficiently solved. The linear relaxation is the linear program which is obtained from the $0/1$-IP by *relaxing* the condition $x(i) \in \{0, 1\}$ to the condition $0 \leq x(i) \leq 1$ for each $i \in \{1, \ldots, n\}$. The value of

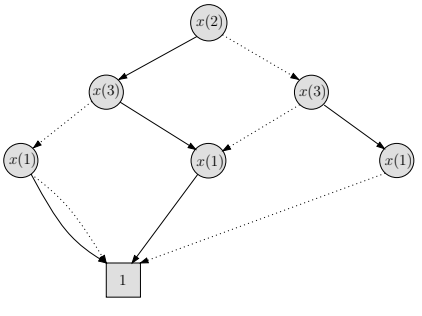
* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 452–463, 2005.
© Springer-Verlag Berlin Heidelberg 2005

the linear programming relaxation can then be used as an upper bound in a branch-and-bound approach to solve the $0/1$-IP. In branch-and-cut, one additionally applies *cutting planes* [8, 26] to improve the quality of the linear programming relaxation. Cutting planes are inequalities which are valid for all feasible integer points, but not necessarily valid for the rational points which are feasible for the linear programming relaxation. Thus the incorporation of cutting planes improves the *tightness* of the linear relaxation and helps to prune parts of the branch-and-bound tree.

In theory a cutting plane can be easily infered from a fractional optimal solution to the linear programming relaxation. The *strength* of the cutting plane is however crucial for the performance of the branch-and-cut process. Classes of strong valid inequalities are for example *knapsack-cover inequalities* [2, 6, 11, 29], *clique inequalities* [21] the *flow-cover inequalities* [23, 24] or the *mixed integer rounding* cuts [21]. Knapsack-cover and flow-cover inequalities in particular are inequalities which are valid for the $0/1$-points which satisfy *one single* constraint of the $0/1$-IP. Up to now there is no satisfactory method available which generates valid inequalities for the $0/1$-solutions of two or more inequalities. This paper aims at a method for this algorithmic problem which is based on *Binary Decision Diagrams*, a datastructure which is widely used in computational logic, hardware verification and logic synthesis.

A Binary Decision Diagram (*BDD*) represents a set of $0/1$-vectors in a compact way, see Fig. 1. We provide a short definition of BDDs as they are used in this paper. A BDD for a set of variables $x(1), \cdots, x(n)$ is a directed acyclic graph $G = (V, A)$ with a labeling $\ell : V \longrightarrow \{x(1), \ldots, x(n)\}$ and a parity function $\mathrm{par} : A \longrightarrow \{0, 1\}$. The graph has one node with in-degree zero, called the *root* and one node with out-degree zero, called *leaf 1*. Each path from $\mathrm{root}$ to $\mathrm{leaf}\ 1$ contains exactly $n$ edges and each $x(i)$, $1 \leq i \leq n$ is the label of a starting node of an edge on this path, thus the BDD is called *complete*. All nodes labelled with $x(i)$ lie on the same level, which means, we have an *ordered* BDD (OBDD). A path $e_1, \ldots, e_n$ from the root to the leaf represents a variable assignment, where the label of the starting node of $e_i$ is assigned to the value $\mathrm{par}(e_i)$. In this way, the BDD *represents* a set of vectors in $\{0, 1\}^n$.




(a) BDD

| $x(1)$ | $x(2)$ | $x(3)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

(b) Represented $0/1$-points

**Fig. 1.** A simple BDD represented as a directed graph. Edges with parity 0 are dashed

BDDs were first proposed by Lee in 1959 [20]. Bryant [3] presented efficient algorithms for the synthesis of BDDs. After that, BDDs became very popular in the area of hardware verification, and computational logics, see e.g. [28]. Lai et. al. [19, 18] have developed a branch-and-bound algorithm for $0/1$-IP that uses an extension of BDDs called *EVBDDs*. EVBDDs represent functions $f : \{0,1\}^n \longrightarrow \mathbb{Z}$. So the EVBDDs are used not only to represent the characteristic functions of the constraints but also for the constraints themselves. In this approach however, one has to build an EVBDD for the conjunction of *all* the constraints of the $0/1$-IP. In many cases this leads to an explosion in memory requirement.

We incorporate BDDs into a cutting plane engine and apply it in an integer programming solver. We use BDDs to represent the feasible solutions of a small *subset* of the given constraints and derive valid inequalities for the polytope which is described by these solutions. Thereby we avoid the explosion of the size of the BDD which happens if the BDD represents all the constraints. The *separation problem* is solved with a sequence of shortest path problems with Lagrangean relaxation techniques. For this we use a standard BDD-package and apply our own efficient implementation of an acyclic shortest path algorithm on the BDD-datastructure. We apply our cutting plane framework to the *MAX-ONES* problem and to *randomly generated* $0/1$-IPs. Our computational results show that we could develop competitive code to solve hard $0/1$-integer programming problems, on which state-of-the-art commercial branch-and-cut codes fall short.

Currently there is active and promising research in the field of combining techniques from computational logic and constraint programming with integer programming, see e.g. [5, 13]. We contribute further to this development by using BDDs sucessfully and for the first time in a cutting plane engine.

### 1.1 Preliminaries from Polyhedral Theory

Before we proceed we review some terminology from polyhedral theory, see e.g. [21, 26]. A *polyhedron* $P$ is a set of vectors of the form $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$, for some matrix $A \in \mathbb{R}^{m \times n}$ and some vector $b \in \mathbb{R}^m$. The polyhedron is *rational* if both $A$ and $b$ can be chosen to be rational. If $P$ is bounded, then $P$ is called a *polytope*. An *integral* $0/1$*-polytope* is a polytope that is the convex hull of a set of $0/1$-vectors $S \subseteq \{0,1\}^n$. The *integer hull* $P_{\mathrm{I}}$ of a polytope $P$ is the convex hull of the integral vectors in $P$. The *dimension* $\dim(P)$ of $P$ is the dimension of its affine hull and $P \subseteq \mathbb{R}^n$ is *full-dimensional* if $\dim(P) = n$.

An inequality $c^T x \leq \delta$ is *valid* for $P$ if it is satisfied by all points in $P$. If $c^T x \leq \delta$ is valid and $\delta = \max\{c^T x \mid x \in P\}$, it defines a *face* $F = \{x \in P \mid c^T x = \delta\}$ of $P$. The face $F$ is a *facet* of $P$, if $\dim(F) = \dim(P) - 1$.

## 2 Using BDDs to Generate Cutting Planes

Suppose we have to solve a $0/1$-integer programming problem,

$$\max\{c^T x : Ax \leq b,\ x \in \{0,1\}^n\} \tag{1}$$

where $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$. Our idea is now to choose a subset $A'x \leq b'$ of the constraints in $Ax \leq b$ and to build the BDD which represents all $0/1$-points which satisfy $A'x \leq b'$. We next distinguish between two $0/1$-polytopes $P_{\mathrm{I}}$ and $P_{\mathrm{BDD}}$. The polytope $P_{\mathrm{I}} = \mathrm{conv}\{x \in \{0,1\}^n \mid Ax \leq b\}$ is the convex hull of the feasible $0/1$-points of the $0/1$-IP. The polytope $P_{\mathrm{BDD}} = \mathrm{conv}\{x \in \{0,1\}^n \mid A'x \leq b'\}$ is the convex hull of the $0/1$-points which are feasible for $A'x \leq b'$. Clearly $P_{\mathrm{BDD}} \supseteq P_{\mathrm{I}}$. We are now interested in an efficient *handling* of the constraints which define $P_{\mathrm{BDD}}$. In a branch-and-cut framework, we want to decide whether our current optimal solution $x^*$ to the linear programming relaxation lies in $P_{\mathrm{BDD}}$. If not, we want to compute an inequality which is valid for $P_{\mathrm{BDD}}$ but not valid for $x^*$. This is the so-called *separation problem* for $P_{\mathrm{BDD}}$.

BDD-SEP
Given $x^* \in \mathbb{Q}^n$ and a BDD $(G, \ell, \mathrm{par})$, decide, whether $x^* \in P_{\mathrm{BDD}}$ and if not, compute a valid inequality for $P_{\mathrm{BDD}}$ which is not valid for $x^*$.

### 2.1 Polynomial Time Solvability of BDD-SEP

In the 1980's, several authors[9, 17, 22] showed that the linear optimization problem over polyhedra and the separation problem over polyhedra are polynomial time equivalent. This *equivalence of separation and optimization* is a central result in combinatorial optimization. It implies that one can solve the separation problem for $P_{\mathrm{BDD}}$ in polynomial time, if one can solve the *optimization problem for* $P_{\mathrm{BDD}}$ in polynomial time.

BDD-OPT
Given $c \in \mathbb{Q}^n$ and a BDD $(G, \ell, \mathrm{par})$, compute a $0/1$-point which is represented by $(G, \ell, \mathrm{par})$ and is maximal w.r.t. the linear objective function $c^T x$.

BDD-OPT is easily shown to be the following longest path problem on $G$ with edge weights $w : E \to \mathbb{R}$, where

$$w(e) = \begin{cases} c(i) & \text{if } \mathrm{par}(e) = 1 \text{ and } \ell(\mathrm{head}(e)) = x(i), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

It is very easy to see that the optimal solutions to BDD-OPT are exactly the $0/1$-points which are represented by a longest path from root to leaf 1. Since $G$ is acyclic, the longest path problem can be solved in linear time. Using the equivalence of separation and optimization, we can thus conclude that BDD-SEP can, in theory, also be efficiently solved.

**Theorem 1.** *The problems* BDD-SEP *and* BDD-OPT *can be solved in polynomial time.*

### 2.2 Separation with the Subgradient Method

The point $x^* \notin P_{\mathrm{BDD}}$ if and only if there exists a $\lambda \in \mathbb{R}^n$ such that

$$\lambda^T x^* > \delta_\lambda. \tag{3}$$

The value $\delta_\lambda$ is the length of the longest path from root to leaf 1 w.r.t. the edge weights

$$w_\lambda(e) = \begin{cases} \lambda(i) & \text{if } \mathrm{par}(e) = 1 \text{ and } \ell(\mathrm{head}(e)) = x(i), \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Since $G$ is acyclic, $\delta_\lambda$ can be computed in linear time. If (3) does not hold, we update $\lambda$ as in the subgradient method to solve the Lagrangean relaxation, see e.g. [26–p. 367ff.]. In words, Alg. 1 does the following. The first guess for a normalvector of a separating

**Algorithm 1** Subgradient separation routine

(1) $k := 1$
(2) $\lambda^{(k)} := c$
(3) Compute a longest path $p^{(k)}$ from root to leaf 1 w.r.t. the edge lengths $w_{\lambda^{(k)}}$.
(4) If $\lambda^T x^* > \delta_\lambda$ then return the separating hyperplane $\lambda^T x \leq \delta_\lambda$
(5) $t^{(k)} := \frac{1}{k}$
(6) $\lambda^{(k+1)} := \lambda^{(k)} + t^{(k)}(x^* - x_{p^{(k)}})$
(7) k := k + 1;
(8) GOTO (3)

hyperplane is the objective function vector $c$, which is why $\lambda$ is initialized with this vector. Let $\lambda^{(k)}$ be the normalvector in the $k$-th iteration such that ${\lambda^{(k)}}^T x^* \leq \delta_{\lambda^{(k)}} = {\lambda^{(k)}}^T x_{p^{(k)}}$. After the update one has ${\lambda^{(k+1)}}^T(x^* - x_{p^{(k)}}) = {\lambda^{(k)}}^T(x^* - x_{p^{(k)}}) + t^{(k)}\|x^* - x_{p^{(k)}}\|^2$. If $t^{(k)} > 0$ is small enough then there exists a longest path w.r.t. $w_{\lambda^{(k)}}$, which is also a longest path w.r.t. $w_{\lambda^{(k+1)}}$. Then ${\lambda^{(k+1)}}^T x^* - \delta_{\lambda^{(k+1)}} > {\lambda^{(k)}}^T x^* - \delta_{\lambda^{(k)}}$. It is known, see [26], that for any $t^{(k)}$ with $\lim_{k\to\infty} t^{(k)} = 0$ and $\sum_{k=1}^{\infty} t^{(k)} = \infty$ the subgradient method terminates. This is the case for $t^{(k)} = 1/k$.

Geometrically, step 6 can be interpreted as a rotation of the hyperplane ${\lambda^{(k)}}^T x \leq \delta_\lambda$ in the direction of the vector $x^* - x_{p^{(k)}}$. Although Alg. 1 cannot be guaranteed to run in polynomial time, we observed that it outperforms linear programming methods for BDD-SEP by far. This is why we implemented this method in our BDD cut-separator.

## 3 Heuristics for Strengthening Inequalities with BDDs

The inequalities generated with the subgradient method naturally define faces of $P_{\mathrm{BDD}}$ with a low dimension. We want to increase their dimension in order to increase the "quality" of the hyperplanes, i.e., we are interested in facets of $P_{\mathrm{BDD}}$. Using facet-defining inequalities in branch-and-cut has led to an enourmous progress in solving large-scale optimization problems, see e.g. [16]. The standard way to turn a separating hyperplane into a facet-defining inequality, see e.g. [10], turned out to be too expensive. Therefore, we developed some heuristics to strengthen inequalities which do not guarantee to produce facets, but can be efficiently implemented.

In the following let $c^T x \leq \delta$ be a valid inequality for $P_{\mathrm{BDD}}$. The right hand side can be set to $\delta = \max\{c^T x \mid x \in P_{\mathrm{BDD}}\}$. We compute the maximum in linear time via optimizing over $P_{\mathrm{BDD}}$ with edge weights set to $w_c$ as in (2). Note that with this method every inequality can be made tight at at least one vertex of the BDD-polytope.

### 3.1 Increasing the Number of Tight Vertices

By increasing the number of vertices of $P_{\mathrm{BDD}}$ that are tight at $c^T x \leq \delta$, chances are high to also increase the dimension of the induced face. In the following we try to strengthen an inequality along the unit vectors. Remember that every path in the BDD-graph from the root to leaf 1 corresponds to a vertex of $P_{\mathrm{BDD}}$. W.l.o.g. assume that for all $i \in \{1, \ldots, n\}$ the variable $x(i)$ lies in level $i$.

Given $i \in \{1, \ldots, n\}$ we want to find a new $c(i)$ so that the number of longest paths w.r.t. the edge weights $w_c$ increases. For that we compute the sets of all longest paths, that use a 0-edge resp. 1-edge in level $i$. Be $\alpha_0$ resp. $\alpha_1$ their costs. If $\alpha_0 \neq \alpha_1$ setting $c(i) := c(i) + \alpha_0 - \alpha_1$ and $\delta := \alpha_0$ increases the number of shortest paths.

The strengthened $c$ depends on the order of the indices which we took to strengthen it. Different permutations of $\{1, \ldots, n\}$ can lead to different strengthened hyperplanes. For the computation of $c(i)$ we look at each edge of $G$ once. As we strengthen every coefficient of $c$ the total running time is $\mathrm{O}(n|A|)$. If we do not consider permutations of the indices but take the canonical order $\{1, \ldots, n\}$ we only have to use each edge three times, so the running time can be reduced to $\mathrm{O}(|A|)$.

In a branch-and-cut framework it may occur that a hyperplane separating a given $x^*$ does not separate $x^*$ after strengthening for an index $i$. In this case we do not change $c(i)$.

### 3.2 Improving Coefficients

In the following we adapted a strategy known for lifting cover inequalities for the knapsack problem, see e.g. [21]. W.l.o.g. assume that $c \geq 0$ holds. If there exists a $c(i) < 0$ replace $\delta := \delta - c(i)$ and $c(i) := -c(i)$. For simplicity reasons assume we want to strengthen $c(1)$ which means, as we have $x \geq 0$, increasing its value. Rewriting the inequality to $c(1)x(1) \leq \delta - \sum_{i=2}^{n} c(i)x(i)$ shows that we can set $c(1) := \delta - \max\{\sum_{i=2}^{n} c(i)x(i) \mid x \in P_{\mathrm{BDD}}, x(1) = 1\}$. Again we use the fact that we can optimize over $P_{\mathrm{BDD}}$ in linear time. Different permutations of indices again lead to different strengthened inequalities.

## 4 Computational Results

The cuts that we developed in this paper can be used for any $0/1$-integer program, even for those, where nothing is known about their structure. We investigated the practical strength of our theory by doing computational experiments. Our results with MAX-ONES problems and randomly generated IPs show, that one can achieve a considerable speedup on hard and small $0/1$-IPs. We report on some techniques that we developed to build BDDs fast and to keep their sizes small.

### 4.1 Heuristics for the Variable Order of the BDD

It is well-known that the variable order used in a BDD has a great influence on its size [28]. Before we start to build BDDs for any subset $A'x \leq b'$, we choose an appropriate variable order, which considers all constraints $A\,x \leq b$.

Experiments have shown that heuristics that do not take the structure of the problem into account will produce bad variable orders. We adapted an algorithm for partitioning the outputs of circuits that was presented in [12]. It proceeds as follows: First the constraint set $A\,x \leq b$ is partitioned into subsets with a similar support. Then, for every subset a partial variable order is computed. These partial orders are merged into one total order using a technique called interleaving [7].

For the partitioning of the constraints generate a new initially empty block. Delete that constraint from the set of constraints that has the largest support and insert it into the new block. This constraint is called the leader of the block. Then all constraints satisfying a certain criterion are moved to the new block. We iterate until the set of constraints becomes empty.

The two citeria we used are:

1. Add a constraint if its support is a subset of the support of the leader (WOG).
2. Add a constraint if its support is a subset of the supports of all constraints already contained in the block (BOG).

For the partial orders we used a simple heuristics. For every variable $x(j)$ we computed $h_j = \sum_{i=1}^{n} |a_{ij}|$ and sorted the variables in every block according to decreasing $h_j$ value. Before we apply the interleaving algorithm we sort the blocks increasingly by the number of variables contained in them.

Besides this algorithm that computes an inital variable order, we use sifting [25] to improve the order dynamically while we build the BDDs.

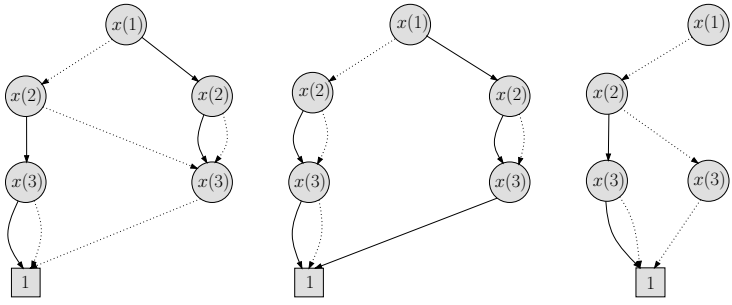
### 4.2 Building a BDD for a Subset of Constraints

We mainly use the following two operations for BDDs: *computing* the BDD for the *characteristic function of a single constraint* and the *conjunction* of two BDDs. Both BDD algorithms work recursively. The top-most variable is set to 0 and 1 and the algorithm is called recursively on the two branches. Figure 2 shows an example of a conjunction of two BDDs.

### 4.3 Decreasing the Size of the BDD

Let $a^T x \leq b$ be a constraint of the $0/1$-IP, that has not been used to build the BDD. We set the edge weights in the BDD-graph to $w_a$ as in (2) and optimize over it. A point $x_p \in P_{\mathrm{BDD}}$ satifies the constraint $a^T x \leq b$ if and only if the costs of its corresponding path $p$ are less or equal $b$. For a given node, consider the costs of all paths that cross it. If the minimum of these costs is greater than $b$, we can delete that node together with its incident edges, since we are only interested in those points of the BDD-polytope, that satisfy the given constraint.

This algorithm runs in linear time in the number of nodes of the BDD and it can be applied while the BDD is built.




(a) First Operand (b) Second Operand (c) Result

**Fig. 2.** Conjunction of BDDs: The first operand is the BDD of the characteristic function of the constraint $2x(1) - x(2) + 3x(3) \leq 2$, the second of $-x(1) + x(3) \geq 0$

### 4.4 Implementation

To evaluate the effectiveness, we implemented our methods in C++. For building BDDs we used the CUDD 2.4.0 package [27].

Before we build a BDD for the first time, we use our WOG-heuristics for finding a good variable order on the initial $0/1$-IP. In terms of finding a variable order which decreases the size of the BDDs, WOG seems to be slightly better than BOG. If the number of nodes exceeds a given limit while building the BDD, we turn on sifting occasionally. $60.000$ turned out to be a good node limit for sifting. If the size of the BDD gets too large, which means, more than 1 million nodes, we stop building it.

The BDDs that we use in our implementation differ from those that we use for theory. In practice we work on reduced BDDs. There are no redundant nodes but long edges that cross levels so that not every path from the $\mathrm{root}$ to the $\mathrm{leaf}$ $1$ contains exactly $n$ edges. This reduces memory and time consumption. On the BDD-datastructure used in CUDD we implemented an efficient version of an acyclic shortest path algorithm.

Our separation routine is called in a node of the branch-and-bound tree. To simplify building the BDD, we fix the variables according to the fixation in the branching that led us to this node. In addition to that we restrict the constraints that will be used for building the BDD to some of those of the $0/1$-IP that are tight at the LP solution in the current node. The BDD is a compact representation of all $0/1$-points that are feasible for the given constraints and possibly given fixations. If there are no fixations, the BDD gives an overapproximation of all $0/1$-points that are feasible for the $0/1$-IP. If some variables were fixed while building the BDD, the generated cuts are only valid for that face of $P_{\mathrm{BDD}}$, which corresponds to the given fixations. To make these cuts valid for $P_{\mathrm{BDD}}$ we lift them by sequentially solving LPs (see e.g. [21]).

We embedded our separation routines in the cutcallback function of the CPLEX 9.0 Branch & Cut framework [15]. Algorithm 2 sketches our separation routine.

Due to numerical problems the subgradient method sometimes does not terminate. We investigated the steplength of the rotation $t^{(k)}$ and increased the denominator by 1 not in every but in every $s$'th iteration where $s = 5$ showed to be a good value for most

**Algorithm 2** Separation via BDDs as cutcallback function

(1) *Restriction:* Fix some variables according to the branching decisions.
(2) *Build the BDD* for some of the constraints that are tight at the LP solution.
(3) *Solve the separation problem* with the subgradient method.
(4) *Strengthen* the cuts.
(5) *Lift* the strengthened cuts into the original full space and return them.

of the cases. If we cannot find a separating hyperplane after 2000 iterations we stop. To make the hyperplanes integer, we multiply them with an adequate integer value and round them. After that we compute a new right-hand-side via a shortest path computation. In almost all of the cases the resulting integer hyperplanes are still separating the current LP solution from the $0/1$-IP.

### 4.5 Benchmarks

**MAX-ONES.** Satisfiability problems notoriously produce hard to solve IPs [1]. Therefore we investigated SAT instances and converted them to MAX-ONES problems. A given SAT-instance over $n$ boolean variables and a set of clauses $C_1, \ldots, C_k$ can easily be transformed into a $0/1$-IP representing a MAX-ONES problem by converting each clause to a linear constraint of the form $\sum_i x(i) + \sum_j (1 - x(j)) \geq 1$ and adding the objective function $\max \sum_{i=1}^{n} x(i)$. From a SAT competition held in 1992 [4], we took the hfo instances. The 5cnf instances are competition benchmarks of SAT-02 [14], and the remaining instances are competition benchmarks from SAT-03 [14].

**Randomly Generated IPs.** Additionally we are interested in how our code performs on problems with less or without any structure. We randomly generated $0/1$-IPs the following way: an entry in the matrix $A$, the right-hand-side $b$ and the objective function $c$ gets a nonzero value with probability $p$. This value is randomly chosen from the integers with absolute value less or equal $c_{\max}$. The instances that we generated are available on request.

### 4.6 Results

Our experiments have been performed on a PC Intel Xeon CPU 3.06 GHz with 4 GByte RAM on GNU/Linux (kernel 2.6) operating system. Every investigated problem was solved to optimality or proved to be infeasible. On the one hand we run CPLEX 9.0 with the default values, i.e. it did presolving and used all types of built-in separation cuts. On the other hand we used the CPLEX Branch & Cut framework with our separation routines (bcBDD), but switched off presolve and all built-in cuts. We switched off presolve since we sometimes encountered problems working on the presolved model. We also tried to switch off presolve for the benchmarks made with CPLEX standalone. It showed, that presolving the randomly generated IPs does not really influence the running time but switching off presolve for the MAX-ONES instances increased CPLEX running times.

For the *MAX-ONES* instances we found out, that generating nearly all of our cuts in the root node is the most promising strategy. Using too few constraints to build the BDD resulted in weaker cutting planes. In practice it showed that 70% of the constraints, that are tight at the current LP solution, is a good threshold for generating cuts with an adequate quality while building the BDD does not consume too much time.

For *randomly generated IPs* building the BDDs is harder as the constraints have no structure. We generated our cuts deeper in the branch-and-bound tree and lifted them. Furthermore we only used 20% of the constraints, that belong to the basis of the LP solution in the current branch-and-bound node.

**Table 1.** Results for the SAT-02 / SAT-03 instances

| Name | #Var. | #Constr. | solvable | CPLEX(s) | bcBDD(s) | Speedup (%) |
|---|---|---|---|---|---|---|
| 5cnf_3800_50f1 | 50 | 760 | yes | 55.59 | 35.21 | 36.66 |
| 5cnf_3900_060 | 60 | 936 | no | 5000.01 | 3519.79 | 29.60 |
| 5cnf_3900_070 | 70 | 1092 | yes | 4523.13 | 3524.89 | 22.07 |
| 5cnf_4000_50f1 | 50 | 800 | no | 183.55 | 180.21 | 1.82 |
| 5cnf_4000_50f7 | 50 | 800 | no | 252.49 | 240.19 | 4.87 |
| 5cnf_4000_50t1 | 50 | 800 | yes | 24.21 | 12.81 | 47.09 |
| 5cnf_4000_50t3 | 50 | 800 | yes | 106.74 | 89.66 | 16.00 |
| 5cnf_4000_50t8 | 50 | 800 | yes | 125.63 | 109.32 | 12.98 |
| 5cnf_4000_60t5 | 60 | 960 | yes | 3905.54 | 3458.66 | 11.44 |
| 5cnf_4100_50f1 | 50 | 820 | no | 291.00 | 206.07 | 29.19 |
| 5cnf_4100_50f2 | 50 | 820 | no | 237.47 | 171.19 | 27.91 |
| 5cnf_4100_50f3 | 50 | 820 | no | 253.30 | 153.63 | 39.35 |
| 5cnf_4100_50f5 | 50 | 820 | no | 259.19 | 166.43 | 35.79 |
| 5cnf_4100_50f7 | 50 | 820 | no | 380.19 | 257.91 | 32.16 |
| 5cnf_4100_50t1 | 50 | 820 | no | 242.31 | 134.71 | 44.41 |
| icosahedron | 30 | 192 | no | 184.35 | 186.81 | -1.39 |
| marg2x5 | 35 | 120 | no | 22.52 | 23.51 | -4.40 |
| marg2x6 | 42 | 144 | no | 207.22 | 237.38 | -14.55 |
| marg2x7 | 49 | 168 | no | 3371.32 | 3330.37 | 1.21 |
| marg3x3add4 | 37 | 160 | no | 453.39 | 414.66 | 8.54 |
| urqh1c2x4 | 35 | 216 | no | 492.38 | 464.90 | 5.58 |
| urqh2x3 | 31 | 240 | no | 465.25 | 413.98 | 11.02 |

In all tables the running times are the total user times given in seconds. We computed the speedup as 1 minus the ratio of our running time divided by the CPLEX running time. The values for the hfo instances are average values taken over 20 different instances of each type. The standard deviation is in brackets. For 109 of the 120 hfo-instances we obtain faster running times compared to CPLEX default MIP-solver. The average of the overall speedup for the hfo-instances is 18.31% with a standard deviation of 14.44%. For the randomly generated IPs we achieved an average speedup of 34.23%.

**Table 2.** Results for the hfo instances

| Name | #Var. | #Constr. | solvable | CPLEX(s) | bcBDD(s) | Speedup(%) |
|---|---|---|---|---|---|---|
| hfo5 | 55 | 1163 | no | 3615.93 (770.64) | 2510.11 (437.74) | 27.32 (21.36) |
| hfo5 | 55 | 1163 | yes | 1917.11 (1108.12) | 1399.99 (764.53) | 22.88 (17.60) |
| hfo6 | 40 | 1745 | no | 966.84 (77.72) | 771.48 (58.14) | 19.97 (5.94) |
| hfo6 | 40 | 1745 | yes | 529.44 (256.00) | 417.65 (222.52) | 21.71 (19.69) |
| hfo7 | 32 | 2807 | no | 662.65 (60.98) | 557.29 (23.68) | 15.33 (7.47) |
| hfo7 | 32 | 2807 | yes | 346.32 (193.71) | 302.31 (156.45) | 8.93 (10.37) |
| hfo8 | 27 | 4831 | no | 690.39 (36.73) | 592.29 (19.20) | 14.02 (4.66) |
| hfo8 | 27 | 4831 | yes | 352.31 (211.31) | 297.77 (171.71) | 16.29 (11.10) |

**Table 3.** Results for the random IP instances

| Name | #Var. | #Constr. | solvable | $p$ | $c_{\max}$ | CPLEX(s) | bcBDD(s) | Speedup (%) |
|---|---|---|---|---|---|---|---|---|
| rand50_00 | 50 | 40 | no | 0.6 | 15 | 28.30 | 17.05 | 39.75 |
| rand50_01 | 50 | 50 | yes | 0.6 | 13 | 49.17 | 17.17 | 65.08 |
| rand50_02 | 55 | 50 | no | 0.6 | 15 | 41.53 | 33.43 | 19.50 |
| rand55_00 | 55 | 55 | no | 0.7 | 17 | 137.50 | 108.69 | 20.95 |
| rand55_01 | 55 | 55 | yes | 0.7 | 17 | 85.20 | 78.61 | 7.73 |
| rand60_00 | 60 | 60 | no | 0.6 | 13 | 151.65 | 85.60 | 43.55 |
| rand60_01 | 60 | 60 | no | 0.6 | 13 | 104.58 | 92.49 | 11.56 |
| rand60_02 | 60 | 60 | no | 0.6 | 13 | 237.59 | 173.62 | 26.92 |
| rand60_03 | 60 | 60 | no | 0.6 | 13 | 191.10 | 134.90 | 29.41 |
| rand60_04 | 60 | 60 | no | 0.6 | 13 | 155.90 | 106.82 | 31.48 |
| rand60_05 | 60 | 60 | no | 0.6 | 13 | 285.83 | 155.83 | 45.48 |
| rand60_06 | 60 | 60 | no | 0.6 | 13 | 678.75 | 406.58 | 40.10 |
| rand60_07 | 60 | 60 | yes | 0.6 | 13 | 84.33 | 56.26 | 33.29 |
| rand60_08 | 60 | 60 | no | 0.6 | 13 | 79.10 | 78.04 | 1.34 |
| rand70_00 | 70 | 70 | no | 0.6 | 12 | 511.62 | 280.85 | 45.11 |
| rand80_00 | 80 | 80 | yes | 0.6 | 4 | 89.47 | 31.30 | 65.02 |
| rand90_00 | 90 | 90 | yes | 0.4 | 4 | 192.36 | 85.45 | 55.58 |

## References


1. G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch & cut framework: A computational study with $\{0, \frac{1}{2}\}$-cuts. Submitted to INFORMS Journal on Computing, 2003.
2. E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.
3. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
4. M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.
5. G. Codato and M. Fischetti. Combinatorial benders' cuts. In D. Bienstock and G. Nemhauser, editors, *Integer Programming and Combinatorial Optimization, IPCO X Proceedings*, Lecture Notes in Computer Science, pages 178–195. Springer, 2004.
6. H. Crowder, E. J. Johnson, and M. Padberg. Solving large-scale 0-1 linear programming problems. *Operations Research*, 31(5):803–834, 1983.

7. H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 38 – 41, 1993.
8. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.
9. M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.
10. M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.
11. P. L. Hammer, E. Johnson, and U. N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.
12. M. Herbstritt, T. Kmieciak, and B. Becker. On the impact of structural circuit partitioning on SAT-based combinational circuit verification. In *Proceedings of 5th IEEE International Workshop on Microprocessor Test and Verification*, Austin, USA, 2004.
13. J. N. Hooker. Planning and scheduling by logic-based benders decomposition. Working paper, 2004.
14. H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent and T. Walsh, editors, *Satisfiability in the year 2000*, pages 283–292. IOS Press, 2000.
15. ILOG. *CPLEX 9.0 User's Manual and Reference Manual*. S.A., 2003.
16. M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. In *Handbook on Operations Research and Management Science*, volume 7, pages 225–330. Elsevier, 1995.
17. R. M. Karp and C. H. Papadimitriou. On linear characterizations of combinatorial optimization problems. In *21st Annual Symposium on Foundations of Computer Science*, pages 1–9. IEEE, New York, 1980.
18. Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. FGILP: an integer linear program solver based on function graphs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 685–689, 1993.
19. Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and functional decomposition. *IEEE Trans. on Computer-Aided Design*, 13(8):959–975, 1994.
20. C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell Systems Technical Journal*, 38:985 – 999, 1959.
21. G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley, 1988.
22. M. W. Padberg and M. R. Rao. The russian method for linear programming III: Bounded integer programming. Technical Report 81-39, New York University, Graduate School of Business and Administration, 1981.
23. M. W. Padberg, T. J. Van Roy, and L. A. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, 33(4):842–861, 1985.
24. M. W. Padberg and L. A. Wolsey. Fractional covers for forests and matchings. *Mathematical Programming*, 29(1):1–14, 1984.
25. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 42–47, 1993.
26. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley, 1986.
27. F. Somenzi. *CU Decision Diagram Package Release 2.4.0*. Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2004.
28. I. Wegener. *Branching programs and binary decision diagrams*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, PA, 2000.
29. L. A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.

# Parallel Smith-Waterman Algorithm for Local DNA Comparison in a Cluster of Workstations*

Azzedine Boukerche[1], Alba Cristina Magalhaes Alves de Melo[2], Mauricio Ayala-Rincon[3], and Thomas M. Santana[2]

[1] SITE, University of Ottawa, Canada
[2] Department of Computer Science, University of Brasilia (UnB)
[3] Department of Mathematics, University of Brasilia (UnB)

**Abstract.** Biological sequence comparison is one of the most important and basic problems in computational biology. Due to its high demands for computational power and memory, it is a very challenging task. Most of sequence comparison methods used are based on heuristics, which are faster but there are no guarantees that the best alignments will be produced. On the other hand, the algorithm proposed by Smith-Waterman obtains the best local alignments at the expense of very high computing power and huge memory requirements. In this article, we present and evaluate our experiments with three parallel strategies to run the Smith-Waterman algorithm in a cluster of workstations using a Distributed Shared Memory System. Our results on an eight-machine cluster presented very good speedups and indicate that impressive improvements can be achieved, depending on the strategy used. Also, we present some theoretical remarks on how to reduce the amount of memory used.

## 1 Introduction

Biological sequence comparison (or sequence alignment) is one of the most important problems in computational biology, given the number and diversity of the sequences and the frequency on which it is needed to be solved daily all over the world [1]. Sequence comparison is in fact a problem of finding an approximate pattern matching between two sequences, possibly introducing spaces (gaps) into them. The most important types of sequence alignment problems are global and local. To solve a global alignment problem is to find the best match between the entire sequences. Local alignment algorithms must find the best match (or matches) between parts of the sequences. In this article, we will treat mainly local alignments.

Smith and Waterman [2] proposed an algorithm (SW) based on dynamic programming to solve the local alignment problem. It is an exact algorithm that finds the best local alignments between two genomic sequences of size $n$ in quadratic time and space complexity $O(n^2)$. In genome projects, the size of the sequences to be compared are constantly increasing, thus an $O(n^2)$ solution is still expensive. For this reason, heuristics were proposed to reduce time complexity to $O(n)$. BLAST [3] and FASTA [4] are

* This work was partially supported by NSERC, Canada Foundation for Innovation and Canada Research Chair Programs.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 464–475, 2005.
© Springer-Verlag Berlin Heidelberg 2005

examples of widely used heuristics to compute local alignments. SW is the most sensitive method but also the slowest one for similarity searches between sequences. One obvious improvement is the use of parallel processing to speedup the SW computations. Even in this case, the quadratic space complexity remains a problem and techniques must be used to reduce it. Martins [5] and MASPAR[6] proposed techniques to run the SW algorithm in, respectively, a Beowulf machine with 128 processors and a massively parallel computer system with 16384 processors. Although the results obtained in both cases were good, the cost of such machines make this an expensive approach. Decipher [7] is a dedicated hardware based on FPGAs that implements the SW algorithm. This is also an expensive approach.

In this paper, we propose and evaluate a new parallel SW algorithm in a cluster of workstations that uses commodity hardware and operating system. We also compare the results obtained with this approach with two parallel SW algorithms [8] [9] that were earlier deployed by our research group. These first two strategies (*heuristic* and *heuristic-block*) are approximate ones since they use heuristics to reduce the space complexity. The strategy proposed in this paper (*pre-process*) is an exact one that stores intermediate results into disk. Additionally, we propose a modification to the original SW algorithm which runs in space complexity $O(n + n'^2)$, where $n'$ is the maximal length of a local alignment between sequences $s$ and $t$ of size $n$. The results obtained in an eight-machine cluster with large sequence sizes show good speedups when compared with the sequential algorithm. Moreover, the proposed technique provides great improvements over the former strategies. For instance, to compare 80KBP (kilo-base pair) sequences using eight processors, the *pre-process* strategy run approximately 12 times faster than *heuristic*.

## 2 Smith-Waterman's Algorithm for Local Sequence Alignment

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters or substrings from the sequences [1]. In an alignment, spaces are inserted in arbitrary locations along the sequences so that they end up with the same size.

Given an alignment between two sequences *s* and *t*, a score is usually associated for it as follows. For each column, we associate *+1* if the two characters are identical, *-1* if the characters are different and *-2* if one of them is a space. The score is the sum of the values computed for each column. The maximal score is the similarity between the two sequences, denoted by *sim(s,t)*. In general, there are many alignments with maximal score. Figure 1 shows the alignment of sequences *s* and *t*, with the score for each column. In this case, there are nine columns with identical characters, one column with distinct character and one column with a space, giving a total score *6*.

$$\begin{array}{cccccccccccc} G & A & - & C & G & G & A & T & T & A & G \\ G & A & T & C & G & G & A & A & T & A & G \\ \hline +1 & +1 & -2 & +1 & +1 & +1 & +1 & -1 & +1 & +1 & +1 \end{array}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\Sigma = 6}$$

**Fig. 1.** Alignment between $s = GACGGATTAG$ and $t = GATCGGAATAG$

For long sequences, it is unusual to obtain a global alignment. Instead, the local alignment algorithm is executed to detect regions inside both sequences that are similar. Smith-Waterman proposed an algorithm (SW) based on dynamic programming to solve the local alignment problem. The time and space complexity of this algorithm is $0(mn)$, where $m$ and $n$ are the lengths of the two sequences, and, if both sequences have approximately the same length, $n$, we get $O(n^2)$ [2].

The SW algorithm is divided into two parts: the calculation of the *similarity array* and the retrieval of the local alignments, which are to be explained in the following sections.

• **Part 1: Calculation of the Similarity Array**: As input, the algorithm receives two sequences *s*, with $|s| = m$, and *t*, with $|t| = n$, where $|s|$ denotes the length of the sequence $s$. There are $m + 1$ possible prefixes for *s* and $n + 1$ prefixes for *t*, including the empty string. An array $A_{m+1,n+1}$ is built, in which the $A[i, j]$ entry contains the value of the similarity between two prefixes of *s* and *t*, $sim(s[1..i], t[1..j])$. Figure 2 shows the similarity array between *s*=ATAGCT and *t*=GATATGCA. To obtain local alignments, the first row and column are initialized with zeros. The other entries are computed using equation 1.

$$sim(s[1..i], t[1..j]) = \max \begin{Bmatrix} sim(s[1..i], t[1..j-1]) - 2, \\ sim(s[1..i-1], t[1..j-1]) + (\text{ if } s[i] = t[j] \text{ then } 1 \text{ else } -1), \\ sim(s[1..i-1], t[1..j]) - 2, \\ 0 \end{Bmatrix} \quad (1)$$

The values $A[i, j]$, for $i, j > 0$, are defined as $sim(s[1..i], t[1..j])$.

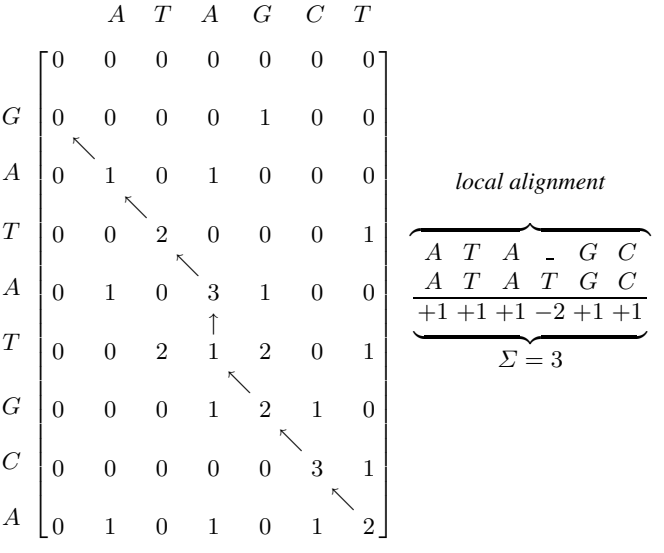



**Fig. 2.** Array to compute the similarity between the sequences $ATAGC$ and $ATATGC$, using the SW algorithm

We have to compute the array $A$ row by row, left to right on each row, or column by column, top to bottom, on each column. Finally, arrows are drawn to indicate where the maximum value comes from, according to equation 1. In the example, the score value of the best local alignment appears in $A[5,4]$ and $A[8,6]$. Figure 2 illustrates the similarity array obtained with the SW algorithm to compute local alignments of two sequences.

• **Part 2: Retrieval of the Local Alignments**: An optimal local alignment between two sequences can be obtained as follows. We begin in a maximal value at array $A$, and follow the arrow going out from this entry until we reach another entry with no arrow going out, or until we reach an entry with value 0. Each used arrow gives us one column of the alignment. A west arrow leaving entry $A[i,j]$ corresponds to a column with a space in $s$ matching $t[j]$, a north arrow corresponds to $s[i]$ matching a space in $t$ and a north-west arrow means $s[i]$ matching $t[j]$. After computing $A$, an optimal local alignment is constructed from right to left following these arrows. Many optimal local alignments may exist for two sequences. The detailed explanation of this algorithm can be found in [2].

## 3 Reducing the Space Complexity of the SW Algorithm

To obtain local alignments, we implemented a variant of the Smith-Waterman algorithm that uses two linear arrays [1]. The bi-dimensional array described in section 2 could not be used since, for long sequences, the memory overhead would be prohibitive. The idea behind this algorithm is that it is possible to simulate the filling of the original bi-dimensional array just using two rows in memory, since, to compute entry $A[i,j]$ we just need the values of $A[i-1,j]$, $A[i-1,j-1]$ and $A[i,j-1]$. So, the space complexity of this version is linear, $O(n)$. The time complexity remains $O(n^2)$. The algorithm works with two sequences $s$ and $t$ with length $|n|$. First, one of the linear arrays is initialized with zeros. Then, each entry of the second linear array is obtained from the first one with the Smith-Waterman algorithm, but using a single character of $s$ on each step. When the calculated score reaches an *opening* threshold, the actual coordinates are saved to the data structure *alignments*, which contains the candidate alignments. When the score drops to less than a *closing* threshold, the actual coordinates are saved into the same data structure and the alignment is said to be *closed*. More details on this heuristics can be found in [5].

• **Parallel Local Sequence Alignment without blocking factors**: The access pattern presented by the algorithm described earlier, presents a non-uniform amount of parallelism and has been extensively studied in the parallel programming literature [10]. The parallelization strategy that is traditionally used in this kind of problem is known as the "wave-front method" since the calculations that can be done in parallel evolve as waves on diagonals.

We deployed a parallel version of the algorithm presented earlier (see also [11]). Each processor $p$ acts on two rows, a writing row and a reading row. Work is assigned in a column basis, i.e., each processor calculates only a set of columns on the same row, as shown in figure 3.

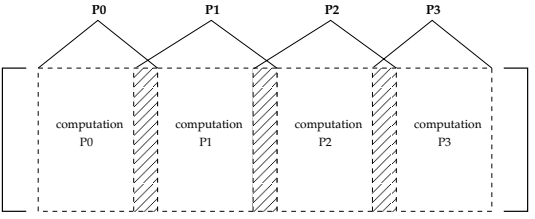



**Fig. 3.** Work assignment in the parallel algorithm. Each processor *p* is assigned *N/P* columns, where *P* is the total number of processors and *N* is the length of the sequence

**Table 1.** Execution times (seconds) and speedups for 5 sequence sizes

| Size | Serial exec | 2 proc exec/speedup | 4 proc exec/speedup | 8 proc exec/speedup |
|---|---|---|---|---|
| 15Kx15K | 296 | 283.18/1.04 | 202.18/1.46 | 181.29/1.63 |
| 50K x 50K | 3461 | 2884.15/1.20 | 1669.53/2.07 | 1107.02/3.13 |
| 80Kx80K | 7967 | 6094.18/1.31 | 3370.40/2.46 | 2162.82/3.68 |
| 150Kx150K | 24107 | 19522.95/1.23 | 10377.89/2.32 | 5991.79/4.02 |
| 400Kx400K | 175295 | 141840.98/1.23 | 72770.99/2.41 | 38206.84/4.58 |

The parallel programming paradigm used was Distributed Shared Memory (DSM) [12], which creates a shared memory abstraction that parallel processes can access. Synchronization is achieved by locks and condition variables provided by JIAJIA [13], which is a scope consistent DSM system. Barriers are only used at the beginning and at the end of computation.

The proposed parallel algorithm was implemented in C, using the software DSM JIAJIA v.2.1. To evaluate the gains of our strategy, we ran our experiments on a dedicated cluster of 8 Pentium II 350 MHz, with 160 MB RAM connected by a 100Mbps Ethernet switch. The JIAJIA software DSM system ran on top of Debian Linux 2.1 with NFS. Our results were obtained with real DNA sequences obtained from [**?**]. Five sequence sizes were considered (15KBP, 50KBP, 80KBP, 150KBP and 400KBP) [11]. Execution times and absolute speedups for each $n$ x $n$ sequence comparisons, where $n$ is the approximate size of both sequences, with 1, 2, 4 and 8 processors, are shown in table 1. Absolute speedups were calculated considering the total execution times and thus include times for initialization and collecting results. As can be seen in table 1, for small sequence sizes, e.g. 15KBP, very bad speedups were obtained since the parallel part is not long enough to surpass the amount of synchronization inherent to the algorithm. As long as sequence sizes increase, better speedups are obtained because more work can be done in parallel.

• **Parallel Local Sequence Alignment with blocking factors**: Using the wavefront method and obtaining good performance results is sometimes tricky and it is worth to investigate if the communication time can be reduced by grouping many values from the border column (figure 3) into one single communication, which is to introduce blocking factors to the strategy presented in section 3.

In order to obtain an appropriate block size, we run our algorithm for the 50KBP sequences, varying the block size and the number of bands, using a blocking multiplier. For instance, a 3x5 blocking multiplier for 8 processors divides the matrix into 40 bands (5x8), each one containing 24 blocks (3x8). In this test, we observed that for the 50KBP sequences and 8 processors, the best results were obtained for a blocking multiplier 5x5, which means 40 bands, each one with 40 blocks.

Execution times and speedups to locally align the 8KBP, 15KBP and 50KBP sequences, with 1,2,4 and 8 processors are shown in Table 2 [9]. The same cluster as described earlier was used. Speedups were calculated considering the total execution time and thus include times for initialization and collecting results.

**Table 2.** Execution times (seconds) and speedups for 3 sequence sizes

| Size | Bands | Serial exec | 2 proc exec/speedup | 4 proc exec/speedup | 8 proc exec/speedup |
|---|---|---|---|---|---|
| 8Kx8K | 40x40 | 57.18 | 38.59/1.48 | 21.18/2.72 | 12.55/4.55 |
| 15Kx15K | 40x40 | 266.51 | 129.22/1.98 | 67.42/3.95 | 36.51/7.29 |
| 50K x 50K | 40x25 | 2620.64 | 1352.76/1.93 | 701.95/3.73 | 363.13/7.21 |

## 4 Reducing Space Complexity Without Introducing Heuristics

The key goal of this third strategy is to calculate the similarity array for local sequence alignment without introducing heuristics, since our objective here is to execute the original SW algorithm. Besides calculating the similarity array, it allows saving of the array to disk, either partially or entirely. As in the previous strategies, this third parallel strategy was designed for the shared memory programming paradigm. However, several decisions were made to limit the amount of shared memory and the activity of the shared memory system. The following concepts guided the design:

- only a limited amount of the similarity array should be shared;
- processing of the array would be done by columns, and nodes should send information to the next node once the bottom of the column had been calculated;
- full alignment tracking would not be implemented, only a scoreboard of points of interest will be kept;
- saving the resulting array should be done to disk.

Figure 4 shows the general layout of the data in each processing node. With the exception of the shaded areas, which represent shared memory areas, all other structures are local to the node. The *passage band* is an array used to allow data to be moved from one node to the other. To avoid the excessive use of locking, columns are processed in *chunks*. Each column is stored in a linear array, to improve intra-node locality. A *band*represents a set of columns in the length of the top sequence. The *result matrix* is used to hold the number of cells in a column that had scores above a given threshold.

The innermost processing is done computing a column from the previous one: only 4 cells are handled at a time. Unlike the previous strategies, there is no structure to keep

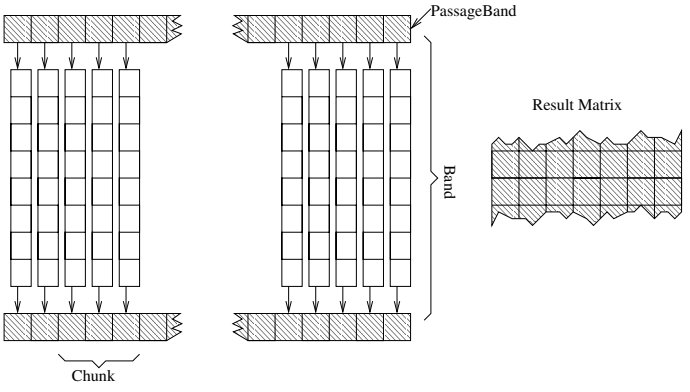



**Fig. 4.** Data structures used in the implementation without heuristics

track of alignments. When a new cell score is calculated, the score value is compared to a threshold and if it is greater than it a *hit* counter is incremented. The total number of hits per column or set of columns is stored in the result matrix. The goal of this simple processing is to reduce the execution time while still maintaining potential similarity regions. Notice that in this way we will provide exact, but also approximate answers.

Although little information is contained in the result matrix, it does indicates interesting regions in the score matrix. If one consider 1,000 columns per cell in the matrix and a 1,000 row band size, each cell will contain the total hits for 1,000,000 cells of the score matrix. So having the total number of hits will hint on whether there is something worth more investigation in that block of data. On some of the test runs, some of the result matrix cell indicated 300,000 hits or more. Values at this level indicate that 30% of the cells were above the threshold, so that region is very likely to contain good alignments.

The limited visibility of the result matrix may be worthwhile if the overall performance is good. Knowing interesting areas of the matrix and having the boundary columns and rows, one can reprocess those limited areas to retrieve the local alignments.

Several parameters were added to control the behavior of this implementation. The most important ones include: the height of the band in rows; the chunk size and growth method; the number of columns to be stored into disk; and the amount of columns summarized in each cell of the result matrix.

The size of the chunks can be set to a fixed value, or grow in arithmetic or geometric projections. It is possible to save all the columns but, in the tests, only a reduced number of columns were saved. Also it is possible to store individual columns at the cost of more shared memory.

Each cell ($R_{i,j}$) of the result matrix contains the total number of hits from all the columns $n$ of band $i$ that satisfy the following equation $\lfloor n/ip \rfloor = j$, where $ip$ is the result matrix interleave parameter.

The columns that should be saved to disk are also controlled by the save interleave parameter $ip$, such that any given column $i$ will be saved to disk if $i \neq 0$ and ($i \mod ip = 0$). This interleave parameter can be used to increase or reduce the amount

of information available in the end result. If the save interleave parameter is set to 1, the entire matrix will be saved, but even for small sequences this can require a huge amount of disk. For instance, the comparison of two 10KBP sequences would require 400 MBytes, just for the column data. All passage bands are saved once the last cell in it is updated.

Besides having ways to control the amount of data being grouped, the band size is controlled by three different schemes. The first one determines fixed band size (or height). Another scheme uses even or equal bands, so that all the nodes have the same amount of data to process. The final mode, attempts to balance the band size so that it is still close to the designated band size according to the following equations:

$$bands_{proc} = \left\lceil \frac{\lceil r_{size}/b_{size} \rceil}{n_{nodes}} \right\rceil$$

$$b_{size_{down}} = \left\lceil \frac{b_{size}}{bands_{proc} * n_{nodes}} \right\rceil$$

$$b_{size_{up}} = \left\lceil \frac{b_{size}}{(bands_{proc} - 1) * n_{nodes}} \right\rceil$$

The new band size will be $b_{size_{up}}$ or $b_{size_{down}}$, whichever is nearer to the original band size. The objective is to make all nodes process the same number of bands of equal size, while maintaining the blocking concept.

### 4.1 Experimental Results

Using the same platform described earlier, several tests were made varying some of the configuration parameters. Processing times shown in our results consider only the parallel execution time since this is the largest of the measured times and gives a good indication of the overall performance. The init time is mostly due to the overhead of starting the remote processes by the DSM environment (and ran under 10 seconds for all tests). The termination time basically contains the time for completing I/O and final synchronization times of the DSM environment. For the latter the worst case was 20 seconds, but most likely because of the NFS and DSM overhead. Most termination times were under 7 seconds.

Figure 5 presents two speedup analysis. The first graph shows the speedup on the average time for all the different configurations. On this case, speedups are roughly 75% of the linear case. On the 8 processor case, the average time for the 15Kx15K sequence is slightly lower due to the fact that with large blocking (4K blocks), several processors where not used at all. In those cases, the 8 node times were close to the 4 node times, resulting in a bad average.

The second graph shows the speedup of the best time of the parallel execution, against the best time of the sequential execution. In this case, for the larger sequences, speedups are near to the 80% mark. In both graphs, the 2 node speedups are slightly worst, and this is due mostly to the fact that the sequential execution does not have any DSM overhead at all, and the parallel implementation is still not having enough effect to overcome the added overhead. However, even with 2 nodes, there is some performance gain.

Figure 6 shows the effects of the blocking parameter in the run time. All three blocking parameters (band size, save interleave, and result matrix interleave) are set to the

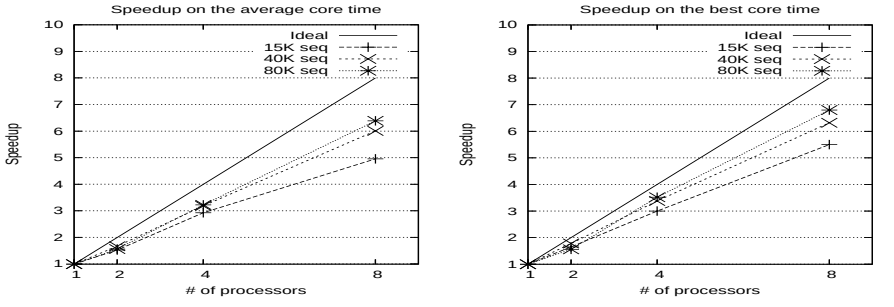



**Fig. 5.** Speedup for the parallel implementation

same value. In this graph, we can observe an interesting effect. On the sequential execution, the runs with "even" blocking factors behave the worst, around 20% above the other times. This is due to the fact that the band size is the length of the sequence and with a 40KBP or 80KBP sequences this has a negative impact in the memory locality within the CPU cache, and results it poor performance.

As the number of nodes increases, the even division of blocks makes the blocks smaller and reduces the amount of DSM activity, resulting in better result for this method. The even distribution of bands size also tries to balance the amount of work on all the nodes.

We compared the results obtainded with 8 processors with the three strategies (*heuristic*, *heuristic-block* and *pre-process* using 15KBP, 50KBP and 80KBP sequences (figure 7). The results illustrate the high performance gain that we achieved only by changing the strategy. For instance, to compare 50KBP sequences, the *heuristic* strategy ran in

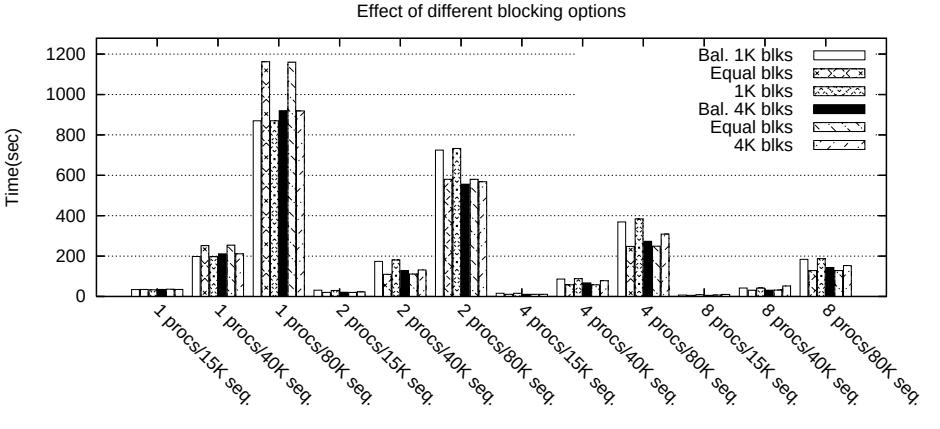



**Fig. 6.** Effect of different blocking parameters on the run time

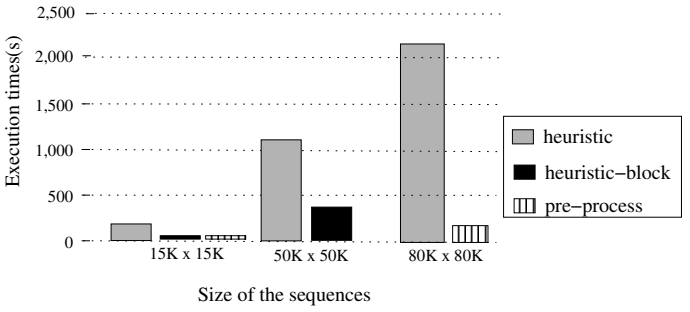



**Fig. 7.** Execution times for 8 processors with the blocking and non-blocking strategies

1107.02s. Using the *heuristic-block* approach, the same experiment took 363.13 seconds. Also, to compare 80KBP sequences, the *heuristic* approach took 2162.82s and, for the same size, the *pre-process* strategy took only 178.5s.

## 5 Reducing Space Complexity Without Intermediary Stores

We present theoretical observations that can be applied to decrease the running time and space of our current implementations. In particular, we show how we can develop an implementation running in space $O(min(n, m) + n'^2)$, where $n'$ is the maximum length of a local alignment between words $s$, $t$ of size $n$ and $m$, respectively. Since in general $n' << n$ and $n' << m$, this is a remarkable improvement. When, after detecting an alignment position, the required space for building the alignment is small (that is $n'$ is small) one could apply the well-known Hirschberg's general method for doing it in linear space while only doubling the worst-case time bound (in $n'$) [14]. But, in general we are supposing that $n'$ is too large to be processed simultaneously in main memory.

The basic idea is founded on the fact that whenever there is an alignment finishing at positions $i$ and $j$ of the words $s$ and $t$, respectively, there is an alignment of the same score starting at positions $n - i + 1$ and $m - j + 1$ between the reverses of these words, which we will denote by $s^{rev}$ and $t^{rev}$, respectively. This alignment corresponds to a global alignment of the same score between the words $s[1..i]^{rev}$ and $t[1..j]^{rev}$. The former is expressed as the following observation.

**Observation 5.1 (Alignments over reverses).** *Suppose there is a local alignment of score $k$ finishing at positions $i$ and $j$ between the words $s$ and $t$. Then there is an alignment of the same score starting at positions $n - i + 1$ and $m - j + 1$ of the words $s^{rev}$ and $t^{rev}$.*

This result justifies the correctness of the following modification of the dynamic programming based solution:

**Algorithm for reducing space complexity**

Run the SW algorithm over the input words $s, t$ using only one linear array (for row, column or diagonal);
For each detected alignment of desirable score, say $k$ at positions $i, j$,

Run the dynamical programming algorithm for over the input words $s[1..i]^{rev}$ and $t[1..j]^{rev}$ until you detect an alignment of score $k$.
Rebuild the alignment of the reverses over the initial inputs.

The space comlexity of the proposed scheme can easily be computed as $O(min(n, m) + n'^2)$, where $n'$ is the maximal length of selected maximal alignments. The proposed modification is relatively easy to implement whenever one could work over the given input sequences from left to right.

## 6 Conclusion and Future Works

In this paper, we discussed two parallel SW strategies and proposed and evaluated a new parallel strategy to solve the DNA local sequence alignment problem. A DSM system was chosen since, for this kind of problem, DSM offers an easier programming model than its message passing counterpart.

The first two strategies (*heuristic* and *heuristic_block*) used heuristics to reduce the space complexity to $O(n)$, where $n$ is the size of the sequences to be compared. The third strategy (*pre_process*) allowed the original SW algorithm to be run by saving information about the most relevant columns of the result matrix to disk. These columns can be later processed in order to retrieve the actual alignments. For all strategies, the wavefront method was used and work was assigned in a column basis. Synchronization was achieved with locks and condition variables. Barriers were only used at the beginning and at the end of computation.

The results obtained to locally align real DNA sequences in an eight machine cluster present good speedups that are improved depending on the strategy used. For instance, in order to compare two sequences of approximately 80KBP using the *heuristic* and *pre-process* approaches, our 8-machine cluster took 36 minutes and 2.8 minutes, respectively.

Thus,our results show that an appropriate blocking factor can further reduce the execution time to obtain local alignments. Besides, using the *pre-process* strategy that does not keep track of the candidate alignments can also provide a great performance gain at the expense of storing some intermediate columns to disk.

We also presented some theoretical results that eliminate the need for disk storage by maintaining only the coordinates of the highest score and retrieving the actual alignments over the reverses of the original sequences.

As immediate future work, we intend to implement the modifications suggested in section 5. Also, we intend to run this modified algorithm to compare very long DNA sequences (larger than 1MBP) in a computational grid, which is composed by several clusters. In this case, message passing will be used for inter-cluster communication and DSM will be used for communicating processes which belong to the same cluster.

## References


1. Setubal, J.C., Meidanis, J.: Introduction to Computational Molecular Biology. Brooks/Cole Publishing Company (1997)
2. Smith, T.F., Waterman, M.S.: Identification of common molecular sub-sequences. Journal of Molecular Biology (1981) 195–197
3. et al., S.F.A.: Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic Acids Research **25** (1997) 3389–3402
4. Pearson, W.R., Lipman, D.L.: Improved tools for biological sequence comparison. In: Proc. National Academy Of Science, NAS (1988) 2444–2448
5. Martins, W.S., Cuvillo, J.B.D., Useche, F.J., Theobald, K.B., Gao, G.R.: A multithread parallel implementation of a dynamic programming algorithm for sequence comparison. In: Symp. on Computer Architecture and HPC (SBAC-PAD). (2001) 1–8
6. group, D.: Smith waterman homology search (2003)
7. Co., D.: Decypher smith waterman solution (2003)
8. Boukerche, A., Melo, A.C.M.A., Walter, M.E.M.T., Melo, R.C.F., Santana, M.N.P., Batista, R.B.: Performance evaluation of a local dna sequence alignment algorithm on a cluster of workstations. In: Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS2004), IEEE Society (2004)
9. Batista, R.B., Silva, D.N., Melo, A.C.M.A., Weigang, L.: Using a dsm application to locally align dna sequences. In: Proc. of the IEEE/ACM Int. Symp. on Cluster Computing and the Grid, IEEE Computer Society (2004)
10. Pfister, G.: In Search of Clusters - The Coming Battle for Lowly Parallel Computing. Prentice-Hall (1995)
11. Melo, R., Walter, M.E.T., Melo, A.C.M.A., Batista, R.B.: Comparing two long dna sequences using a dsm system. In: Euro-Par 2003, Springer-Verlag (2003) 517–524
12. Mosberger, D.: Memory consistency models. Operating Systems Review (1993) 18–26
13. Hu, S., Shi, W., Tang, Z.: Jiajia: An svm system based on a new cache coherence protocol. In: High Performance Computing and Networking (HPCN), Springer-Verlag (1999) 463–472
14. Hirschberg, D.S.: Alagorithms for the longest common sequence problem. Journal of the ACM **24** (1977) 664–675

# Fast Algorithms for Weighted Bipartite Matching

Justus Schwartz, Angelika Steger, and Andreas Weißl

ETH Zürich, Institute of Theoretical Computer Science, ETH Zentrum,
CH 8092 Zürich, Switzerland
{jschwart,steger,aweissl}@inf.ethz.ch

**Abstract.** Let $G = (V_1 \cup V_2, E)$ be a bipartite graph on $n$ nodes and $m$ edges and let $w : E \to \mathbb{R}_+$ be a weight function on the edges. We give several fast algorithms for computing a minimum weight (perfect) matching for a given complete bipartite graph (i.e. $m = n^2$) by pruning the edge set. The algorithm will also output an upper bound on the achieved approximation factor. Under the assumption that the edge weights are uniformly distributed, we show that our algorithm will compute an optimal solution with high probability. From this we deduce an algorithm with fast expected running time that will always compute an optimal solution. For real edge weights we achieve a running time of $O(n^2 \log n)$ and for integer edge weights a running time of $O(n^2)$.

## 1 Introduction and Motivation

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph on $n$ nodes and $m$ edges and let $w : E \to \mathbb{R}_+$ be a weight function on the edges. A perfect matching of $G$ is a subset $M \subseteq E$ of the edges such that for every node $v \in V = V_1 \cup V_2$ there is exactly one incident edge $e \in M$. The weight of a matching $M$ is given by the sum of the weights of its edges, i.e. $w(M) := \sum_{e \in M} w(e)$. Now, the minimum weight bipartite matching problem is to find for a given bipartite graph $G$ and a given weight function $w$ a perfect matching of minimum weight. In operations research this problem is often called the assignment problem. In the following we assume w.l.o.g. that $|V_1| = |V_2|$.

In practice, this problem appears in a wide range of applications. For example, in bioinformatics a well know problem is the protein structure alignment to compare and classify all known protein structures. Therefore a protein is compared to every protein in a database, which holds information about their 3-D structures. This can give insights into the nature of protein structures and their functional mechanisms. For aligning two proteins, one has to find an isometric transformation of one structure to the other. To find good choices of atoms pairings between the two structures, one has to solve a weighted bipartite matching problem on an appropriate graph, where the nodes represent atoms or groups of atoms and the edge weights are given by a suitable distance function [18].

Another example appears in logistics and warehousing. Here, one wants to place products in shelves such that the access time to the products is minimised.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 476–487, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The weights can model the access time depending as well on attributes of the products as access frequency, weight and size as on the place in the shelf – for example access time can be fastest in the centre of the shelf. Clearly, this problem can be formulated as a weighted bipartite matching.

Other important applications can be found in shape matching and object recognition [2], image processing and computer vision [3], and VLSI design [14].

Over the last decades the problem has attracted considerable attention and several algorithms – improving theoretical and practical running times – have been developed. The first polynomial time algorithm, the so called Hungarian method, was given by Kuhn [17]; implementations of this algorithm had a running time of $O(n^3)$, which is still optimal for dense graphs. Edmonds and Karp [4] gave a reduction of the minimum weight bipartite matching problem to the minimum cost flow problem on a special network (see Section 2.1) and developed an algorithm for solving the flow problem. They proved a running time of $O(n^3)$. This was improved for sparse graphs by Fredman and Tarjan [6] using Fibonacci Heaps for the shortest paths computations to a running time of $O(n(m + n \log n))$, which is still optimal for sparse graphs with arbitrary edge weights.

Under the assumption that the input weights are integers in the range $[0, \ldots, C]$ – in practice this can almost always be achieved by scaling the input weights – Gabow and Tarjan improved the running time to $O(\sqrt{n}m \log(nC))$ by a cost scaling approach and blocking flow techniques [9]. In [11] Goldberg and Kennedy developed an algorithm with the same running time using global price updates and a push-relabel implementation for the minimum cost flow. Very efficient implementations of this algorithm show that it is quite fast in practice, too. Several cost scaling algorithms with the same running time of $O(\sqrt{n}m \log(nC))$ were independently developed. Gabow and Tarjan [8] presented an algorithm with the same running time and a similar idea to global updates. Moreover, two-phase algorithms with the same running time were developed by Goldberg, Plotkin and Vaidya [12] and by Orlin and Ahuja [20]. These two algorithms use in a first phase a push-relabel method or an auction method (due to Bertsekas) and in a second phase a successive shortest paths augmentation approach. Karp [16] gave an algorithm with expected running time of $O(n^2 \log n)$ on graphs with independent random edge weights. For further references we refer to [1].

Furthermore, for many of these algorithms experimental studies for evaluating and comparing their run time behaviour in practice were done. Additionally, heuristics for speeding up these algorithms were developed [15].

For many practical problems, in particular those involving dense graphs, these running times are still too time consuming even on today's hardware. In practice, one often does not need to find the optimal solution, but an approximation of the optimal solution is good enough, as long as the algorithm guarantees an upper bound on the approximation factor.

Hence, much effort was put into developing faster algorithms with good approximation guarantees. Most of these algorithms exploit a restriction of the weight function, for example they assume that the triangle inequality holds

(e.g. [10, 21]). A fast and quite simple approximation algorithm for the weighted matching in general graphs and weight functions fulfilling the triangle inequality was recently given by Wattenhofer and Wattenhofer [22].

**Our results**

This research was initiated by a joint project with a company developing software for combinatorial optimisation. The problem involved solving minimum weight bipartite perfect matching for large (i.e. $n > 10000$) and dense (i.e. $m = \theta(n^2)$) graphs where the standard algorithms with running times of $O(n^3)$ (and $O(\sqrt{n}m \log(nC))$ for integer weights) were impracticable. We therefore implemented and tested several heuristics with a running time of $\tilde{O}(n^2)$. In this paper we report on one variant that performs well on real-world data and is also provable almost optimal on complete bipartite graphs with random edge weights.

The main idea of our approach is to discard a large subset of the edges of the input graph. Then we can use one of the fastest known algorithms for sparse graphs to achieve a good running time. So in case of arbitrary edge weights we use the algorithm of Fredman and Tarjan [6] and for integer weights we use a cost scaling algorithm of Goldberg and Kennedy [11].

In detail we prove the following theorems. For a complete bipartite graph with arbitrary weight function we compute with high probability (w.h.p. that is with probability tending to 1 as $n \to \infty$)in time $O(n^2 \log n)$ a perfect matching – not necessarily optimal – and an upper bound on the achieved approximation factor. This algorithm is quite fast and still gives good approximations for many input classes.

**Theorem 1.** *There exists a randomised algorithm with running time $O(n^2 \log n)$, which computes with high probability for a given complete bipartite graph with arbitrary weight function on the edges a weighted perfect matching and an upper bound on the achieved approximation factor, if the matching is not optimal.*

Under the assumption that the input graph is a complete bipartite graph and the edge weights are uniformly distributed on the interval $[0, 1]$, we prove that the algorithm will output an optimal solution with high probability. For the proof we use a result of Frieze and Sorkin [7] which bounds the heaviest edge in a minimum weight perfect matching for a complete bipartite graph, if the edge weights are uniformly distributed.

**Theorem 2.** *There exists an algorithm with running time $O(n^2 \log n)$, which computes with probability $1 - o(n^{-1})$ for a given complete bipartite graph with uniformly distributed edge weights, a minimum weight perfect matching. Moreover it computes an upper bound on the achieved approximation factor, if the matching is not optimal.*

From this we can easily deduce an algorithm with fast expected running time which always computes an optimal solution: First, we run the algorithm of Theorem 2. Then we check the optimality of the solution for the complete input graph using the dual variables that were computed on the sparse graph. We

show that w.h.p. this dual solution will also fulfil the complementary slackness conditions of the original graph. If the check fails, we run the algorithm on the complete graph. As this check won't fail w.h.p., we achieve a good expected running time. Thus, we get the following corollary:

**Corollary 1.** *There exists an algorithm with expected running time $O(n^2 \log n)$, which computes a minimum weight perfect matching for a given complete bipartite graph with uniformly distributed edge weights.*

If the weights are integers in the range $[0, C := O(n^c)]$, where $c$ is a constant, we can run a faster algorithm for the matching computation, for example a cost scaling algorithm of Goldberg and Kennedy [11] which runs in $O(\sqrt{n}m \log(nC))$. Then the overall running time is dominated by choosing a small subset of the $n^2$ input edges and testing for optimality by checking the $n^2$ complementary slackness conditions, which both need $O(n^2)$ time.

**Theorem 3.** *There exists an algorithm with expected running time $O(n^2)$ which computes a minimum weight perfect matching for a given complete bipartite graph with integer edge weights which are drawn uniformly at random from the interval $[0, O(n^c)]$, where $c$ is a constant.*

**Outline**
In Section 2.1 we first give a reduction of the minimum weight bipartite matching problem to minimum cost flow and introduce the linear programming formulations of this problem. Then, in Section 2.2, we present our main algorithm, its main idea and a short proof of Theorem 1. The proofs for the other theorems and the corollary will be shown in Sections 2.3 and 2.4. Finally, in Section 3, we give a short overview of the computational results.

## 2 Algorithm

We first give a reduction of the minimum weight matching problem to the minimum cost flow problem; then we present the linear programming formulations for this problem. These are useful as the dual variables and the objective functions are used in the algorithms and in the proofs. The general algorithm will be described in Section 2.2, where we also give a proof of Theorem 1. In Sections 2.3 and 2.4, we will present the algorithms and the necessary proofs for uniform edge weights and then for the case of integer edge weights.

### 2.1 Reduction to Minimum Cost Flow

Following the ideas of Edmonds and Karp [4] and Goldberg and Kennedy [11], we give a reduction of the minimum weight matching problem for bipartite graphs to the minimum cost flow problem.

A network $N$ is a set of nodes $V$ and a subset $A$ of the ordered pairs $(u, v) \in V \times V$, $u \neq v$, called the arcs. There are two special nodes $s$ and $t$ called *source* and *sink*. A network has a special return arc from $t$ to $s$.

With every arc $(u, v)$ a nonnegative cost $w(u, v)$ and a nonnegative capacity $c(u, v)$ is associated. A flow is a nonnegative function $f : A \to \mathbb{R}_+$ such that

1. $f(u, v) \leq c(u, v) \quad \forall (u, v) \in A$
2. $\sum_{v:(u,v)\in A} f(u, v) = \sum_{v:(v,u)\in A} f(v, u) \quad \forall u \in V$

A maximum flow is a flow in $N$ such that $f(t, s)$ is maximum. The cost of a flow is $\sum_{(u,v)\in A} f(u, v)w(u, v)$. In the minimum cost flow problem we seek for a maximum flow of minimum cost.

To reduce the minimum weight bipartite matching problem to the minimum cost flow problem we introduce two new nodes $s$ and $t$. $s$ is connected by an arc $(s, v)$ with capacity $c(s, v) = 1$ and weight $-n \cdot \max_{e\in E}(\{w(e)\})$ to every node $v \in V_1$ and every node $v \in V_2$ is connected to $t$ by an arc $(v, t)$ with capacity $c(v, t) = 1$ and weight $w(v, t) = 0$. We direct every edge in the original graph from $V_1$ to $V_2$ with capacity 1 and the original weight. The arc $(t, s)$ has weight 0 and capacity $\infty$. Clearly an integral minimum cost flow in this graph corresponds to a minimum weight matching in the original bipartite graph. The large negative costs on the outgoing edges of $s$ assure that there will be maximum flow on these edges and therefore every node will be matched. The minimum cost flow problem can be formulated as a linear program as follows:

$$\min \sum_{(u,v)\in A} f(u, v)w(u, v)$$

$$\forall u \in V : \sum_{v:(u,v)\in A} f(u, v) - \sum_{v:(v,u)\in A} f(v, u) = 0$$

$$\forall (u, v) \in A : 0 \leq f(u, v) \leq c(u, v)$$

The associated dual program is:

$$\max - \sum_{(u,v)\in A} c(u, v)z(u, v)$$

$$\forall (u, v) \in A : p(v) - p(u) - z(u, v) \leq w(u, v)$$

$$\forall (u, v) \in A : z(u, v) \geq 0$$

Let $\bar{w}(u, v) = w(u, v) + p(u) - p(v)$ be the reduced weight of edge $(u, v)$. For the dual variables $z(u, v)$ we have the two constraints $z(u, v) \geq 0$ and $z(u, v) \geq -\bar{w}(u, v)$. As $z(u, v)$ has a negative coefficient in the dual objective function, $z(u, v)$ will be chosen as small as possible, i.e.:

$$z(u, v) = \max\{0, -\bar{w}(u, v)\} \tag{1}$$

Thus the dual variables $z(u, v)$ are determined by the dual variables $p(u)$ and it is sufficient to compute the "potentials" $p(u)$.

Now by complementary slackness conditions we have

$$f(u, v) < c(u, v) \Rightarrow z(u, v) = 0 \Rightarrow p(v) - p(u) \leq w(u, v), \tag{2}$$

and

$$\begin{aligned} f(u,v) > 0 \Rightarrow \bar{w}(u,v) = -z(u,v) \leq 0 \\ \Rightarrow -\bar{w}(u,v) \geq 0 \Rightarrow p(v) - p(u) - w(u,v) \geq 0. \end{aligned} \tag{3}$$

### 2.2 General Algorithm

Algorithm 2.2 keeps for every node $O(\log n)$ edges. On this subset of the edges we run the algorithm MATCHING. For MATCHING we use the algorithm by Fredman and Tarjan [6] which improves the algorithm of Edmonds and Karp [4] for sparse graphs by using a Fibonacci-Heap for the shortest paths computations. The algorithm uses the reduction to minimum cost flow (see Section 2.1) and returns a minimum weight matching and the potentials $p(u)$ of the dual solution of the linear program. The running time of the algorithm is $O(n(m + n \log n))$.

**Algorithm 2.2** Minimum Weight Matching

1: procedure MIN-WEIGHT-MATCHING $(V, E, c, w)$;
2: $E_1 = \{e = (v, w) | e \in E \wedge e$ is one of the $c \cdot log(n)$ smallest edges of $v$ or $w\}$
3: $E_2 = \{c' n \log n$ uniformly at random chosen edges of $E\}$
4: $E' = E_1 \cup E_2$
5: $(M, p) =$ MATCHING$(V, E', w)$
6: return $(M, p)$

To guarantee that the pruned edge set $E'$ contains w.h.p. a *perfect* matching, we make use of a result of Erdős and Rényi [5] which is stated in Theorem 4.

**Theorem 4 (Erdős and Rényi [5]).**
*Let $G(n, n, p)$ be a random bipartite graph on $2n$ nodes with edge probability $p$. Then*

$$\Pr\left(G(n,n,p) \text{ has a perfect matching}\right) \rightarrow \begin{cases} 0 & \text{if } np - \log n \rightarrow -\infty \\ e^{-2e^{-c}} & \text{if } np - \log n \rightarrow c \\ 1 & \text{if } np - \log n \rightarrow \infty \end{cases}$$

Theorem 4 guarantees that by adding uniformly at random $c' n \log n$ additional edges (see Algorithm 2.2), where $c' > 1$, to the pruned edge set $E'$ the graph contains w.h.p. a perfect matching.

If the dual solution does not fulfil the complementary slackness conditions for the original edge set, we compute the value of the dual objective function and use the difference to the computed primal objective as an upper bound on the error. The dual objective for the original edge set can be easily computed from the potentials by using Equation (1) and the primal objective is obtained from the returned matching. The objective of a dual feasible solution is always a lower bound for the optimum of the primal, thus the difference is an upper bound on the error.

*Proof (Theorem 1).* By pruning the edge set we obtain a set of cardinality $|E_1| = O(n \log n)$ in time $O(n^2 \log n)$. The randomly chosen edge set $E_2$ has cardinality $|E_2| = O(n \log n)$. Thus $|E'| = O(n \log n)$ and the running time of MATCHING is $O(n(|E'| + n \log n)) = O(n^2 \log n)$. By computing the dual objective value for the original edge set we obtain an upper bound on the error, which is also returned. The constant $c$ controls the quality of the computed solution. Good indications for the choice of $c$ are given in the next section. For computational experiences see Section 3.

### 2.3 Uniform Edge Weights

Now we consider a complete bipartite graph where the edge weights are generated according to the uniform distribution. Frieze and Sorkin consider in [7] such graphs. For a complete bipartite graph $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2|$ with uniform edge weights $w(u, v)$ and for any perfect matching $\pi$ the directed graph $D_{G,\pi}$ on $V_1$, $V_2$ has the edge set $\overrightarrow{E} = \overrightarrow{E}_\pi \cup \overrightarrow{E}_1 \cup \overrightarrow{E}_2$ where

$$\overrightarrow{E}_\pi = \{(y, x) | x \in V_1, y \in V_2, y = \pi(x)\}$$
$$\overrightarrow{E}_1 = \{(x, y) | x \in V_1, y \in V_2, (x, y) \text{ is one of the 40 shortest edges out of } x\}$$
$$\overrightarrow{E}_2 = \{(x, y) | x \in V_1, y \in V_2, (x, y) \text{ is one of the 40 shortest edges into } y\}.$$

That is all matching edges are oriented backwards and for every node $x \in V_1$ the 40 smallest arcs out of $x$ and for every $y \in V_2$ the 40 smallest arcs into $y$ are added. Let the weight in $D_{G,\pi}$ for the forward edges be $w(u, v)$ and for the backward edges (the matched edges) be - $w(u, v)$.

Frieze and Sorkin prove the following Lemma.

**Lemma 1 ([7] Lemma 7).** *Let $G = (V_1 \cup V_2, E)$ be a complete bipartite graph with $|V_1| = |V_2|$ and uniform $[0, 1]$ edge weights $w(u, v)$. For any perfect matching $\pi$ the probability that the weighted diameter of $D_{G,\pi}$ is greater than $2\zeta \frac{\ln n}{n}$ is at most $o(n^{-1})$ for all $\zeta \geq 24.3$. Moreover, for each pair of nodes there exists a path of weight less than $2\zeta \frac{\ln n}{n}$ consisting of at most $3 \log_4 n$ edges.*

The constants for this result are not best possible (compare also Section 3). Using this lemma Frieze and Sorkin prove the following bound on the maximum edge weight in a minimum weight perfect matching.

**Lemma 2 ([7] Theorem 2).** *Let $\zeta \geq 24.3$. The maximum edge weight $C_{\max}$ in a minimum weight perfect matching in a complete bipartite graph with uniform $[0, 1]$ edge weights is with probability $1 - o(n^{-1})$ smaller than $2\zeta \frac{\ln n}{n}$.*

The idea of the algorithm for Theorem 2 is to prune away heavy edges and run the matching algorithm on the smaller edge set. To prove Theorem 2 we need the following lemma about the dual solution for the pruned edge set.

**Lemma 3.** *Let $\zeta \geq 24.3$. The optimal dual solution computed by the matching algorithm on the set of edges containing only edges of weight smaller than $2\zeta \frac{\ln n}{n}$*

*is with probability* $1-o(n^{-1})$ *also an optimal solution for the dual of the original graph.*

*Proof.* The dual solution is considered not optimal if one of the complementary slackness conditions on the complete edge set is violated. This can only happen for unmatched edges which have not been in the smaller edge set. We show that with probability $1-o(n^{-1})$ no such edge exists. From the complementary slackness conditions we have for $(u,v) \in V_1 \times V_2$ that $p(u)-p(v) \leq -w(u,v)$ for the matched edges (Equation (3)) and $p(v)-p(u) \leq w(u,v)$ for the unmatched edges (Equation (2)).

For $n$ such that $2\zeta \ln n \geq 40$, the graph consisting of all nodes and the 40 smallest edges of every node is contained in the pruned edge set. Therefore by directing again the computed matched edges in the pruned edge set from $V_2$ to $V_1$ and setting their weights $\tilde{w}(v,u) := -w(u,v)$ and directing all other edges from $V_1$ to $V_2$ with weight $\tilde{w}(u,v) := w(u,v)$, we have by Lemma 1: for every pair $(u,v) \in V_1 \times V_2$ with probability $1-o(n^{-1})$ there exists a path in $D_{G,\pi}$ from $u$ to $v$ of weight less than $2\zeta\frac{\ln n}{n}$. Summing over the complementary slackness conditions for the edges of this path $P$ gives:

$$\begin{aligned} p(v)-p(u) &= \sum_{(i,j)\in P} (p(j)-p(i)) \\ &= \sum_{(i,j)\in P \wedge (i,j)\in V_1\times V_2} (p(j)-p(i)) + \sum_{(j,i)\in P \wedge (j,i)\in V_2\times V_1} (p(i)-p(j)) \\ &\leq \sum_{(i,j)\in P} \tilde{w}(i,j) \leq 2\zeta\frac{\ln n}{n} \end{aligned} \tag{4}$$

Thus, the complementary slackness condition $p(v)-p(u) \leq w(u,v)$ can only be violated if $w(u,v)$ is smaller than $2\zeta\frac{\ln n}{n}$ and all these edges are contained in the pruned edge set. Thus the check for optimality succeeds with probability $1-o(n^{-1})$.

*Proof (Theorem 2).* If we keep in Algorithm 2.2 all edges of weight $\leq 2\zeta \ln n/n$ in $E_1$, by Lemma 2 this graph still contains with probability $1-o(n^{-1})$ an optimal matching. The pruned edge set is w.h.p. of cardinality $O(n \ln n)$, thus Algorithm 2.2 returns with probability $1-o(n^{-1})$ a minimum weight matching in time $O(n^2 \log n)$.

The complementary slackness conditions may not hold for the complete edge set, but by Lemma 3 we pass the test of the complementary slackness conditions with probability $1-o(n^{-1})$. Thus, with probability $1-o(n^{-1})$ the algorithm recognises and returns an optimal matching.

We can now easily deduce Corollary 1:

*Proof (Corollary 1).* After running the matching algorithm in $O(n^2 \log n)$ time, checking perfectness of the matching and the complementary slackness conditions takes $O(n^2)$ time. This fails with probability $o(n^{-1})$ and in this case we run

the matching algorithm on the original edge set to compute an optimal perfect matching. Hence, the combined algorithm computes an optimal perfect matching in expected running time $O(n^2 \log n) + o(n^{-1})O(n^3) = O(n^2 \log n)$ where the last term counts for the unlikely cases where the solution on the pruned edge set was not optimal.

### 2.4 Integer Edge Weights

If the edge weights are integers in the range $[0, \ldots, C]$, we can use a fast scaling algorithm with global price updates of Goldberg and Kennedy [11] for the MATCHING algorithm in Algorithm 2.2. Then MATCHING has running time $O(\sqrt{n}m \log(nC))$. The algorithm has for polynomial $C$ that is $C \leq n^c$ for some constant $c$ on the pruned edge set of size $O(n \log^2 n)$ a running time of $O(n^{1.5} \log^2(n) \log(nn^c)) = o(n^2)$. Thus the overall running time of the above algorithms will be dominated by the pruning and the optimality testing (or computation of the upper bound of the achieved approximation factor) which takes time $O(n^2)$. Small $C$ have to be treated in a special way, for details see the proof of Theorem 3. In this way, the algorithm of Theorem 1 can be implemented in running time $O(n^2)$ if the edge weights are integers.

The algorithm of Theorem 2 can be implemented in running time $O(n^2)$ too, if the edge weights are integers drawn uniformly at random from the interval $[0, C \leq n^c]$, where $c$ is a constant. The idea is that by multiplying all edge weights by $1/C$ they are approximately uniformly distributed in $[0, 1]$ (for details see proof of Theorem 3).

*Proof (Theorem 3).* Suppose we have continuous uniformly distributed edge weights $w_c(u, v) \in [0, C + 1)$. Then we know by Lemma 1 that for any perfect matching $\pi$ for each pair of nodes $(u, v)$ and large enough $\zeta$ there exists w.h.p. a path in $D_{G,\pi}$ from $u$ to $v$ of weight $\zeta \frac{\ln n}{n}(C + 1)$ where the path consists of at most $3 \log_4 n$ edges. Let $w(u, v) = \lfloor w_c(u, v) \rfloor$ and $d(u, v) = w_c(u, v) - w(u, v)$ then $w(u, v)$ are integer random variables distributed uniformly on the interval $[0, C]$ and $d(u, v)$ are continuous $[0, 1]$ random variables.

First we consider $C \geq \frac{n}{2 \ln n}$. We discard all edges with weight greater than $\lceil \zeta' \frac{\ln n}{n}(C + 1) \rceil$, where $\zeta' = \Theta(\ln n)$ is large enough. On the remaining edges a minimum weight perfect matching $\pi$ is computed using the integer weights $w$. By Lemma 1 we know that for this matching $\pi$ for every pair $(u, v) \in V_1 \times V_2$ we have w.h.p. a path $u = x_1, y_1, \ldots, x_k, y_k = v$ in $D_{G,\pi}$ with $k \leq 3 \log_4 n$ such that the weight of the path on $w_c$ is for a constant $\zeta$ smaller than $\zeta \frac{\ln n}{n}(C + 1)$. Showing that the probability that the sum over the differences $d(\cdot, \cdot)$ on such a path is $> (\zeta' - \zeta) \frac{\ln n}{n}(C + 1)$ is $o(n^{-2})$ gives that w.h.p. the paths with integer weights have weight $\leq \zeta' \frac{\ln n}{n}(C + 1)$ as there are $n^2$ pairs of nodes. We set $\zeta'' := \zeta' - \zeta$ which is still $\Theta(\log n)$.

As the backward edges are negative and the forward edges are positive we bound the probability that one of the respective sums deviates more than $\frac{\zeta''}{2} \frac{\ln n}{n}(C + 1)$ from its mean $\mu = k/2$. We use the following Chernoff bound.

Suppose $0 \leq X_i \leq 1$ independent random variables, $X = \sum_{i=0}^{k} X_i$, and $\mu = E[X]$ then $\Pr(|X - \mu| \geq kt) \leq 2e^{-2kt^2}$ for all $t \geq 0$. Hence,

$$\begin{aligned}
&\Pr\left(|\sum_{i=1}^{k} d(x_i, y_i) - \mu| > \zeta''/2 \cdot \frac{\ln n}{n}(C+1)\right) \\
&= \Pr\left(|\sum_{i=1}^{k} d(x_i, y_i) - k/2| > k\left(\zeta'' \frac{\ln n}{2kn}(C+1)\right)\right) \\
&\leq 2e^{-2k\left(\frac{\zeta'' \ln n \cdot C}{2kn}\right)^2} \leq 2e^{-2k\left(\frac{\zeta''}{4k}\right)^2} \leq 2e^{-\frac{\zeta''^2}{24 \log_4 n}} = o(n^{-2}),
\end{aligned}$$

as $k \leq 3\log_4 n$, $C \geq \frac{n}{2\ln n}$ and for large enough $\zeta'$ (such that $\zeta''$ is large). For the backward edges the same probability is calculated in an analogue way and it follows that we have w.h.p. for the computed matching $\pi$ and all pairs $(u, v) \in V_1 \times V_2$ a path in $D_{G,\pi}$ of weight at most $\zeta' \frac{\ln n}{n}(C+1)$ on the integer weights. Therefore by the same argument as in the proof of Lemma 3 (see Equation (4)) the dual of the solution on the edge set consisting of the edges with weight $\leq \zeta' \frac{\ln n}{n}(C+1)$ will w.h.p. fulfil the complementary slackness conditions on the complete edge set and therefore the solution will be optimal.

This set of small edges is w.h.p. of size $m = O(n \log^2 n)$, thus the running time of the integer matching algorithm is $O(\sqrt{n} m \log(nC)) = o(n^2)$ and the overall running time is dominated by the pruning of the edge set and the optimality check which is $O(n^2)$.

Now let $C \leq \frac{n}{2\ln n}$. Then the edges of weight 0 form a random graph with edge probability $p \geq 2\ln(n)/n$. Again, by Theorem 4 this graph contains w.h.p. a perfect matching. Therefore a graph with edge weights from such an interval has w.h.p. a matching of weight 0. Thus, in this case we are only looking for a maximum cardinality matching within the graph consisting only of edges with weight 0. If there are more than $O(n \log n)$ such edges, we choose $O(n \log n)$ of them uniformly at random. A maximum cardinality matching in this graph can be computed in time $O(\sqrt{n}m)$ for example by the algorithm of Hopcroft and Karp [13], which leads to a running time of $O(n^{1.5} \log n)$ for the pruned edge set. In general for denser graphs a maximum cardinality matching can be computed in $O(n^\omega)$ where $\omega$ is the exponent of the best known matrix multiplication algorithm which was shown by Mucha and Sankowski [19].

## 3 Computational Results

### 3.1 Real World

On real-world data ($n$ ranging from about 5000 to 20000 and with integer weights) that was provided by our partner our algorithms performed quite well. As the data was highly structured it was essential to introduce the randomly chosen edges of the edge set $E_2$ of Algorithm 2.2.

The constant $c'$ of Algorithm 2.2 was always set to a small value, i.e. $c' \approx 4$. For obtaining a solution within 5% of the optimum setting $c = 15$ was sufficient

for all provided test inputs. For the highly structured data sets $c$ had to be set to $c > 120$ to achieve an approximation within 3% of the optimum.

### 3.2 Random Edge Weights

On test cases with edge weights generated according to the uniform distribution experiments showed that a cut off value of $c \cdot \log n/n$ with $c = 4$ was sufficient instead of the $c = 48.6$ which was obtained from Lemma 1.

**Table 1.** Testing the algorithm on different input graphs with edge weights drawn uniformly at random from the interval $[0, n^2]$

| $n$ | 10 | | 100 | | 500 | | 800 | | 1000 | | 1200 | | 1500 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Opt | Check | Opt | Check | Opt | Check | Opt | Check | Opt | Check | Opt | Check | Opt | Check |
| $c = 2$ | 100 | 100 | 100 | 100 | 98 | 98 | 96 | 92 | 98 | 98 | 100 | 100 | 100 | 100 |
| $c = 3$ | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 98 | 100 | 100 |
| $c = 4$ | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |

Table 1 lists for $c \in \{2, 3, 4\}$ and $n$ from 10 to 1500 the percentage of having found an optimal matching (column Opt) and the percentage of passing the complementary slackness checks on the complete input graph with the dual variables computed on the pruned edge set (column Check). For every $n$ the algorithm was run on 50 input graphs. The edge weights were integers drawn uniformly at random from the interval $[0, n^2]$.

## References

1. R. Ahuja, T. Magnanti, and J. Orlin. *Network flows.* Prentice Hall Inc., Englewood Cliffs, NJ, 1993. Theory, algorithms, and applications.
2. S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(4):509–522, 2002.
3. Y. Cheng, V. Wu, R. Collins, A. Hanson, and E. Riseman. Maximum-weight bipartite matching technique and its application in image feature matching. In *SPIE Conference on Visual Communication and Image Processing*, 1996.
4. J. Edmonds and R. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In *Combinatorial Structures and their Applications (Proc. Calgary Internat. Conf., Calgary, Alta., 1969)*, pages 93–96. Gordon and Breach, New York, 1970.
5. P. Erdős and A. Rényi. On random matrices. *Magyar Tud. Akad. Mat. Kutató Int. Közl*, 8:455–461, 1964.
6. M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34(3):596–615, 1987.
7. A. Frieze and G. Sorkin. The probabilistic relationship between the assignment and asymmetric traveling salesman problems. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (Washington, DC, 2001)*, pages 652–660, Philadelphia, PA, 2001.

8. H. Gabow and R. Tarjan. Almost-optimal speed-ups of algorithms for matching and related problems. In *Proceedings 20th Annual ACM Symposium on Theory of Computing (Chicago, IL)*, pages 514–527, New York, 1988.
9. H.. Gabow and R. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.
10. M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.
11. A. Goldberg and R. Kennedy. Global price updates help. *SIAM J. Discrete Math.*, 10(4):551–572, 1997.
12. A. Goldberg, . Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *J. Algorithms*, 14(2):180–213, 1993.
13. J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.
14. C. Huang, Y. Chen, Y. Lin, and Y. Hsu. Data path allocation based on bipartite weighted matching. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 499–504, 1990.
15. D. S. Johnson and C. C. McGeoch (editors). Network flows and matching: First DIMACS implementation challenge. volume 12. AMS, 1993.
16. R. Karp. An algorithm to solve the $m \times n$ assignment problem in expected time $O(mn \log n)$. *Networks*, 10(2):143–152, 1980.
17. H. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.
18. Y. Lin, H. Chang, and Y. Lin. A study on tools and algorithms for 3-d protein structures alignment and comparison. In *Internat. Computer Symposium (ICS'2004)*, pages 1000–1005, 2004.
19. M. Mucha and P. Sankowski. Maximum matchings via gaussian elimination. In *FOCS*, pages 248–255, 2004.
20. J. Orlin and R. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Math. Programming*, 54(1, Ser. A):41–56, 1992.
21. K. Varadarajan and P. Agarwal. Approximation algorithms for bipartite and non-bipartite matching in the plane. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 1999)*, pages 805–814, New York, 1999.
22. M. Wattenhofer and R. Wattenhofer. Fast and simple algorithms for weighted perfect matching, submitted, 2004.

# A Practical Minimal Perfect Hashing Method*

Fabiano C. Botelho[1], Yoshiharu Kohayakawa[2], and Nivio Ziviani[1]

[1] Dept. of Computer Science, Federal Univ. of Minas Gerais, Belo Horizonte, Brazil
{fbotelho, nivio}@dcc.ufmg.br
[2] Dept. of Computer Science, Univ. of São Paulo, São Paulo, Brazil
yoshi@ime.usp.br

**Abstract.** We propose a novel algorithm based on random graphs to construct minimal perfect hash functions $h$. For a set of $n$ keys, our algorithm outputs $h$ in expected time $O(n)$. The evaluation of $h(x)$ requires two memory accesses for any key $x$ and the description of $h$ takes up $1.15n$ words. This improves the space requirement to 55% of a previous minimal perfect hashing scheme due to Czech, Havas and Majewski. A simple heuristic further reduces the space requirement to $0.93n$ words, at the expense of a slightly worse constant in the time complexity. Large scale experimental results are presented.

## 1 Introduction

Suppose $U$ is a universe of *keys*. Let $h : U \to M$ be a *hash function* that maps the keys from $U$ to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$. Let $S \subseteq U$ be a set of $n$ keys from $U$. Given a key $x \in S$, the hash function $h$ computes an integer in $[0, m-1]$ for the storage or retrieval of $x$ in a *hash table*. Hashing methods for *non-static sets* of keys can be used to construct data structures storing $S$ and supporting membership queries "$x \in S$?" in expected time $O(1)$. However, they involve a certain amount of wasted space owing to unused locations in the table and waisted time to resolve collisions when two keys are hashed to the same table location.

For *static sets* of keys it is possible to compute a function to find any key in a table in one probe; such hash functions are called *perfect*. Given a set of keys $S$, we shall say that a hash function $h : U \to M$ is a *perfect hash function* for $S$ if $h$ is an injection on $S$, that is, there are no *collisions* among the keys in $S$: if $x$ and $y$ are in $S$ and $x \neq y$, then $h(x) \neq h(y)$. Since no collisions occur, each key can be retrieved from the table with a single probe. If $m = n$, that is, the table has the same size as $S$, then $h$ is a minimal perfect hash function. Minimal perfect hash functions totally avoid the problem of wasted space and time.

* This work was supported in part by GERINDO Project–grant MCT/CNPq/CT-INFO 552.087/02-5, CAPES/PROF Scholarship (Fabiano C. Botelho), FAPESP Proj. Tem. 03/09925-5 and CNPq Grant 30.0334/93-1 (Yoshiharu Kohayakawa), and CNPq Grant 30.5237/02-0 (Nivio Ziviani).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 488–500, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Minimal perfect hash functions are widely used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, universal resource locations (URLs) in Web search engines, or item sets in data mining techniques.

The aim of this paper is to describe a new way of constructing minimal perfect hash functions. Our algorithm shares several features with the one due to Czech, Havas and Majewski [4]. In particular, our algorithm is also based on the generation of random graphs $G = (V, E)$, where $E$ is in one-to-one correspondence with the key set $S$ for which we wish to generate the hash function. The two main differences between our algorithm and theirs are as follows: $(i)$ we generate random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, where $c = 1.15$, and hence $G$ contains cycles with high probability, while they generate *acyclic* random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, with a greater number of vertices: $|V| \geq 2.09n$; $(ii)$ they generate order preserving minimal perfect hash functions while our algorithm does not preserve order (a perfect hash function $h$ is *order preserving* if the keys in $S$ are arranged in some given order and $h$ preserves this order in the hash table). Thus, our algorithm improves the space requirement at the expense of generating functions that are not order preserving.

Our algorithm is efficient and may be tuned to yield a function $h$ with a very economical description. As the algorithm in [4], our algorithm produces $h$ in $O(n)$ expected time for a set of $n$ keys. The description of $h$ requires $1.15n$ computer words, and evaluating $h(x)$ requires two accesses to an array of $1.15n$ integers. We further derive a heuristic that improves the space requirement from $1.15n$ words down to $0.93n$ words. Our scheme is very practical: to generate a minimal perfect hash function for a collection of 100 million universe resource locations (URLs), each 63 bytes long on average, our algorithm running on a commodity PC takes 811 seconds on average.

## 2 Related Work

Czech, Havas and Majewski [5] provide a comprehensive survey of the most important theoretical results on perfect hashing. In the following, we review some of those results.

Fredman, Komlós and Szemerédi [10] showed that it is possible to construct space efficient perfect hash functions that can be evaluated in constant time with table sizes that are linear in the number of keys: $m = O(n)$. In their model of computation, an element of the universe $U$ fits into one machine word, and arithmetic operations and memory accesses have unit cost. Randomized algorithms in the FKS model can construct a perfect hash function in expected time $O(n)$: this is the case of our algorithm and the works in [4, 14].

Many methods for generating minimal perfect hash functions use a *mapping*, *ordering* and *searching* (MOS) approach, a description coined by Fox, Chen and Heath [9]. In the MOS approach, the construction of a minimal perfect hash function is accomplished in three steps. First, the mapping step transforms the key set from the original universe to a new universe. Second, the ordering step places

the keys in a sequential order that determines the order in which hash values are assigned to keys. Third, the searching step attempts to assign hash values to the keys. Our algorithm and the algorithm presented in [4] use the MOS approach.

Pagh [14] proposed a family of randomized algorithms for constructing minimal perfect hash functions. The form of the resulting function is $h(x) = (f(x) + d_{g(x)}) \bmod n$, where $f$ and $g$ are universal hash functions and $d$ is a set of displacement values to resolve collisions that are caused by the function $f$. Pagh identified a set of conditions concerning $f$ and $g$ and showed that if these conditions are satisfied, then a minimal perfect hash function can be computed in expected time $O(n)$ and stored in $(2+\epsilon)n$ computer words. Dietzfelbinger and Hagerup [6] improved [14], reducing from $(2+\epsilon)n$ to $(1+\epsilon)n$ the number of computer words required to store the function, but in their approach $f$ and $g$ must be chosen from a class of hash functions that meet additional requirements. Differently from the works in [14, 6], our algorithm uses two universal hash functions $h_1$ and $h_2$ randomly selected from a class of universal hash functions that do not need to meet any additional requirements.

The work in [4] presents an efficient and practical algorithm for generating order preserving minimal perfect hash functions. Their method involves the generation of acyclic random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = n$, with $c \geq 2.09$. They showed that an order preserving minimal perfect hash function can be found in optimal time if $G$ is acyclic. To generate an acyclic graph, two vertices $h_1(x)$ and $h_2(x)$ are computed for each key $x \in S$. Thus, each set $S$ has a corresponding graph $G = (V, E)$, where $V = \{0, 1, \ldots, t\}$ and $E = \big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. In order to guarantee the acyclicity of $G$, the algorithm repeatedly selects $h_1$ and $h_2$ from a family of universal hash functions until the corresponding graph is acyclic. Havas et al. [11] proved that if $|V(G)| = cn$ and $c > 2$, then the probability that $G$ is acyclic is $p = e^{1/c}\sqrt{(c-2)/c}$. For $c = 2.09$, this probability is $p \simeq 0.342$, and the expected number of iterations to obtain an acyclic graph is $1/p \simeq 2.92$.

## 3 The Algorithm

Let us show how the minimal perfect hash function $h$ will be constructed. We make use of two auxiliary random functions $h_1$ and $h_2 : U \to V$, where $V = [0, t-1]$ for some suitably chosen integer $t = cn$, where $n = |S|$. We build a random graph $G = G(h_1, h_2)$ on $V$, whose edge set is $\big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. There is an edge in $G$ for each key in the set of keys $S$.

In what follows, we shall be interested in the *2-core* of the random graph $G$, that is, the maximal subgraph of $G$ with minimal degree at least 2 (see, e.g., [1, 12]). Because of its importance in our context, we call the 2-core the *critical* subgraph of $G$ and denote it by $G_{\text{crit}}$. The vertices and edges in $G_{\text{crit}}$ are said to be *critical*. We let $V_{\text{crit}} = V(G_{\text{crit}})$ and $E_{\text{crit}} = E(G_{\text{crit}})$. Moreover, we let $V_{\text{ncrit}} = V - V_{\text{crit}}$ be the set of *non-critical* vertices in $G$. We also let $V_{\text{scrit}} \subseteq V_{\text{crit}}$ be the set of all critical vertices that have at least one non-critical vertex as a neighbour. Let $E_{\text{ncrit}} = E(G) - E_{\text{crit}}$ be the set of *non-critical* edges in $G$. Finally, we let $G_{\text{ncrit}} = (V_{\text{ncrit}} \cup V_{\text{scrit}}, E_{\text{ncrit}})$ be the *non-critical* subgraph of $G$.

```
procedure GenerateMPHF (S, g)
    Mapping (S, G);
    Ordering (G, G_crit, G_ncrit);
    Searching (G, G_crit, G_ncrit, g);
```

**Fig. 1.** Main steps of the algorithm for constructing a minimal perfect hash function

The non-critical subgraph $G_{\text{ncrit}}$ corresponds to the "acyclic part" of $G$. We have $G = G_{\text{crit}} \cup G_{\text{ncrit}}$.

We then construct a suitable labelling $g : V \to \mathbb{Z}$ of the vertices of $G$: we choose $g(v)$ for each $v \in V(G)$ in such a way that $h(x) = g(h_1(x)) + g(h_2(x))$ $(x \in S)$ is a minimal perfect hash function for $S$. We will see later on that this labelling $g$ can be found in linear time if the number of edges in $G_{\text{crit}}$ is at most $\frac{1}{2}|E(G)|$.

Figure 1 presents a pseudo code for the algorithm. The procedure GenerateMPHF $(S, g)$ receives as input the set of keys $S$ and produces the labelling $g$. The method uses a mapping, ordering and searching approach. We now describe each step.

### 3.1 Mapping Step

The procedure Mapping $(S, G)$ receives as input the set of keys $S$ and generates the random graph $G = G(h_1, h_2)$, by generating two auxiliary functions $h_1$, $h_2 : U \to [0, t-1]$.

The functions $h_1$ and $h_2$ are constructed as follows. We impose some upper bound $L$ on the lengths of the keys in $S$. To define $h_j$ $(j = 1,2)$, we generate an $L \times \Sigma$ table of random integers $\text{table}_j$. For a key $x \in S$ of length $|x| \leq L$ and $j \in \{1, 2\}$, we let

$$h_j(x) = \Big( \textstyle\sum_{i=1}^{|x|} \text{table}_j[i, x[i]] \Big) \bmod t.$$

The random graph $G = G(h_1, h_2)$ has vertex set $V = [0, t-1]$ and edge set $\big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. We need $G$ to be simple, i.e., $G$ should have neither loops nor multiple edges. A loop occurs when $h_1(x) = h_2(x)$ for some $x \in S$. We solve this in an ad hoc manner: we simply let $h_2(x) = (2h_1(x) + 1) \bmod t$ in this case. If we still find a loop after this, we generate another pair $(h_1, h_2)$. When a multiple edge occurs we abort and generate a new pair $(h_1, h_2)$.

**Analysis of the Mapping Step.** We start by discussing some facts on random graphs. Let $G = (V, E)$ with $|V| = t$ and $|E| = n$ be a random graph in the uniform model $\mathcal{G}(t, n)$, the model in which all the $\binom{\binom{t}{2}}{n}$ graphs on $V$ with $n$ edges are equiprobable. The study of $\mathcal{G}(t, n)$ goes back to the classical work of Erdős and Rényi [7, 8, 13] (for a modern treatment, see [1, 12]). Let $d = 2n/t$ be the average degree of $G$. It is well known that, if $d > 1$, or, equivalently, if $c < 2$ (recall that we have $t = cn$), then, almost every $G$ contains[1] a "giant"

[1] As is usual in the theory of random graphs, we use the terms 'almost every' and 'almost surely' to mean 'with probability tending to 1 as $t \to \infty$'.

component of order $(1+o(1))bt$, where $b = 1-T/d$, and $0 < T < 1$ is the unique solution to the equation $Te^{-T} = de^{-d}$. Moreover, all the other components of $G$ have $O(\log t)$ vertices. Also, the number of vertices in the 2-core of $G$ (the maximal subgraph of $G$ with minimal degree at least 2) that do not belong to the giant component is $o(t)$ almost surely.

Pittel and Wormald [15] present detailed results for the 2-core of the giant component of the random graph $G$. Since table$_j$ ($j \in \{1,2\}$) are random, $G = G(h_1, h_2)$ is a random graph. In what follows, we work under the hypothesis that $G = G(h_1, h_2)$ is drawn from $\mathcal{G}(t,n)$. Thus, following [15], the number of vertices of $G_{\text{crit}}$ is

$$|V(G_{\text{crit}})| = (1+o(1))(1-T)bt \tag{1}$$

almost surely. Moreover, the number of edges in this 2-core is

$$|E(G_{\text{crit}})| = (1+o(1))\Big((1-T)b + b(d+T-2)/2\Big)t \tag{2}$$

almost surely. Let $d_{\text{crit}} = 2|E(G_{\text{crit}})|/|V(G_{\text{crit}})|$ be the average degree of $G_{\text{crit}}$. We are interested in the case in which $d_{\text{crit}}$ is a constant.

As mentioned before, for us to find the labelling $g : V \to \mathbb{Z}$ of the vertices of $G = G(h_1, h_2)$ in linear time, we require that $|E(G_{\text{crit}})| \le \frac{1}{2}|E(G)| = \frac{1}{2}|S| = n/2$. The crucial step now is to determine the value of $c$ (in $t = cn$) to obtain a random graph $G = G_{\text{crit}} \cup G_{\text{ncrit}}$ with $|E(G_{\text{crit}})| \le \frac{1}{2}|E(G)|$.

Table 3.1 gives some values for $|V(G_{\text{crit}})|$ and $|E(G_{\text{crit}})|$ using Eqs (1) and (2). The theoretical value for $c$ is around 1.152, which is remarkably close to the empirical results presented in Table 3.1. In this table, generated from real data, the probability $P_{|E(G_{\text{crit}})|}$ that $|E(G_{\text{crit}})| \le \frac{1}{2}|E(G)|$ tends to 0 when $c < 1.15$ and it tends to 1 when $c \ge 1.15$ and $n$ increases. We found this match between the empirical and the theoretical results most pleasant, and this is why we consider that this random graph, conditioned on being simple, strongly resembles the random graph from the uniform model $\mathcal{G}(t,n)$.

We now briefly argue that the expected number of iterations to obtain a simple graph $G = G(h_1, h_2)$ is constant for $t = cn$ and $c = 1.15$. Let $p$ be the probability of generating a random graph $G$ without loops and without multiple edges. If $p$ is bounded from below by some positive constant, then we are done, because the expected number of iterations to obtain such a graph is then $1/p = O(1)$. To estimate $p$, we estimate the probability of obtaining $n$ *distinct* objects when we independently draw $n$ objects from a universe of cardinality $\binom{t}{2} =$

**Table 1.** Determining the $c$ value theoretically

| $d$ | $T$ | $b$ | $\lvert V(G_{\text{crit}})\rvert$ | $\lvert E(G_{\text{crit}})\rvert$ | $c$ |
|---|---|---|---|---|---|
| 1.734 | 0.510 | 0.706 | $0.399n$ | $0.498n$ | 1.153 |
| 1.736 | 0.509 | 0.707 | $0.400n$ | $0.500n$ | 1.152 |
| 1.738 | 0.508 | 0.708 | $0.401n$ | $0.501n$ | 1.151 |
| 1.739 | 0.508 | 0.708 | $0.401n$ | $0.501n$ | 1.150 |
| 1.740 | 0.507 | 0.709 | $0.401n$ | $0.502n$ | 1.149 |

**Table 2.** Probability $P_{|E_{\text{crit}}|}$ that $|E(G_{\text{crit}})| \leq n/2$ for different $c$ values and different number of keys for a collections of URLs

| $c$ | URLs ($n$) | | | | | | |
|---|---|---|---|---|---|---|---|
| | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $2 \times 10^6$ | $3 \times 10^6$ | $4 \times 10^6$ |
| 1.13 | 0.22 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1.14 | 0.35 | 0.15 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 1.15 | 0.46 | 0.55 | 0.65 | 0.87 | 0.95 | 0.97 | 1.00 |
| 1.16 | 0.67 | 0.90 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 1.17 | 0.82 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

$\binom{cn}{2} \sim c^2n^2/2$, with replacement. This latter probability is about $e^{-\binom{n}{2}/\binom{t}{2}}$ for large $n$. As $e^{-\binom{n}{2}/\binom{t}{2}} \to e^{-1/c^2} > 0$ as $n \to \infty$, the expected number of iterations is $e^{1/c^2} = 2.13$ (recall $c = 1.15$). As the expected number of iterations is $O(1)$, the mapping step takes $O(n)$ time.

### 3.2 Ordering Step

The procedure Ordering ($G$, $G_{\text{crit}}$, $G_{\text{ncrit}}$) receives as input the graph $G$ and partitions $G$ into the two subgraphs $G_{\text{crit}}$ and $G_{\text{ncrit}}$, so that $G = G_{\text{crit}} \cup G_{\text{ncrit}}$. For that, the procedure iteratively remove all vertices of degree 1 until done.

Figure 2(a) presents a sample graph with 9 vertices and 8 edges, where the degree of a vertex is shown besides each vertex. Applying the ordering step in this graph, the 5-vertex graph showed in Figure 2(b) is obtained. All vertices with degree 0 are non-critical vertices and the others are critical vertices. In order to determine the vertices in $V_{\text{scrit}}$ we collect all vertices $v \in V(G_{\text{crit}})$ with at least one vertex $u$ that is in $\text{Adj}(v)$ and in $V(G_{\text{ncrit}})$, as the vertex 8 in Figure 2(b).

**Analysis of the Ordering Step.** The time complexity of the ordering step is $O(|V(G)|)$ (see [5]). As $|V(G)| = t = cn$, the ordering step takes $O(n)$ time.

### 3.3 Searching Step

In the searching step, the key part is the *perfect assignment problem*: find $g : V(G) \to \mathbb{Z}$ such that the function $h : E(G) \to \mathbb{Z}$ defined by

$$h(e) = g(a) + g(b) \qquad (e = \{a, b\}) \tag{3}$$

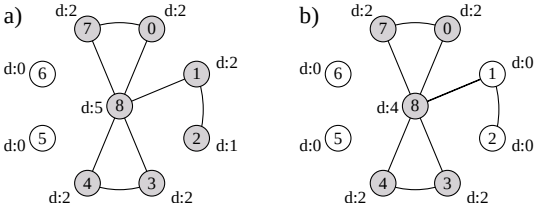



**Fig. 2.** Ordering step for a graph with 9 vertices and 8 edges

is a bijection from $E(G)$ to $[0, n-1]$ (recall $n = |S| = |E(G)|$). We are interested in a labelling $g : V \to \mathbb{Z}$ of the vertices of the graph $G = G(h_1, h_2)$ with the property that if $x$ and $y$ are keys in $S$, then $g(h_1(x))+g(h_2(x)) \neq g(h_1(y))+g(h_2(y))$; that is, if we associate to each edge the sum of the labels on its endpoints, then these values should be all distinct. Moreover, we require that all the sums $g(h_1(x)) + g(h_2(x))$ $(x \in S)$ fall between 0 and $|E(G)| - 1 = n - 1$, so that we have a bijection between $S$ and $[0, n-1]$.

The procedure Searching $(G, G_{\text{crit}}, G_{\text{ncrit}}, g)$ receives as input $G$, $G_{\text{crit}}$, $G_{\text{ncrit}}$ and finds a suitable $\log_2 |V(G)| + 1$ bit value for each vertex $v \in V(G)$, stored in the array $g$. This step is first performed for the vertices in the critical subgraph $G_{\text{crit}}$ of $G$ (the 2-core of $G$) and then it is performed for the vertices in $G_{\text{ncrit}}$ (the non-critical subgraph of $G$ that contains the "acyclic part" of $G$). The reason the assignment of the $g$ values is first performed on the vertices in $G_{\text{crit}}$ is to resolve reassignments as early as possible (such reassignments are consequences of the cycles in $G_{\text{crit}}$ and are depicted hereinafter).

**Assignment of Values to Critical Vertices.** The labels $g(v)$ $(v \in V(G_{\text{crit}}))$ are assigned in increasing order following a greedy strategy where the critical vertices $v$ are considered one at a time, according to a breadth-first search on $G_{\text{crit}}$. If a candidate value $x$ for $g(v)$ is forbidden because setting $g(v) = x$ would create two edges with the same sum, we try $x + 1$ for $g(v)$. This fact is referred to as a *reassignment*.

Let $A_E$ be the set of addresses assigned to edges in $E(G_{\text{crit}})$. Initially $A_E = \emptyset$. Let $x$ be a candidate value for $g(v)$. Initially $x = 0$. Considering the subgraph $G_{\text{crit}}$ in Figure 2(b), a step by step example of the assignment of values to vertices in $G_{\text{crit}}$ is presented in Figure 3. Initially, a vertex $v$ is chosen, the assignment $g(v) = x$ is made and $x$ is set to $x + 1$. For example, suppose that vertex 8 in Figure 3(a) is chosen, the assignment $g(8) = 0$ is made and $x$ is set to 1.

In Figure 3(b), following the adjacency list of vertex 8, the unassigned vertex 0 is reached. At this point, we collect in the temporary variable $Y$ all adjacencies of vertex 0 that have been assigned an $x$ value, and $Y = \{8\}$. Next, for all $u \in Y$, we check if $g(u) + x \notin A_E$. Since $g(8) + 1 = 1 \notin A_E$, then $g(0)$ is set to 1, $x$ is incremented by 1 (now $x = 2$) and $A_E = A_E \cup \{1\} = \{1\}$. Next, vertex 3 is reached, $g(3)$ is set to 2, $x$ is set to 3 and $A_E = A_E \cup \{2\} = \{1, 2\}$. Next, vertex 4

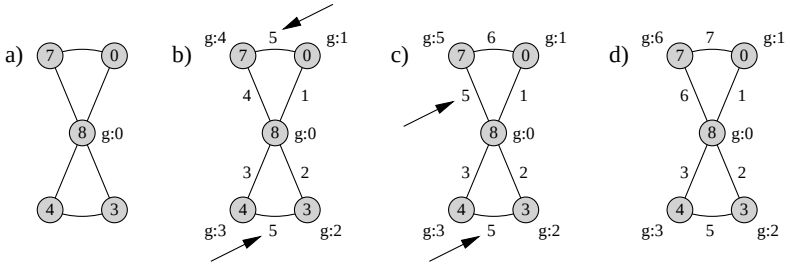



**Fig. 3.** Example of the assignment of values to critical vertices

is reached and $Y = \{3, 8\}$. Since $g(3) + 3 = 5 \notin A_E$ and $g(8) + 3 = 3 \notin A_E$, then $g(4)$ is set to 3, $x$ is set to 4 and $A_E = A_E \cup \{3, 5\} = \{1, 2, 3, 5\}$. Finally, vertex 7 is reached and $Y = \{0, 8\}$. Since $g(0)+4 = 5 \in A_E$, $x$ is incremented by 1 and set to 5, as depicted in Figure 3(c). Since $g(8)+5 = 5 \in A_E$, $x$ is again incremented by 1 and set to 6, as depicted in Figure 3(d). These two reassignments are indicated by the arrows in Figure 3. Since $g(0) + 6 = 7 \notin A_E$ and $g(8) + 6 = 6 \notin A_E$, then $g(7)$ is set to 6 and $A_E = A_E \cup \{6, 7\} = \{1, 2, 3, 5, 6, 7\}$. This finishes the algorithm.

**Assignment of Values to Non- critical Vertices.** As $G_{\text{ncrit}}$ is acyclic, we can impose the order in which addresses are associated with edges in $G_{\text{ncrit}}$, making this step simple to solve by a standard depth first search algorithm. Therefore, in the assignment of values to vertices in $G_{\text{ncrit}}$ we benefit from the unused addresses in the gaps left by the assignment of values to vertices in $G_{\text{crit}}$. For that, we start the depth-first search from the vertices in $V_{\text{scrit}}$ because the $g$ values for these critical vertices have already been assigned and cannot be changed.

Considering the subgraph $G_{\text{ncrit}}$ in Figure 2(b), a step by step example of the assignment of values to vertices in $G_{\text{ncrit}}$ is presented in Figure 4. Figure 4(a) presents the initial state of the algorithm. The critical vertex 8 is the only one that has non-critical neighbours. In the example presented in Figure 3, the addresses $\{0, 4\}$ were not used. So, taking the first unused address 0 and the vertex 1, which is reached from the vertex 8, $g(1)$ is set to $0 - g(8) = 0$, as shown in Figure 4(b). The only vertex that is reached from vertex 1 is vertex 2, so taking the unused address 4 we set $g(2)$ to $4 - g(1) = 4$, as shown in Figure 4(c). This process is repeated until the UnAssignedAddresses list becomes empty.

**Analysis of the Searching Step.** We shall demonstrate that (i) the maximum value assigned to an edge is at most $n-1$ (that is, we generate a minimal perfect hash function), and (ii) the perfect assignment problem (determination of $g$) can be solved in expected time $O(n)$ if the number of edges in $G_{\text{crit}}$ is at most $\frac{1}{2}|E(G)|$.

We focus on the analysis of the assignment of values to critical vertices because the assignment of values to non-critical vertices can be solved in linear time by a depth first search algorithm.

We now define certain complexity measures. Let $I(v)$ be the number of times a candidate value $x$ for $g(v)$ is incremented. Let $N_t$ be the total number of times

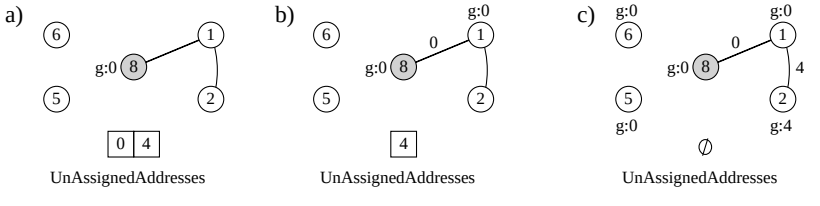



**Fig. 4.** Example of the assignment of values to non-critical vertices

that candidate values $x$ are incremented. Thus, we have $N_t = \sum I(v)$, where the sum is over all $v \in V(G_{\text{crit}})$.

For simplicity, we shall suppose that $G_{\text{crit}}$, the 2-core of $G$, is connected.[2] The fact that every edge is either a tree edge or a back edge (see, e.g., [3]) then implies the following.

**Theorem 1.** *The number of back edges* $N_{\text{bedges}}$ *of* $G = G_{\text{crit}} \cup G_{\text{ncrit}}$ *is given by* $N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1$. □

Our next result concerns the maximal value $A_{\max}$ assigned to an edge $e \in E(G_{\text{crit}})$ after the assignment of $g$ values to critical vertices.

**Theorem 2.** *We have* $A_{\max} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$.

*Proof.* (Sketch) The assignment of $g$ values to critical vertices starts from 0, and each edge $e$ receives the label $h(e)$ as given by Eq. (3). The $g$ value for each vertex $v$ in $V(G_{\text{crit}})$ is assigned only once. A little thought shows that $\max_v g(v) \leq |V(G_{\text{crit}})| - 1 + N_t$, where the maximum is taken over all vertices $v$ in $V(G_{\text{crit}})$. Moreover, two distinct vertices get distinct $g$ values. Hence, $A_{\max} \leq (|V(G_{\text{crit}})| - 1 + N_t) + (|V(G_{\text{crit}})| - 2 + N_t) \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$, as required. □

**Maximal Value Assigned to an Edge.** In this section we present the following conjecture.

*Conjecture 1.* For a random graph $G$ with $|E(G_{\text{crit}})| \leq n/2$ and $|V(G)| = 1.15n$, it is always possible to generate a minimal perfect hash function because the maximal value $A_{\max}$ assigned to an edge $e \in E(G_{\text{crit}})$ is at most $n - 1$.

Let us assume for the moment that $N_t \leq N_{\text{bedges}}$. Then, from Theorems 1 and 2, we have $A_{\max} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t \leq 2|V(G_{\text{crit}})| - 3 + 2N_{\text{bedges}} \leq 2|V(G_{\text{crit}})| - 3 + 2(|E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1) \leq 2|E(G_{\text{crit}})| - 1$. As by hypothesis $|E(G_{\text{crit}})| \leq n/2$, we have $A_{\max} \leq n - 1$, as required.

*In the mathematical analysis of our algorithm, what is left open is a single problem: prove that* $N_t \leq N_{\text{bedges}}$.[3]

We now show experimental evidence that $N_t \leq N_{\text{bedges}}$. Considering Eqs (1) and (2), the expected values for $|V(G_{\text{crit}})|$ and $|E(G_{\text{crit}})|$ for $c = 1.15$ are $0.401n$ and $0.501n$, respectively. From Theorem 1, $N_{\text{bedges}} = 0.501n - 0.401n + 1 = 0.1n + 1$. Table 3 presents the maximal value of $N_t$ obtained during 10,000 executions of the algorithm for different sizes of $S$. The maximal value of $N_t$ was always smaller than $N_{\text{bedges}} = 0.1n + 1$ and tends to $0.059n$ for $n \geq$ 1,000,000.

[2] The number of vertices in $G_{\text{crit}}$ outside the giant component is provably very small for $c = 1.15$; see [1, 12, 15].

[3] Bollobás and Pikhurko [2] have investigated a very close vertex labelling problem for random graphs. However, their interest was on denser random graphs, and it seems that different methods will have to be used to attack the sparser case that we are interested in here.

**Table 3.** The maximal value of $N_t$ for different number of URLs

| $n$ | Maximal value of $N_t$ |
|---|---|
| 10,000 | $0.067n$ |
| 100,000 | $0.061n$ |
| 1,000,000 | $0.059n$ |
| 2,000,000 | $0.059n$ |

**Time Complexity.** We now show that the time complexity of determining $g(v)$ for all critical vertices $x \in V(G_{\text{crit}})$ is $O(|V(G_{\text{crit}})|) = O(n)$. For each unassigned vertex $v$, the adjacency list of $v$, which we call $\text{Adj}(v)$, must be traversed to collect the set $Y$ of adjacent vertices that have already been assigned a value. Then, for each vertex in $Y$, we check if the current candidate value $x$ is forbidden because setting $g(v) = x$ would create two edges with the same endpoint sum. Finally, the edge linking $v$ and $u$, for all $u \in Y$, is associated with the address that corresponds to the sum of its endpoints. Let $d_{\text{crit}} = 2|E(G_{\text{crit}})|/|V(G_{\text{crit}})|$ be the average degree of $G_{\text{crit}}$, note that $|Y| \leq |\text{A}dj(v)|$, and suppose for simplicity that $|\text{A}dj(v)| = O(d_{\text{crit}})$. Then, putting all these together, we see that the time complexity of this procedure is

$$\begin{aligned} C(|V(G_{\text{crit}})|) &= \textstyle\sum_{v \in V(G_{\text{crit}})} \left[\, |\text{A}dj(v)| + (I(v) \times |Y|) + |Y| \right] \\ &\leq \textstyle\sum_{v \in V(G_{\text{crit}})} (2 + I(v))|\text{A}dj(v)| = 4|E(G_{\text{crit}})| + O(N_t d_{\text{crit}}). \end{aligned}$$

As $d_{\text{crit}} = 2 \times 0.501n/0.401n \simeq 2.499$ (a constant) we have $O(|E(G_{\text{crit}})|) = O(|V(G_{\text{crit}})|)$. Supposing that $N_t \leq N_{\text{bedges}}$, we have, from Theorem 1, that $N_t \leq |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1 = O(|E(G_{\text{crit}})|)$. We conclude that $C(|V(G_{\text{crit}})|) = O(|E(G_{\text{crit}})|) = O(|V(G_{\text{crit}})|)$. As $|V(G_{\text{crit}})| \leq |V(G)|$ and $|V(G)| = cn$, the time required to determine $g$ on the critical vertices is $O(n)$.

## 4 Experimental Results

We now present some experimental results. The same experiments were run with our algorithm and the algorithm due to Czech, Havas and Majewski [4], referred to as the CHM algorithm. The two algorithms were implemented in the C language and are available at `http://cmph.sf.net`. Our data consists of a collection of 100 million universe resource locations (URLs) collected from the Web. The average length of a URL in the collection is 63 bytes. All experiments were carried out on a computer running the Linux operating system, version 2.6.7, with a 2.4 gigahertz processor and 4 gigabytes of main memory.

Table 4 presents the main characteristics of the two algorithms. The number of edges in the graph $G = (V, E)$ is $|S| = n$, the number of keys in the input set $S$. The number of vertices of $G$ is equal to $1.15n$ and $2.09n$ for our algorithm and the CHM algorithm, respectively. This measure is related to the amount of space to store the array $g$. This improves the space required to store a function in our algorithm to 55% of the space required by the CHM algorithm. The number of critical edges is $\frac{1}{2}|E(G)|$ and 0 for our algorithm and the CHM

**Table 4.** Main characteristics of the algorithms

| | $c$ | $\|E(G)\|$ | $\|V(G)\| = \|g\|$ | $\|E(G_{\text{crit}})\|$ | $G$ | Order preserving |
|---|---|---|---|---|---|---|
| Our algorithm | 1.15 | $n$ | $cn$ | $0.5\|E(G)\|$ | cyclic | no |
| CHM algorithm | 2.09 | $n$ | $cn$ | 0 | acyclic | yes |

algorithm, respectively. Our algorithm generates random graphs that contain cycles with high probability and the CHM algorithm generates acyclic random graphs. Finally, the CHM algorithm generates order preserving functions while our algorithm does not preserve order.

Table 5 presents time measurements. All times are in seconds. The table entries are averages over 50 trials. The column labelled $N_i$ gives the number of iterations to generate the random graph $G$ in the mapping step of the algorithms. The next columns give the running times for the mapping plus ordering steps together and the searching step for each algorithm. The last column gives the percentage gain of our algorithm over the CHM algorithm.

**Table 5.** Time measurements for our algorithm and the CHM algorithm

| $n$ | Our algorithm | | | | CHM algorithm | | | | Gain (%) |
|---|---|---|---|---|---|---|---|---|---|
| | $N_i$ | Map+Ord | Search | Total | $N_i$ | Map+Ord | Search | Total | |
| 6,250,000 | 2.20 | 33.09 | 10.48 | 43.57 | 2.90 | 62.26 | 6.76 | 69.02 | 58 |
| 12,500,000 | 2.00 | 63.26 | 23.04 | 86.30 | 2.60 | 117.99 | 14.94 | 132.92 | 54 |
| 25,000,000 | 2.00 | 130.79 | 51.55 | 182.34 | 2.80 | 262.05 | 33.68 | 295.73 | 62 |
| 100,000,000 | 2.07 | 567.47 | 243.13 | 810.60 | 2.80 | 1,131.06 | 157.23 | 1,288.29 | 59 |

The mapping step of the new algorithm is faster because the expected number of iterations in the mapping step to generate $G$ are 2.13 and 2.92 for our algorithm and the CHM algorithm, respectively. The graph $G$ generated by our algorithm has $1.15n$ vertices, against $2.09n$ for the CHM algorithm. These two facts make our algorithm faster in the mapping step. The ordering step of our algorithm is approximately equal to the time to check if $G$ is acyclic for the CHM algorithm. The searching step of the CHM algorithm is faster, but the total time of our algorithm is, on average, approximately 58% faster than the CHM algorithm.

The experimental results fully backs the theoretical results. It is important to notice the times for the searching step: for both algorithms they are not the dominant times, and the experimental results clearly show a linear behavior for the searching step.

We now present a heuristic that reduces the space requirement to any given value between $1.15n$ words and $0.93n$ words. The heuristic reuses, when possible, the set of $x$ values that caused reassignments, just before trying $x + 1$ (see Section 3.3). The lower limit $c = 0.93$ was obtained experimentally. We generate 10,000 random graphs for each size $n$ ($n = 10^5$, $5 \times 10^5$, $10^6$, $2 \times 10^6$). With $c = 0.93$ we were always able to generate $h$, but with $c = 0.92$ we never succeeded.

Decreasing the value of $c$ leads to an increase in the number of iterations to generate $G$. For example, for $c = 1$ and $c = 0.93$, the analytical expected number of iterations are 2.72 and 3.17, respectively (for $n$ = 12,500,000, the number of iterations are 2.78 for $c = 1$ and 3.04 for $c = 0.93$). Table 6 presents the total times to construct a function for $n$ = 12,500,000, with an increase from 86.31 seconds for $c = 1.15$ (see Table 5) to 101.74 seconds for $c = 1$ and to 102.19 seconds for $c = 0.93$.

**Table 6.** Time measurements for our tuned algorithm with $c = 1.00$ and $c = 0.93$

| $n$ | Our algorithm $c = 1.00$ | | | | Our algorithm $c = 0.93$ | | | |
|---|---|---|---|---|---|---|---|---|
| | $N_i$ | Map+Ord | Search | Total | $N_i$ | Map+Ord | Search | Total |
| 12,500,000 | 2.78 | 76.68 | 25.06 | 101.74 | 3.04 | 76.39 | 25.80 | 102.19 |

We compared our algorithm with the ones proposed by Pagh [14] and Dietzfelbinger and Hagerup [6], respectively. The authors sent to us their source code. In their implementation the set of keys is a set of random integers. We modified our implementation to generate our $h$ from a set of random integers in order to make a fair comparison. For a set of $10^6$ random integers, the times to generate a minimal perfect hash function were $2.7s$, $4s$ and $4.5s$ for our algorithm, Pagh's algorithm and Dietzfelbinger and Hagerup's algorithm, respectively. Thus, our algorithm was 48% faster than Pagh's algorithm and 67% faster than Dietzfelbinger and Hagerup's algorithm, on average. This gain was maintained for sets with different sizes. Our algorithm needs $kn$ ($k \in [0.93, 1.15]$) words to store the resulting function, while Pagh's algorithm needs $kn$ ($k > 2$) words and Dietzfelbinger and Hagerup's algorithm needs $kn$ ($k \in [1.13, 1.15]$) words. The time to generate the functions is inversely proportional to the value of $k$.

## 5 Conclusion

We have presented a practical method for constructing minimal perfect hash functions for static sets that is efficient and may be tuned to yield a function with a very economical description.

## References

1. B. Bollobás. *Random graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001.
2. B. Bollobás and O. Pikhurko. Integer sets with prescribed pairwise differences being distinct. *European Journal of Combinatorics*. To Appear.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
4. Z.J. Czech, G. Havas, and B.S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

5. Z.J. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.
6. M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *The 9th European Symposium on Algorithms (ESA), volume 2161 of Lecture Notes in Computer Science*, pages 109–120, 2001.
7. P. Erdős and A. Rényi. On random graphs I. *Pub. Math. Debrecen*, 6:290–297, 1959.
8. P. Erdős and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 5:17–61, 1960.
9. E.A. Fox, Q.F. Chen, and L.S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 266–273, 1992.
10. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, July 1984.
11. G. Havas, B.S. Majewski, N.C. Wormald, and Z.J. Czech. Graphs, hypergraphs and hashing. In *19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 153–165. Springer Lecture Notes in Computer Science vol. 790, 1993.
12. S. Janson, T. Łuczak, and A. Ruciński. *Random graphs*. Wiley-Inter., 2000.
13. P. Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Scientia Hungary*, 12:261–267, 1961.
14. R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures*, pages 49–54, 1999.
15. B. Pittel and N. C. Wormald. Counting connected graphs inside-out. *Journal of Combinatorial Theory*. To Appear.

# Efficient and Experimental Meta-heuristics for MAX-SAT Problems

Dalila Boughaci[1] and Habiba Drias[2]

[1] ITS, University of sciences and technology
BP.32- EL-ALIA Bab-Ezzouar, 16111, Algiers, Algeria
Fax/Phone (+213) 21 24 78 92,
dalila_info@yahoo.fr
[2] National Institute of Computer Science
INI- BP 68M OUED SMAR EL HARRACH, Alger
Tel 213 21 51 60 77 - Fax 213 021 51 61 56,
drias@wissal.dz

**Abstract.** Many problems in combinatorial optimization are NP-Hard. This has forced researchers to explore meta-heuristic techniques for dealing with this class of complex problems and finding an acceptable solution in reasonable time. The satisfiability problem, SAT, is studied by a great number of researchers the three last decades. Its wide application to the domain of AI in automatic reasoning and problem solving for instance and other domains like VLSI and graph theory motivates the huge interest shown for this problem. In this paper, tabu search, scatter search, genetic algorithms and memetic evolutionary meta-heuristics are studied for the NP-Complete satisfiability problems, in particular for its optimization version namely MAX-SAT. Experiments comparing the proposed approaches for solving MAX-SAT problems are represented. The empirical tests are performed on DIMACS benchmark instances.

## 1 Introduction

The satisfiabilty problem, SAT, is one of the most known NP-complete problems: given a collection of m clauses C= { C1, C2, ….. Cm} involving n Boolean variables x1, x2 ,…, xn. The SAT problem is to determine whether or not there exists a truth assignment for C that satisfies the m clauses. A clause is a disjunction of literals. A literal is a variable or its negation. A formula in conjunctive normal form (CNF) is a conjunction of clauses. The formula is said to be satisfiable if there exists an assignment that satisfies all the clauses and unsatisfiable otherwise. In the latter situation, we are interested to other variants of SAT. We mention among them the maximum satisfiability problem (MAX-SAT) which consists in finding an assignment that satisfies the maximum number of clauses. MAX-SAT is the optimization variant of SAT [14]. It is an important and widely studied combinatorial optimization problem with applications in artificial intelligence and other areas of computing science. The decision variants of both SAT and MAX-SAT are NP-Complete [5, 9, 16]. The interest focuses on the development and implementations of heuristics.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 501 – 512, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Many algorithms have been proposed and important progress has been achieved. These algorithms can be divided into two main classes:

- *Exact or Complete algorithms*: dedicated to solve the decision version of SAT problem. The well- known algorithms are based on the Davis-Putnam-Loveland procedure [6]. Satz [20, 21] is a famous example of a complete algorithm.
- *Incomplete algorithms*: they are mainly based on local search and evolutionary algorithms. Gsat [25], Tabu search [22, 1, 2], simulated annealing [14], genetic algorithms [8], Grasp [24], scatter search [7] and recently memetic algorithm [3] are examples of incomplete algorithms for SAT. These meta-heuristics are a good approach for finding a near solution of very large instances, in particular for unsatisfiable or unknown instances.

In this paper, tabu search, scatter search, genetic and memetic evolutionary meta-heuristics are studied and compared for solving MAX-SAT problems. Empirical tests are performed on DIMACS benchmark instances. The paper starts with a brief review of the Tabu Search (TS). Section 3 introduces the Population-based Tabu Search (PTS). In section 4, we explain the most key components of the classical Genetic algorithms. In section 5, we propose our memetic approach. Section 6, presents the scatter search- based population meta-heuristic (SS). Our comparative study and experiments results are summarized in section 7. Finally, conclusion and future work are explained in section 8.

## 2 A Tabu Search Meta-heuristic

Tabu search is one of the efficient methods for large combinatorial optimization problems. Given the search space, the method attempts to find a global minimum state. It is a general meta-heuristic that has been proposed by Fred Glover [10, 11]. It has been applied to various optimization problems
including the arc routing problem [13], Satisfiability problem [22, 1, 2], bid evaluation [4], job shop scheduling [26] and the mix fleet vehicle routing problem [27]. Like local search, Tabu search starts with an initial configuration generated randomly, then, the best neighbor solutions are selected. Tabu search uses also a list called "Tabu list" to keep information about solutions recently selected, that, in order to escape the solutions already visited. In the case, in which, a Tabu move applied to a current solution gives a better solution; we accept this move in spite of its Tabu status by aspiration criterion. The search stops when the quality of the solution is not improved during a maximum number of iterations or when we reach the optimum global.

### 2.1 Tabu Search Items

To use Tabu search for solving MAX-SAT problem, we define the following items:

**A Solution** is represented by a binary chain X (n Vector). Each of whose components xi receives the value 0 or 1. It is defined as a possible configuration verifying the

problem constraints and satisfying the goal that consists in finding an assignment of truth values to the n variables that maximizes the number of satisfied clauses.

**A move** is an operator used to generate neighbor solutions. An elementary move consists in flipping one of the variables of the solution. The neighborhood of a solution is constituted by all the solutions obtained by applying an elementary move on this solution. A variable has the tabu state if it has been modified during the current move and it keeps it during a certain number of iterations called **tabu tenure**.

**A Tabu List** is used to keep information about the solutions already visited in order to escape the local optimum by searching in new regions not already explored.

### 2.2 A Tabu Search Outline

**Step1. Initialization,**

Set Tabu search parameters
// maxiter is the maximum number of iterations
// iter is the Number of process iterations,
//bestiter is the iteration where the best solution has been found
//S*is the best solution with the minimum F* corresponds to S*,
// F* objective function value that is F*=F(S*)

iter:=0; bestiter:=0; TL: = Φ; //TL is the Tabu list ,
Generate an arbitrary solution S; Evaluate F (S); S*:= S, F*: = F;

**Step2. Iteration,**

While (iter- bestiter < maxiter) **do**
**begin**
iter:= ier+1;
Generate neighbor of solutions using the move;
Select the best move**;**
Ignore the tabu status by aspiration Criterion if such move lead to a best solution**;**
Apply the selected move to the current configuration S; Save the move in TL;
Evaluate F(S); If (F >F*) **then begin** S* := S; F*:= F; bestiter: = iter; **end;**
**End;**

**Step3. Termination,** Print the best solution with the best cost.

## 3 A Population–Based Tabu Search

In order to scan the most important part of the whole search space for attaining good solutions, we add a population strategy on the tabu search single-oriented solutions.
The tabu search with the added constraints of the population strategy, PTS, maintains a pool of solutions rather than a single solution. Initially, the new approach starts from an arbitrary set of individuals, then a neighbor of each individual in the current population is generated using a variable flipping move operator. The best neighbor solution is selected and the moves that return a previously generated solution are considered tabu and not accepted. In addition, the back-tracking to previous solutions is permitted in the case, in which a tabu moves lead to a high quality solution. The basic components of the population-based tabu search are depicted on figure1.

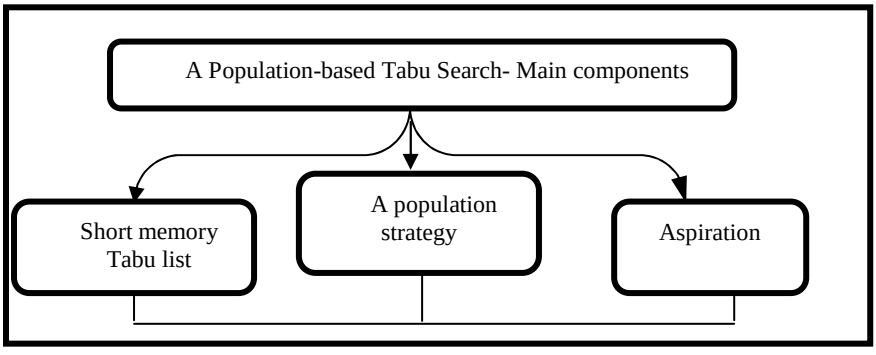



**Fig. 1.** Population- based Tabu Search Main components

### 3.1 A Population-Based Tabu Search Outline

The overall of the population–based Tabu Search algorithm is described as:

**Step1. Initialization:**
Set PTS parameters.
// Maxiter: is the maximum number of iterations,
// S* is the best solution with the minimum F* corresponds to S*,
F* objective function value that is F*=F(S*),
// Psize is the population size,
iter=0 ; // iter is the number of PTS process iterations
TL = Φ; //TL is the Tabu list ,
- Generate an arbitrary Population P; - Evaluate the individuals in P;
- S* is the best individual in P, F* = F(S*)

**Step 2 Iteration**
While (iter < maxiter) do
Begin iter++;
For i= 1 to PSize do
begin
- Generate neighbor of the individual "Si" using the move,
- Select the best move,
- Ignore the tabu status by aspiration, criterion if such move generates a best solution,
- Apply the selected move to the current configuration "Si",
- Save the move in TL,
- Evaluate F(Si), if F >F* S* = Si; F*= F;
end;
end;

**Step3 termination**. Print the best solution with the best cost.

## 4 Genetic Algorithms

Genetic algorithms [12, 8] are an evolutionary meta-heuristic that have been used for solving difficult problems. They have been applied to complex optimization problems

with remarkable success in some cases. Their behavior mimics the process of natural evolution. A population initially made of candidate solutions representing individuals improves towards another population of individuals with higher quality along a process repeating a finite number of times, sequentially reproduction between individuals, and mutation of chromosomes and selection of better individuals. The goal is to create a very fit individual.

**Input:** an instance of satisfiability ;
**Output**: an assignment of variables that maximizes the number of satisfied clauses**;**
**Begin**
Generate at random the initial population;
**While** (the maximum number of generations is not reached and the optimal solution is not found) **do Begin**
**Repeat** Select two individuals;
Generate at random a number Rc from [0, 100];
If (Rc > crossover rate) then apply the crossover;
Generate at random Rm from [0, 100];
*While* (Rm < mutation rate *do Begin*
Choose at random a chromosome from the individual obtained by the cross over and flip it; Generate at random Rm from [0, 100]; *End* ;
Evaluate the new individual;
**End repeat;**
Replace the bad individuals of the population by the fittest new ones.
**End;**
Return the best chromosome
**End;**

**Fig. 2.** Genetic algorithm outline for MAX-SAT

The algorithm, depicted on figure 2 above, operates as follows. From a population of points (parents), the algorithm constructs a new population (children) in combining several parents and applying some random modifications (mutation). The selection phase chooses the best points among parents and children to produce the next population for the next iteration. Usually, the genetic algorithms converges, ie, the population has the tendency to lose its diversity, so it loses its efficacy; it is why the convergence is often used like stop criteria. However, the premature convergence of genetic algorithms is an inherent characteristic that makes them incapable of searching numerous solutions of the problem domain why it is frequent to stop searching after a certain number of generations.

## 5 Memetic Algorithms

The Memetic algorithms [23] can be viewed as a marriage between a population-based global technique and a local search made by each of the individuals. They are a special kind of genetic algorithms with a local hill climbing. Like genetic algorithms,

memetic algorithms are a population-based approach. They have shown that they are orders of magnitude faster than traditional *Genetic Algorithms* for some problem domains. Basically, they combine local search heuristics with crossover operators. Therefore, some researchers have viewed them as *Hybrid Genetic Algorithms* [12], others known it as *Genetic Local Search*. They have also received the denomination of *Parallel Genetic Algorithms* [17, 12]. In a memetic algorithm the population is initialized at random or using a heuristic. Then, each individual makes local search to improve its fitness. To form a new population for the next generation, higher quality individuals are selected. The selection phase is identical inform to that used in the classical genetic algorithm selection phase. Once two parents have been selected, their chromosomes are combined and the classical operators of crossover are applied to generate new individuals. The latter are enhanced using a local search technique. The memetic meta-heuristic is a competitive- cooperative approach. The cooperative aspect is supplied by a crossover operators and the local search is supplied by a search technique ST. The competitive aspect is supplied by a selection phase. As you can see on the figure 3, the memetic approach can not include the mutation phase but it may include two necessary phases:

- A local search and cooperation (muting of individuals)
- A competition (selection of better individuals)

The two phases above are repeated until a stopping criterion is satisfied.

**Step1**. Initialize the parameters of the Memetic algorithm (***MA***)
**Step 2.** Generate at random an initial population of individuals for the MA
**Step 3.** For each of the individuals apply a ***ST*** to improve its fitness
**Step 4.** Selection: each individual may choose a partner for muting
**Step 5**. Reproduction using crossover and ***ST***
**Step 6.** Replacement of individuals
**Step 7 .** If not finished, go to Step3.

**Fig. 3.** A Memetic algorithm in pseudo-code

## 6 Scatter Search Meta-heuristic

Scatter search [18, 19] is a population-based meta-heuristic like genetic and memetic algorithms. It is an evolutionary method that constructs solutions by combining others**.** The approach starts with an initial population (collection of solutions) generated using both diversification and improvement strategies, then, a set of best solutions (reference set that incorporates both diverse and high quality solutions) are selected from the population. These collections of solutions are a basis for creating new solutions consisting of structured combinations of subsets of the current reference solutions.

### 6.1 A Scatter Search Template

Four methods are used to achieve the scatter search template:

1. **Diversification generation and reference set method**. This step generates, at first, diverse solutions, then, improves them and selects the most elite and diverse to create the reference set solutions.
2. **Improvement method**. The improvement heuristic is used in step 1 and step 4 to enhance the quality of solutions.
3. **Subset generation method**. This method creates new solutions based on deterministic of subsets of the reference set solutions.
4. **Solution combination method**. The structured combination produces solution inside and outside of the regions spanned by the reference set.

**Fig. 4.** Scatter Search template

## 7 A Comparative Study Between TS, PTS, SS, GA, and MA

In this work, computational experience regarding five well-known meta-heuristics (Tabu search, Population- based Tabu Search, genetic algorithms, memetic algorithms and scatter search) for solving MAX-SAT instances are reported. The purpose of this comparative experiment is to evaluate the performance of each one of the different techniques to solve MAX-SAT instances. First of all, we compare on the table below the different approaches regarding their Principles and the operators used by each approach. Further, we give some numerical results obtained by applying each algorithm on MAX-SAT instances.

**Computational Results**

The purpose of this experiment is to evaluate the performance of the evolutionary approaches (PTS, SS, GA, and MA) for solving MAX-SAT instances. All experiments were run on a 350 Mhz Pentium II with 128 MO RAM. All instances have been taken from the SATLIB [15]. They are hard Benchmark Problems. On each instance the four algorithms have been executed in order to compute the average of the maximum number of satisfied clauses. The tables below show the results obtained by our algorithms. These columns contain the number of variables, the number of clauses, the number of satisfied clauses, the rate of satisfied clauses and the running time in second.

**The DIMACS Benchmarks**

Two kinds of experimental tests have been undertaken. The goal of the first ones is the setting of the different parameters of the TS, PTS, SS, GA and MA algorithms like the Tabu tenure, population size, crossover rate, the mutation rate, and the type of crossover, the number of iterations, the population size and the interaction between the algorithms parameters. These parameters are fixed as:

**Table 1.** Comparison of TS, PTS, SS, GA, and MA approaches

| | **Tabu Search** | **Population-based Tabu Search** | **Genetic Algorithms** | **Memetic Algorithms** | **Scatter Search** |
|---|---|---|---|---|---|
| **Principles** | - neighbor search meta-heuristic<br>- Single oriented approach<br>-Interdiction | - Evolutionary meta-heuristic<br>-Based-population<br>-Evolution with interdiction | Evolutionary meta-heuristic<br>-Based population<br>-Biological evolution | - Evolutionary meta-heuristic<br>- Based population<br>- Cultural evolution | Evolutionary meta-heuristic<br>- Based-population<br>-Biological evolution |
| **Operators** | Move<br>Tabu list<br>Aspiration-criterion | Selection<br>Move<br>Tabu list<br>Aspiration-criterion | Selection<br>Crossover<br>Mutation | Selection<br>Crossover + local search<br>Mutation | Reference set selection<br>Structured combination<br>Improvement technique |
| **Configuration or Population generation** | At Random | At Random | At Random | At Random or using a heuristic | Using diversification generator |

- PTS algorithm parameters are fixed as: the maximum number of iterations was set to Maxiter=200, the population size was set to Psize = 100 and the tabu list size was set to |TL|=7.
- Scatter search variant algorithm parameters: the maximum number of generations was set to maxiter=5, the population size was set to Psize = 50, the reference set was set to 10 (B1= 5, B2= 5) and the Improvement tabu search parameters was set as: maximum number of iteration = 30 and tabu list size was set to |TL|=7.
- Memetic algorithm parameters: the maximum number of generations was set to maxiter=10, the population size was set to Psize = 50, Rc=0.6 and the ST parameters was set as. Number of iterations, T=20.
- Genetic algorithm parameters: the maximum number of generations was set to maxiter=10000, the population size was set to Psize = 50, Rc=0.6 and Rm=0.5.

The second kind of experiments concerns MAX-SAT instances. All these instances are encoded in DIMACS CNF format. For more information about these instances, see the SATLIB Web site at [www.satlib.org].
The results found are classed by class:

- **AIM class:** Artificially generated random 3-SAT, defined by Kazuo Iwama, Eiji Miyano and Yuichi Asahiro. We have chosen four Satisfiable instances.

**JNH class:** Randomly generated instances- constant density model. The instances have originally been contributed by John Hooker (jh38+@andrew.cmu.edu). "Yes", means that the instance is satisfiable, "no" means that the instance is not satisfiable.

**-Dubois class:** Instance from generator gensathard.c defined by Olivier Dubois (dubois@laforia.ibp.fr). All the instances are not satisfiable.

**Table 2.** Solutions quality and running time results obtained by PTS, MA, GA and SS on AIM instances

| Instances | # var | # clauses | **PTS-Sol** | Rate % | time | **MA-Sol** | MA Rate | MA Time |
|---|---|---|---|---|---|---|---|---|
| Aim-50-1 | 50 | 80 | 79 | 99 | 67,3 | 79 | 99% | 43,32 |
| Aim-50-2 | 50 | 100 | 100 | 100 | 16,3 | 99 | 99% | 44,84 |
| Aim-50-3 | 50 | 170 | 170 | 100 | 22,0 | 168 | 99% | 71,37 |
| Aim-50-6 | 50 | 300 | 300 | 100 | 51,4 | 295 | 99% | 121,8 |

| **GA-** sol | GA-Rate% | GATime | **SS-** Sol | SS - Rate % | SS-Time |
|---|---|---|---|---|---|
| 78 | 97,5 | 57,88 | 80 | 100 | 20,0 |
| 97 | 97 | 65,21 | 99 | 99 | 34,1 |
| 162 | 95 | 93,45 | 170 | 100 | 0,7 |
| 280 | 93 | 124,0 | 300 | 100 | 1,1 |

**Table 3.** Solutions quality and running time results obtained by PTS, MA, GA and SS on JNH instances

| Instances | # var | #clauses | **PTS-Sol** | Rate % | Time | **MA -Sol** | MA Rate% | Time |
|---|---|---|---|---|---|---|---|---|
| Jnh201-yes | 100 | 800 | 800 | 100 | 406,3 | 800 | 100 | 290,99 |
| Jnh202- no | 100 | 800 | 798 | 99 | 1516,7 | 794 | 99 | 621,41 |
| Jnh203-no | 100 | 800 | 798 | 99 | 1737,8 | 792 | 99 | 799,12 |
| Jnh204-yes | 100 | 800 | 799 | 99 | 1452,0 | 793 | 99 | 779,88 |
| Jnh205-yes | 100 | 800 | 800 | 100 | 1283,3 | 796 | 99 | 520,3 |
| Jnh206-no | 100 | 800 | 799 | 99 | 1430,9 | 793 | 99 | 824,5 |
| Jnh207-yes | 100 | 800 | 800 | 100 | 1064,5 | 795 | 99 | 783,34 |
| Jnh208-no | 100 | 800 | 799 | 99 | 1430,1 | 791 | 99 | 526,11 |
| Jnh209-yes | 100 | 800 | 800 | 100 | 614,1 | 794 | 99 | 782,6 |
| Jnh210-yes | 100 | 800 | 800 | 100 | 313,0 | 797 | 99 | 520,01 |

| **GA sol** | GA Rate% | GA Time | **SS sol** | SS Rate% | Time |
|---|---|---|---|---|---|
| 780 | 97,5 | 444,70 | 800 | 100 | 831,7 |
| 776 | 97 | 430,98 | 797 | 99 | 800,0 |
| 774 | 97 | 432,60 | 798 | 99 | 700,0 |
| 777 | 97 | 430,66 | 798 | 99 | 799,3 |
| 775 | 97 | 433,66 | 800 | 100 | 0,945 |
| 775 | 97 | 429,74 | 799 | 100 | 848,7 |
| 777 | 97 | 431.78 | 799 | 99 | 790,4 |
| 772 | 96,5 | 439,25 | 798 | 99 | 225,7 |
| 775 | 97 | 429,72 | 800 | 100 | 635,7 |
| 776 | 97 | 432,31 | 800 | 100 | 218,3 |

Above, some results found by the population-based tabu search, scatter search, genetic and memetic Algorithms. The results obtained by the evolutionary approaches are acceptable. In general, we can observe the superiority of the population-based tabu search and scatter search algorithms in solving the MAX-SAT problems from the quality of solution point of view (we have reached the optimum for lot of instances). **PTS**: When a population strategy is incorporated in TS the solutions space is better searched. **SS**: When intensified improvement Tabu search and diversified components are incorporated in SS, the solutions space is better searched. But the process search takes more time to find the solution. **MA**: When the local search ST is incorpo-

rated in GA the solutions space is better searched. We precise, that in general, the role of a local search technique in Scatter, PTS, and memetic search algorithms is to locate the solution more efficiently. According to the experimental results, we can conclude that the single-oriented approaches play an important role in population-based processes.

**Table 4.** Solutions quality and running time results obtained by PTS, MA, GA and SS on Dubois instances

| Instances | # Var | #clauses | **PTS-Sol** | Rate% | Time | **MA sol** | MA Rate% | MA Time | **GA-sol** | GAate% | GA time | **SS sol** | SS Rate% | SS Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Duboi20 | 60 | 160 | 159 | 100 | 84.75 | 159 | 100 | 1,63 | 155 | 97 | 430,01 | 159 | 100 | 53,9 |
| Duboi21 | 63 | 168 | 167 | 100 | 95.77 | 167 | 100 | 1,61 | 161 | 96 | 144,07 | 167 | 100 | 66,8 |
| Duboi22 | 66 | 176 | 175 | 100 | 106.00 | 175 | 100 | 1,68 | 171 | 98 | 90,02 | 175 | 100 | 61,7 |
| Duboi23 | 69 | 184 | 183 | 100 | 116,61 | 183 | 100 | 1,63 | 175 | 96 | 92,62 | 183 | 100 | 77,6 |
| Duboi24 | 72 | 192 | 191 | 100 | 120.89 | 191 | 100 | 1,69 | 185 | 97 | 88,84 | 191 | 100 | 49,9 |
| Duboi25 | 75 | 200 | 199 | 100 | 129,89 | 199 | 100 | 1,74 | 189 | 95 | 96,03 | 199 | 100 | 44,0 |
| Duboi26 | 78 | 208 | 207 | 100 | 165,03 | 207 | 100 | 1,80 | 199 | 96 | 101,66 | 207 | 100 | 86,4 |
| Duboi27 | 81 | 216 | 215 | 100 | 149,60 | 215 | 100 | 1,78 | 205 | 95 | 110.24 | 215 | 100 | 101,8 |
| Duboi28 | 84 | 224 | 223 | 100 | 225,92 | 223 | 100 | 1,72 | 211 | 95 | 106,19 | 223 | 100 | 108,8 |
| Duboi29 | 87 | 232 | 231 | 100 | 287,96 | 231 | 100 | 1,62 | 219 | 95 | 111,62 | 231 | 100 | 122,2 |
| Duboi30 | 90 | 240 | 239 | 100 | 214,35 | 239 | 100 | 1,81 | 227 | 95 | 117.98 | 239 | 100 | 67,3 |
| Duboi50 | 150 | 400 | 399 | 100 | 548,47 | 399 | 100 | 1,51 | 373 | 94 | 282,8 | 399 | 100 | 335,8 |

## 8 Conclusion and Future Work

In this paper, we have presented, at first, the single–oriented meta-heuristic called Tabu search. We have proposed to hybridize it with a population strategy to solve the MAX-SAT optimization problems. The solutions found by the PTS method are very encouraging. When a population strategy is incorporated in TS the solutions space is better searched. Then, we have presented some well-known evolutionary meta-heuristics which are genetic algorithms, memetic algorithm and scatter search. The four proposed approaches have been implemented on machine for solving MAX-SAT hard instances. The solutions found by the methods are encouraging; the numerical

results show a considerable performance in favor of PTS and scatter search. The scatter search method takes its superiority from the use of several techniques as diversification used by scatter search components (generator, combination methods) and intensification strategy by the improvement procedure. But the search process takes more time for finding the solution in comparison with memetic and PTS approaches. The solutions found by the memetic method are encouraging; the result method takes its superiority from the use of several techniques as cooperation supplied by crossover operators and the local search heuristic and the competitive aspect supplied by a selection phase. When ST is incorporated in GA the solutions space is better searched. We plan to improve our framework to implement a parallel hybrid evolutionary approach.

## References


1. D.Boughaci H.Drias "Solving Weighted Max-Sat Optimization Problems Using a Taboo Scatter Search Meta-heuristic". *In proceedings of ACM* Sac 2004, pp35-36, 2004.
2. D.Boughaci , H.Drias "PTS: A Population-based Tabu Search for the Maximum Satisfiability Problems"**,** *in the Proceedings of the 2 IEEE GCC conferences*, pp 622-625, 2004.
3. D.Boughaci , Drias H and Benhamou B. 2004. "Solving Max-Sat Problems Using a memetic evolutionary Meta-heuristic", *in proceedings of 2004 IEEE CIS*, pp 480-484, 2004.
4. D. Boughaci and H. Drias, "Taboo Search as an Intelligent Agent for Bid Evaluation", *in proceedings of CE2003, book1*, pages 43-48, Portugal, 2003.
5. S.Cook . "The Complexity of Theorem Proving Procedures", *in the Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, 1971*.
6. M.Davis, G.Putnam , and D.Loveland. "A machine program for theorem proving", *communication of the ACM*, 394-397, Jul 1962.
7. H.Drias. "Scatter search with walk strategy for solving hard Max-W-Sat problems", *in proceedings of IEA- AIE2001, lectures note in computer science, LNAI-2070, Springer*, Budapest, 35-44, 2001.
8. J.Frank, "A study of genetic algorithms to find approximate solutions to hard 3CNF problems", *in proceedings of Golden West international conference on artificial intelligence,1994*.
9. MR.Garey, S.Johnson. "Computer and intractability, a guide to the theory of NP-Completeness, Freeman company Sanfranscsco, 1978.
10. F. Glover, "Future paths for integer programming and links to Artificial intelligence", *operational search*, vol 31, 1986.
11. F. Glover, "Tabu search": Part I, *ORSA, journal on computing*, 1989.
12. D. E Goldberg,."Genetic Algorithms in search, Optimization & Machine Learning". Wokingham, Addison-Wesley. 1989)
13. P. Greistorfer , "A Tabu Scatter search meta-heuristic for the arc routing problem", submitted to Elsevier Science February 2002.
14. P.Hansen P; B.Jaumard. "Algorithms for the Maximum Satisfiability", journal of computing 44-279-303, 1990.
15. H. H. Hoos and T. Stutzle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in theYear 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

16. S.Johnson. “Approximation algorithms for combinatorial problems”, Journal of Computer and System Sciences 9, 256-278, 1974.
17. SA. Kauffman, Levin .” Towards a general theory of adaptive walks on rugged landscapes”, J theory. Biol, 128,pp, 11,1987.
18. M.Laguna , F.Glover *“Scatter Search”*, Graduate school of business, University of Colorado, Boulder, 1999.
19. M.Laguna., R.Marti and V Campos, “Scatter Search for the linear ordering problem”, University of Colorado, Boulder, 1999.
20. CM.Li., “Exploiting yet more the power of unit clause propagation to solve 3-sat problem”, *in ECAI 96, workshop on advances in* propositional deduction, pqges 11-16, Boudapest, Hungray, 1996.
21. CMLi and. Anbulagan, “Heuristic based on unit propagation for satisfiability”, in Proceedings of CP 97, springer-Verlag, LNCS 1330, pp 342-356, Austria, 1997.
22. B.Mazure, L.Sais and E.Greroire,.” A Tabu search for Sat”, in proceedings of AAAI 1997.
23. P.Moscato. “On evolution, search, optimization, genetic algorithms and martial arts: toward memetic algorithms”, 1989.
24. PM.Pardalos, L.Pitsoulis and MGC. Resende. “A Parallel GRASP for MAX-SAT Problems. PARA96 Workshop on Applied Parallel Computing in Industrial Problems and Optimization”, Lynghy. Denmark August 18-21, 1996.
25. B.Selman , A.Henry , Z.Kautz and B.Cohen . “Local Search Strategies for Satisfiability Testing”, *presented at the second DIMACS Challenge on Cliques, Coloring, and Satisfiability, October 1993*
26. E. Taillard , "Parallel Taboo search techniques for Job Shop Scheduling problem", *ORSA Journal on Computing*, 6:108-117, 1994.
27. N.A. Wassan, and I.H. Osman, "Tabu search variants for the mix fleet vehicle routing problem", *Tech. rep.*, School of Business, American University of Beirut, forthcoming in J. Opl Res. Soc, 2002.

# Experimental Evaluation of the Greedy and Random Algorithms for Finding Independent Sets in Random Graphs*

M. Goldberg, D. Hollinger, and M. Magdon-Ismail

Computer Science Department
Rensselaer Polytechnic Institute

**Abstract.** This work is motivated by the long-standing open problem of designing a polynomial-time algorithm that with high probability constructs an asymptotically maximum independent set in a random graph. We present the results of an experimental investigation of the comparative performance of several efficient heuristics for constructing maximal independent sets. Among the algorithms that we evaluate are the well known randomized heuristic, the greedy heuristic, and a modification of the latter which breaks ties in a novel way. All algorithms deliver online upper bounds on the size of the maximum independent set for the specific input-graph. In our experiments, we consider random graphs parameterized by the number of vertices $n$ and the average vertex degree $d$. Our results provide strong experimental evidence in support of the following conjectures:

1. for $d = c \cdot n$ ($c$ is a constant), the greedy and random algorithms are asymptotically equivalent;
2. for fixed $d$, the greedy algorithms are asymptotically superior to the random algorithm;
3. for graphs with $d \leq 3$, the approximation ratio of the modified greedy algorithm is asymptotically $< 1.005$.

We also consider random 3-regular graphs, for which non-trivial lower and upper bounds on the size of a maximum independent set are known. Our experiments suggest that the lower bound is asymptotically tight.

## 1 Introduction

The problem of constructing a maximum independent set (*MIS*) in a graph is a classical NP-hard computational problem [9], which arises in many applications. It is also NP-hard to approximate the size of the maximum independent set [11]. For general graphs, the best approximation algorithm for the independence number has an approximation ratio of $O(n/(\log n)^2)$ [4]; for bounded-degree graphs, the best ratio known is $O(\Delta/\log\log\Delta)$ ([8]). The problem is 2-approximable on

* This research was partially supported by NSF grants 0324947 and 0346341.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 513–523, 2005.
© Springer-Verlag Berlin Heidelberg 2005

random graphs with fixed edge probability $p$ (the average degre $d$ is $(n-1)p$). Here we consider random graphs in the Erdös-Rényi, or $G(n,p)$-model [2, 5]: given $n$ vertices, each of the $\binom{n}{2}$ edges is generated independently with probability $p$. Let $\alpha(n,p)$ be a random variable denoting the size of a maximum independent set in a $G(n,p)$ graph. For fixed $p$, it is known that the standard randomized algorithm *Random* (described below) outputs an independent set of size [1] $\sim \log_{1/(1-p)} n$, with probability $\to 1$ (w.p.1). The 2-approximability of *MIS* on $G(n,p)$ with fixed $p$ follows from the results by Bollobás and Erdös ([3]) and Matula ([12]), where they show that for a $G(n,p)$ graph with fixed $p$, $\alpha(n,p) \sim 2\log_{1/(1-p)} n$, w.p.1. For $c > 0$, no polynomial algorithm for *MIS* on a $G(n,p)$ is known to find, w.p.1, an independent set of size $\sim (1+c)\log_{1/(1-p)} n$. Frieze and McDiamard [6–page 11] pose the following research problem:

> *Construct a polynomial algorithm that finds an independent set of size* $\sim (\frac{1}{2}+c)\alpha(n,p)$ *with high probability, or show that no such algorithm exists, modulo some reasonable conjecture on the computational complexity hierarchy (e.g.* $P \neq NP$*).*

Two well-known algorithms for *MIS* are *Random* and *Greedy*, which are instantiations of the following general *sequential* algorithm:

1: *Sequential algorithm for MIS*:
2: $I = \emptyset$ (the independent set);
3: **while** $G \neq \emptyset$ **do**
4: select a vertex $v \in G$;
5: $I \leftarrow I \cup \{v\}$;
6: $G \leftarrow G \setminus \{N(v) \cup v\}$

In *Random*, the vertex selected in step 4 is random, whereas in *Greedy* it is a minimum degree vertex (ties are broken randomly). *Random* is generally easier to analyze mathematically ([2]), but it under-performs *Greedy* on most graph domains. It is not known whether *Greedy* is asymptotically better than, worse than, or equivalent to *Random* on $G(n,p)$ graphs. It is not known whether there are any polynomial-time algorithms that perform asymptotically better than *Random*. The difficulty encountered by existing analysis techniques in attempting to analyze deterministic algorithms, *e.g. Greedy*, on random objects is that during execution, the algorithm destroys the randomness in an analytically unpredictable way. This difficulty already surfaces when one analyzes *Greedy* on random graphs with average degree 3. The only results that carry through the analysis of *Greedy* are those for finding matchings in random graphs ([10], [1]) and for 3-regular random graphs ([7]). However, it is not known how *Greedy* performs on random $d$-regular graphs with $d \geq 4$; on random graphs with the average vertex degree $d$, for any $d > 0$; or on random graphs with a fixed edge-probability $p > 0$ $(d = p(n-1))$.

[1] We use the notation $f(n) \sim g(n)$ if $\lim_{n\to\infty} f(n)/g(n) = 1$.

In this paper, we present the results of a comparative experimental investigation of the performance of several fast polynomial heuristics for constructing maximum independent sets in random graphs. The procedures that we tested include *Random*, *Greedy*, and a modification of the latter, *Greedy*$^m$, which breaks the ties in a novel way. All three heuristics deliver, *on-line*, upper bounds on the size of the maximum independent set in the input-graph. The input domain of all experiments is the set of random graphs with a given number $n$ of vertices and a given average vertex degree $d$. Based on the experiments, we formulate the following conjectures:

- for $d = c \cdot n$ ($c$ is a constant), *Random* and the greedy algorithms are asymptotically equivalent, i.e., the independent sets they construct are asymptotically of the same size;
- for fixed $d$, *Greedy* and *Greedy*$^m$ are asymptotically superior to *Random*;
- for $d \leq 3$, the approximation ratio of *Greedy*$^m$ is asymptotically $< 1.005$ w.p.1.

We also tested our algorithms on random 3-regular graphs for whichnon-trivial lower and upper bounds on the *MIS* exist ([7]). The results suggest that the lower bound is asymptotically tight.

We will use standard graph theory notation as described in [13]. For a graph $G$, the independence number $\alpha(G)$ is the maximal size of an independent set in $G$. For $v \in V(G)$, $\alpha_v(G)$ denotes the maximal size of an independent set containing $v$. Vertex $v$ is **correct** if $\alpha_v(G) = \alpha(G)$. $N(v)$ is the neighborhood of $v$, the set of vertices adjacent to $v$; $deg(v)$, the degree of $v$, is the size of the neighborhood, $|N(v)|$; $deg(G) = \max_v deg(v)$. For a subset of the vertices, $T \subseteq V(G)$, $N(T)$, the neighborhood of $T$, is the union of the neighborhoods of the vertices in $T$.

Let $\mathcal{A}$ be a sequential algorithm for *MIS*, which constructs an independent set by repeatedly selecting vertices, *e.g. Random*, or *Greedy*. When $\mathcal{A}$ is applied to $G$, suppose that the sequence of vertices selected is $\{v_1, v_2, \ldots, v_k\}$ ($k$ is the size of the independent set). Let $G_i$ be the subgraph obtained from $G$ by removing $N(\{v_1, \ldots, v_{i-1}\})$. The **dynamic degree** $\overline{deg}(v_i)$ is the degree of $v_i$ in $G_i$ (the dynamic degree depends on the algorithm $\mathcal{A}$); $\overline{deg}(G)$, the dynamic degree of $G$, is $\max_i\{\overline{deg}(v_i)\}$. Let $I_\ell$ denote the vertices in the output independent set whose dynamic degrees are $\ell$ ($\ell \geq 0$); thus, $I = \cup_{\ell \geq 0} I_\ell$.

## 2 On-line Upper Bounds

A key feature of our algorithms is that we bound the approximation ratio of the output independent set, for the specific input graph. Our bounds are based on the following observations.

**Lemma 1.** *Let $v \in V(G)$. If $deg(v) = 0$, then $\alpha_v(G) = \alpha(G)$, otherwise $\alpha_v(G) \geq \alpha(G) - deg(v) + 1$. If $\alpha_v(G) < \alpha(G)$, then every MIS of $G$ contains at least $\alpha(G) - \alpha_v(G) + 1$ vertices from $N(v)$.* ■

**Lemma 2.** *For a sequential algorithm $\mathcal{A}$ which selects vertices $\{v_1, \ldots, v_k\}$, $\alpha(G) \leq |I_0| + \sum_{i=1}^{k} \overline{deg}(v_i)$.* ■

The lemmas imply that selecting any vertex of dynamic degree 0 or 1 is correct, and each vertex of dynamic degree 2 gives an error of size at most 1. Our experiments show that for $G(n,p)$ graphs with average degree $\leq 6$, *Greedy* selects a small number of vertices of dynamic degree 2, and the probability that *Greedy* selects a vertex of dynamic degree $\geq 3$ is asymptotically 0. The general problem of determining whether a given degree two vertex belongs to an *MIS* is NP-complete. We present polynomial-time sufficient conditions for a degree two vertex to be correct. These conditions are incorporated in *Greedy*$^m$, an enhancement over *Greedy* in which ties between dynamic degree 2 vertices may be broken optimally to yield a correct vertex. Since we can always remove any vertex with dynamic degree 0 or 1, we assume that the dynamic degrees of all vertices are at least 2.

A **straight path** is a path containing only degree two vertices. An **interval** is a straight path which is not a proper subpath of another straight path. A straight path is **even** (resp. **odd**) if its length is even (resp. odd). (The smallest interval consisting of one vertex is even and has length 0.) The **end-points** of an interval are the first and last vertices of the interval. A **connector** is a vertex of degree $> 2$ adjacent to the end-points of an interval. A **leaf-interval (or leaf)** is an interval with length $> 0$ whose end-points are adjacent to the same connector. A **loop-interval (or loop)** is an interval whose end-points are adjacent.

Define a bi-partite graph $Q = (I \cup C; F)$ as follows: the vertex set is $I \cup C$, where $I$ is the set of intervals and $C$ is the set of connectors; the edge $(p, c)$ is in the edge set $F$ *iff* $p$ is an interval in $I$ and $c$ is a connector adjacent to at least one end-point of $p$. The even (resp. odd) subgraphs $Q^{ev}$ (resp $Q^{odd}$) are the induced subgraphs after removing the even (resp. odd) intervals from $Q$. We now give sufficient conditions for a degree 2 vertex to be correct (proofs are postponed to a full version of the paper).

**Theorem 1.** *If $v$ is an end-point of a loop or leaf, then $v$ is a correct.*

**Theorem 2.** *Let $\mathcal{C}$ be a connected component of $Q^{ev}$ and $v$ an end-point of an interval in $\mathcal{C}$. Then $v$ is correct if one of the following hold:*

1. *$\mathcal{C}$ contains a cycle;*
2. *two connectors from $\mathcal{C}$ are adjacent in $G$;*
3. *there is an odd interval whose connectors are both vertices in $\mathcal{C}$.*

### 2.1 *Greedy*$^m$: Modifying the Greedy Algorithm

Theorem 2 indicates how we can modify *Greedy* to obtain a better algorithm. If a degree 2 vertex $v$ is encountered, we may try to determine if it is correct. If it is correct, then we may safely take $v$. A polynomial-time scan through all degree 2

vertices can be used to find a correct degree two vertex if one exists. If no degree 2 vertex is found, then the degree 2 vertex that is a member of the longest path is selected, resulting in a mistake of at most 1 for all the vertices selected in the path. Thus, *Greedy*$^m$ differs from *Greedy* (described in the introduction) only in how it breaks ties among dynamic degree 2 vertices.

### 2.2 Efficiently Generating Random Graphs

For dense graphs (fixed edge probability $p$), the expected size of the input is $\Omega(n^2)$. Thus, one cannot significantly improve upon the algorithm that checks each of the $\binom{n}{2}$ pairs to generate $\binom{n}{2}p$ of edges independently with probability $p$.

When $p = o(1)$, generating the edges by examining all pairs of vertices is excessive since only $O(n^2p)$ edges need be generated. It is therefore more efficient to first generate the number of edges $M$, and then select the specific edges. For sparse graphs, with $p = d/(n-1)$, this approach yields considerable computational savings.

Specifically, the number of edges in the graph $M$ is a binomial random variable $B(p, \binom{n}{2})$. The following code can be used to generate $M$ efficiently,

```
1: x = 0; y = 0; N = (n 2); c = ln(1 − p);
2: if c = 0 then
3:    return 0;
4: while TRUE do
5:    Generate a uniform random variate u ∈ [0, 1];
6:    y ← y + ⌊ln u/c⌋ + 1;
7:    if y ≤ N then
8:       x ← x + 1;
9:    else
10:      BREAK;
11: return x;
```

The runtime of the algorithm is $O(x)$, where $x$ is the output value for $M$. Since the expected value of $M$ is $\binom{n}{2}p = O(n)$ (sparse graphs), we see that generating $M$ is quite efficient.

Given $M$, it is now a simple matter to uniformly pick $M$ edges from the available $\binom{n}{2}$ edges. The following algorithm accomplishes this task,

```
1: Let S = ∅ be the set of selected edges;
2: while |S| < M do
3:    Generate a uniform integer random variate u ∈ [1, (n 2)];
4:    if edge u ∉ S then
5:       S ← S ∪ u;
```

The probability that a sampled edge is not placed in $S$ is $O(1/n)$ for sparse graphs. Therefore in $M = O(n)$ samples, $O(1)$ edges are rejected. Thus, $O(2M)$

samples should suffice. If the set $S$ is stored in a data structure that allows for efficient searching in time $O(\log |S|)$, for example a balanced binary search tree, then the entire algorithm has expected run time $O(M \log M)$.

## 3 Experimental Results

The experiments described here involve running *Random*, *Greedy*, and *Greedy*$^m$ on randomly generated graphs from different domains: $G(n,p)$ for a fixed $p$; $G(n,p)$ for $p = d/(n-1)$, where $d$ is an integer in [1,10]; and 3-regular graphs. All our experiments are reproducible, since we use a seeded pseudo-random number generator. The machines used to run the experiments range in size and power from Sun Ultra 10 workstations with 256MB RAM running Solaris, to Intel IA64 based workstations with 16GB RAM running Linux.

### 3.1 *Greedy* Versus *Random* for Fixed Edge Probability

It is not known whether *Greedy* outperforms *Random* on $G(n,p)$ graphs for fixed $p$ – no theoretical or experimental evidence to the contrary is available. We present experimental data comparing *Random* to *Greedy* on $G(n,p)$ graphs for fixed $p$. We show the results for $p = 0.1$ and $n$ up to 100,000 (other values of $p$ gave similar results).

For each $n$, we generated 1000 graphs, and compared the average independent set size found by *Greedy* to that found by *Random*. Let $g(n,p)$ (resp. $r(n,p)$) be the average independent set size found by *Greedy* (resp. *Random*). We are interested in the ratio $R(n,p) = g(n,p)/r(n,p)$, which is plotted in Figure 1(a). Figure 1(a) shows that $R(n,p)$ is not a monotone function (as was expected initially) but a unimodular function with the maximum near $n = 1000$.

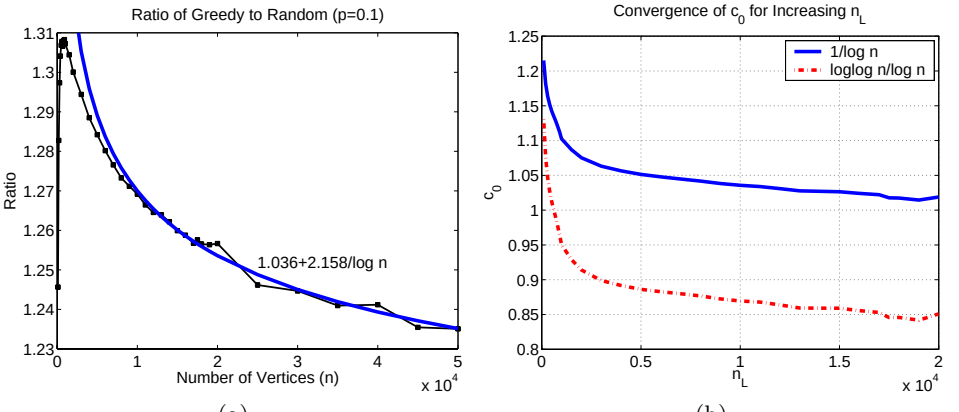



**Fig. 1.** (a) $R(n,p)$ for $p = 0.1$, $n \leq 50000$. Also shown is the fit to $1/\log n$ for $n \geq 10,000$. (b) Convergence of $c_0(n_L)$ for $f(n) \in \{1/\log n, \log\log n/\log n\}$

Although *Greedy* consistently finds a larger independent set than *Random*, and $R(100,000, 0.1) > 1.15$, the analysis of $R(n, 0.1)$ suggests that indeed $R(n, 0.1) \to 1$, as $n \to \infty$. We analyze the experimental data for $R(n, 0.1)$ by fitting a curve of the form $R(n, p) = c_0 + c_1 f(n)$, where $f(n) \to 0$. We have tried many functional forms for $f(n)$ (yielding similar results), but the functional forms $f(n) = 1/\log n$ and $f(n) = \log\log n/\log n$ are suggested by the theoretical analysis of *Random*, and so we only show the results for these two functional forms. For $n \geq n_L$, we can obtain the optimal least squares fit for $c_0$, denoted $c_0(n_L)$. As $n_L$ gets larger, $c_0(n_L)$ should converge to the true value of $c_0$. We show the convergence of $R(n, 0.1)$ as a function of $n$ and the convergence of $c_0(n_L)$ in Figure 1. From Figure 1(b) it appears that $c_0$ converges to 1 for $f(n) = 1/\log n$. $f(n) = \log\log n/\log n$ does not give a reasonable value for $c_0$ as it is below 1. Thus the data indicates that $c_0 \to 1$ and that $f(n) = 1/\log n$. In particular, this indicates that *Greedy* only outperforms *Random* by a constant number of vertices.

### 3.2 *Greedy* Versus *Random* for Fixed Average Degree

Frieze and Suen [7] show that for random 3-regular graphs, *Greedy* constructs an independent set of the size at least $(6\log\frac{3}{2} - 2)n \approx 0.432791n$ w.p.1., which is asymptotically larger than the independent set constructed by *Random* (the asymptotics for *Random* can be found in [2]). The data in Table 1 show that the lower bound on the performance of *Greedy* is tight. Each entry is an average over 1000 randomly generated graphs.

**Table 1.** Size of independent sets found in 3-Regular graphs

| $n$ | *Random* | *Greedy* |
|---|---|---|
| 1,000 | 0.3749 $n$ | 0.4320 $n$ |
| 5,000 | 0.3751 $n$ | 0.4324 $n$ |
| 10,000 | 0.3750 $n$ | 0.4326 $n$ |
| 50,000 | 0.3750 $n$ | 0.4327 $n$ |

Our experiments on random graphs with fixed average degree show a similar performance gain for *Greedy* over *Random*. Figure 2 shows the ratio $R(n, p)$ for $d = 1, 2, 3, 4, 5, 6$; and $n \in [1000, 50, 000]$; each data point is an average over 1000 graphs. The main conclusion: on average, *Greedy* outperforms *Random* by a multiplicative constant which is increasing in $d$ for small $d$.

### 3.3 Dynamic Degrees

We now consider the dynamic degrees of the vertices in the independent sets found by *Greedy* and *Random*. The dynamic degrees allow us to establish nontrivial upper bounds on the independence number of the input-graph. The dynamic degree distributions with respect to *Greedy* for random average degree 3

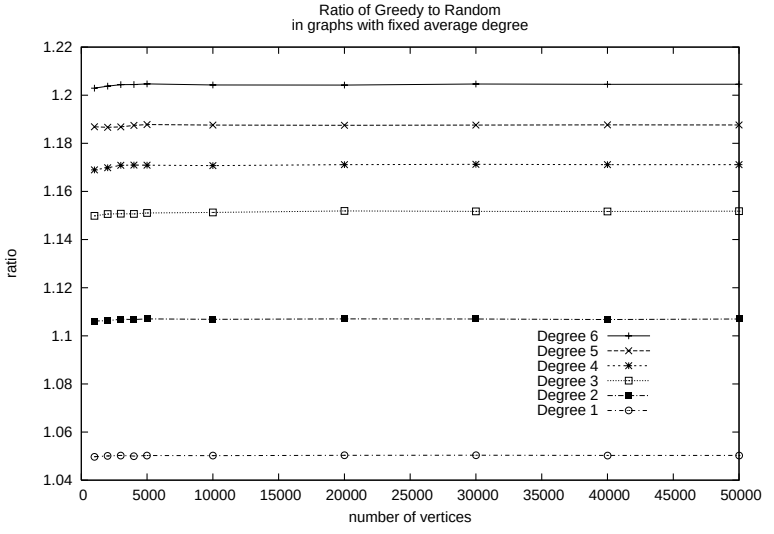



**Fig. 2.** Ratio of *Greedy* vs. *Random* for graphs with fixed average degree

**Table 2.** Distribution of dynamic degrees for graphs with average degree 3

| n | $I_0/n$ | $I_1/n$ | $I_2/n$ |
|---|---|---|---|
| 1000 | 0.076 | 0.442 | 0.013 |
| 5000 | 0.073 | 0.449 | 0.009 |
| 10000 | 0.073 | 0.451 | 0.009 |
| 50000 | 0.072 | 0.452 | 0.008 |
| 100000 | 0.072 | 0.452 | 0.008 |

**Table 3.** Distribution of dynamic degrees for graphs with average degree 4

| $n$ | $I_0/n$ | $I_1/n$ | $I_2/n$ |
|---|---|---|---|
| 1000 | 0.025 | 0.363 | 0.083 |
| 5000 | 0.023 | 0.368 | 0.081 |
| 10000 | 0.023 | 0.368 | 0.080 |
| 50000 | 0.022 | 0.369 | 0.080 |

**Table 4.** Distribution of dynamic degrees for graphs with average degree 5

| $n$ | $I_0/n$ | $I_1/n$ | $I_2/n$ |
|---|---|---|---|
| 1000 | 0.010 | 0.255 | 0.160 |
| 5000 | 0.008 | 0.260 | 0.157 |
| 10000 | 0.008 | 0.260 | 0.157 |
| 50000 | 0.008 | 0.262 | 0.156 |

**Table 5.** Distribution of dynamic degrees for graphs with average degree 6

| $n$ | $I_0/n$ | $I_1/n$ | $I_2/n$ | $I_3/n$ |
|---|---|---|---|---|
| 1000 | 0.004 | 0.161 | 0.224 | 0.0002 |
| 5000 | 0.003 | 0.165 | 0.222 | 0.0000 |
| 10000 | 0.003 | 0.166 | 0.222 | 0.0000 |
| 50000 | 0.003 | 0.166 | 0.221 | 0.0000 |

graphs are shown in Table 2. Tables Tables 3, 4 and 5 give the corresponding distributions for average degrees 4, 5 and 6 respectively.

These statistics (averaged over 1000 graphs) were collected from the same data used in the previous section. Although there are a few vertices of dynamic degree 3 for average degree and 6 graphs, it is clear that the number of such vertices is approaching zero as the graph size increases.

**Table 6.** Accuracy of *Greedy* on random graphs with a fixed average degree

| Average Degree | Approximation Ratio | | | |
|---|---|---|---|---|
| | $n = 1000$ | $n = 5000$ | $n = 10000$ | $n = 50000$ |
| 1 | 1.0000784 | 1.0000195 | 1.0000077 | 1.0000014 |
| 2 | 1.0006551 | 1.0001155 | 1.0000603 | 1.0000111 |
| 3 | 1.0249426 | 1.0174993 | 1.0161715 | 1.0150454 |
| 4 | 1.1767924 | 1.1709950 | 1.1708517 | 1.1694409 |
| 5 | 1.3761916 | 1.3700855 | 1.3695010 | 1.3674423 |
| 6 | 1.5764024 | 1.5693010 | 1.5685679 | 1.5670232 |

Using the data collected above we are able to evaluate the accuracy of *Greedy* using the bound in Lemma 2. Table 6 shows the approximation ratio for *Greedy* on average degree graphs with up to 50,000 vertices. The ratio shown is the average upper bound on the size of independent set derived from the dynamic degrees over the average size found by the *Greedy* algorithm.

Our interpretation of the data presented in Table 6 is that *Greedy* is very near optimal for graphs with average degree 1 and 2, and probably 3 as well (asymptotically).

### 3.4 *Greedy*$^m$ for Average Degree 3

*Greedy*$^m$ is similar to *Greedy* except that the ties between dynamic degree 2 vertices are broken by incorporating the sufficient conditions in Theorem 2. Thus, not only will the independent set found be larger, but the bound obtained in Lemma 2 can be improved by subtracting the number of dynamic degree two vertices that are correct. The approximation ratios in Table 7 uses this improved upper bound from *Greedy*$^m$. As before, each entry is based on an average over 1000 random graphs.

**Table 7.** Approximation ratios for *Random*, *Greedy*, and *Greedy*$^m$; average degree = 3

| $n$ | *Random* | *Greedy* | *Greedy*$^m$ |
|---|---|---|---|
| 1000 | 1.183317 | 1.021652 | 1.013076 |
| 5000 | 1.170949 | 1.012581 | 1.006536 |
| 10000 | 1.171350 | 1.012373 | 1.006349 |
| 50000 | 1.166931 | 1.009917 | 1.004922 |
| 100000 | 1.166914 | 1.009740 | 1.004830 |

## 4 Conclusions

The main objective of this research is the development of a database of experimental results to aid the theoretical investigation of the problem of constructing

large independent sets in random graphs. Our experiments support a conclusion that traditional randomized algorithms are not optimal on a variety of random graphs domains, and may give clues as to what results may hold and how to prove them.

Specifically, for sparse random graphs, the greedy algorithm asymptotically outperforms the randomized algorithm by a constant factor. Up to average degree 3, the modified greedy algorithm which breaks the ties among dynamic degree 2 vertices appears to be asymptotically optimal or near optimal (approximation ratio $< 1.005$).

For dense graphs, with fixed edge probability $p$, our results indicate that the greedy and randomized heuristics are asymptotically equivalent. In particular, the ratio of *Greedy* to *Random* appears to have the dependence $Ratio = 1 + c_1/\log n$. Since the asymptotics of *Random* are well known on this random graph domain, $Random = \log_{1/(1-p)} n + o(\log n)$, our results indicate that *Greedy* is asymptotically only a constant better than *Random*.

Our results indicate that for sparse graphs, the modified greedy algorithm is asymptotically superior to the randomized algorithm. However, for fixed edge probability $p$ the greedy and randomized algorithms are asymptotically equivalent. An interesting open question raised by our results is to determine the threshold $p(n)$ for the edge probability below which the greedy algorithm is superior to the randomized algorithm, and above which the two are equivalent.

## References

1. J. Aronson, A. M. Frieze, and B. G. Pitel. Maximum matchings in sparse random graphs: Karp-Sipser re-visited. *Random Structures and Algorithms*, pages 111–178, 1998.
2. B. Bollobás. *Random Graphs*. Cambridge University Press, New York; second edition, 2001.
3. B. Bollobás and P. Erdös. Cliques in random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 80:419–427, 1976.
4. R. Boppana and M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 20:180–196, 1992.
5. P. Erdös and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kut. Int. Közl*, 4:17–61, 1960.
6. A. M. Frieze and B. Read. Probabilistic analysis of algorithms. *Probabilistic Methods for Algorithmic Discrete Mathematics*, pages 36–92, 1998.
7. A. M. Frieze and S. Suen. On the independence number of random cubic graphs. *Random Structures and Algorithms*, 5:649–664, 1994.
8. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparce and bounded-degree graphs. *Algorithmica*, 18:145–163, 1997.
9. R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computatiolns*, pages 85–103, 1972.
10. R. M. Karp and M. Sipser. Maximum matchings in sparse graphs. *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computing*, pages 364–375, 1982.

11. C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Proceedings of 25th Annual ACM Symp. on Theory of Computing*, pages 286–293, 1993.
12. D. Matula. The largest clique size in a random graph. *Southern Methodist University, Tech. Report, CS 7608*, 1976.
13. D. B. West. *Introduction to Graph Theory –Second edition.* Prentice Hall, New York, 2001.

# Local Clustering of Large Graphs by Approximate Fiedler Vectors

Pekka Orponen* and Satu Elisa Schaeffer**

Laboratory for Theoretical Computer Science, P.O. Box 5400,
FI-02015 TKK Helsinki University of Technology, Finland

**Abstract.** We address the problem of determining the natural neighbourhood of a given node $i$ in a large nonunifom network $G$ in a way that uses only local computations, i.e. without recourse to the full adjacency matrix of $G$. We view the problem as that of computing potential values in a diffusive system, where node $i$ is fixed at zero potential, and the potentials at the other nodes are then induced by the adjacency relation of $G$. This point of view leads to a constrained spectral clustering approach. We observe that a gradient method for computing the respective Fiedler vector values at each node can be implemented in a local manner, leading to our eventual algorithm. The algorithm is evaluated experimentally using three types of nonuniform networks: randomised "caveman graphs", a scientific collaboration network, and a small social interaction network.

## 1 Introduction

The recent interest in the analysis of natural network data [16, 17, 24] has given rise to an array of fascinating algorithmic research issues. One key task is that of extracting natural *clusters* of nodes in a network that have a relatively high interconnectivity among themselves, and a relatively low connectivity to the rest of the network. (Also called "communities" in, e.g. [17, 18].) Most of the existing literature on this topic considers the task of finding an ideal *global* clustering of a given graph. This is, however, infeasible by present techniques in the case of very large networks such as the WWW. For large networks, an effective clustering algorithm should scale at most linearly in the number of nodes $n$, whereas global clustering methods typically scale as $m \log m$ or $mn$, where $m$ is the number of edges. In the case of the WWW, where $n$ and $m$ are currently in the order of several billions, such methods are quite inadequate. The fastest global algorithms can currently deal with networks containing up to maybe a few millions of nodes [13, 17, 18]. An added complication with online networks such as the WWW is that not all the nodes are directly accessible, and the graph can only be explored "on demand".

In many applications it would in fact be sufficient to know the relevant cluster of a given source node, or maybe a group of nodes. Some recent papers, such as [20, 22, 25]

* Research supported by the Academy of Finland, grant 204156. E-mail: pekka.orponen@tkk.fi

** Research supported by the Academy of Finland, grant 206235, and the Nokia Foundation. E-mail: elisa.schaeffer@tkk.fi

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 524–533, 2005.
© Springer-Verlag Berlin Heidelberg 2005

address also this more limited goal. E.g. in [20, 22], a parameter-free local clustering quality measure is optimised using simulated annealing; the computational effort needed to obtain the cluster of a given source node is quite modest and — most importantly — independent of the total size of the network, and the results seem to be quite robust with respect to variations in the annealing process.

One fascinating aspect of the clustering problem is that it is not even clear how the notion of a "natural cluster" of nodes in a graph should be defined. It is usually apparent to the human eye what the "correct" or at least "reasonable" clustering of a given node neighbourhood is, but this intuition is difficult to make precise in a way that could be reliably automated. The clustering quality measure in [20, 22] is robust, easily computable and gives good results, but is somewhat heuristic. In the general literature, spectral and conductance-based notions are preferred on conceptual grounds [7, 8, 10, 11, 12, 14, 19, 21], but are computationally demanding. (See, however, [15] for a distributed algorithm for decentralising the computational load.) Also flow-based and other more heuristic approaches have been proposed; see [1, 9] for overviews and comparisons.

In [25] the clustering task is formulated, with the goal of efficient computation, as a problem of determining voltage levels in an electrical circuit with unit resistances corresponding to the edges of the original network. The source node is fixed at a high potential and a randomly selected target node at low potential; an approximate solution to the Kirchhoff equations is computed by an iteration scheme, and the eventual cluster of the source node is deemed to consist of those nodes whose voltages are "close" to the high value. The possibility that the target node is accidentally selected from within the natural cluster of the source node is decreased by repeating the experiment some small number of times and determining cluster membership by majority vote.

This electrical circuit analogue appears to have been first suggested in [18], where however the aim is to compute a global clustering of a given network by considering all possible source-target pairs, and for each pair solving the Kirchhoff equations exactly by explicitly inverting the corresponding Laplacian matrix. (We note that since solutions of the Kirchhoff equations can be decomposed in terms of the eigenvectors of the circuit graph Laplacian, this method is actually also a variant of the spectral partitioning techniques.)

In Section 2, we present our local clustering algorithm, which improves on [18, 25] by eliminating the need for arbitrary "target" nodes, and by making the connection to spectral methods explicit. Section 3 discusses our experiments with the method. Section 4 summarises the work and addresses directions for further research.

## 2 Approximate Computation of Fiedler Vectors

We continue the analogue of representing cluster membership values as physical potentials, but eliminate the unnatural choice of random "target" nodes by basing our model on diffusion in an *unbounded* medium rather than the electrical closed-circuit model. Thus, given a graph $G$ and a source node $i$, we fix $i$ at a constant potential level, which we choose to be zero, and consider the solution to the discrete Dirichlet problem on $G$ with this single-node boundary condition [3, p. 128]. For clustering purposes, we find an eigenvector $u$ corresponding to the smallest eigenvalue $\sigma_1$ of the respective Dirichlet

matrix, i.e. the Laplacian matrix of $G$ with row and column $i$ removed [3, 4]. This eigenvector $u$, the *(Dirichlet-)Fiedler* vector of $G$, will now assign potential values $u(j)$ close to 0 for nodes $j$ that are within a densely interconnected neighbourhood of the source node $i$, and larger values for nodes that have sparser connections to the source. The method obviously generalises to starting from a larger set of source nodes, if desired.

An alternative point of view [5] is to consider a simple random walk on the graph $G$: from each node in $G$, move to one of its neigbours with uniform probability. Modify the transition matrix $P$ of this Markov chain by making node $i$ absorbing; denote the new transition matrix by $P'$. Then our Fiedler vector $u$ is the right eigenvector associated to the second largest eigenvalue of $P'$, i.e. the (super)harmonic potential associated to the most slowly decaying mode of this absorbing chain.[1]

It is shown in [3, 4] that the Fiedler vector $u$ associated to (the $i$-absorbing simple random walk on) graph $G$ can be obtained by minimising the degree-adjusted Rayleigh quotient:

$$\sigma_1 \quad = \quad \inf_u \frac{\sum_{j\sim k}(u(j)-u(k))^2}{\sum_j \deg(j)\cdot u(j)^2}, \tag{1}$$

where the infimum is computed over vectors $u$ satisfying the boundary condition $u(i) = 0$ at the source node(s). (The notation $j \sim k$ is an abbreviation for $(j, k) \in E$.) This representation enables the approximation of $u$ by a local algorithm, not requiring the full adjacency matrix of the network.

Since we are free to normalise our eventual Fiedler vector $u$ in any way we wish, we can constrain the minimisation to vectors $u$ that satisfy, say,

$$\|u\| \doteq \sum_j \deg(j)\cdot u(j)^2 = 2m,$$

where $m$ is the total number of edges in $G$. The exact value of the normalisation constant does not actually matter: $2m$ is chosen here because it is an upper bound on the value of the sum if all components $u(j)$ are between 0 and 1.

Thus, the task becomes one of finding a vector $u$ that satisfies:

$$u \quad = \quad \text{argmin}\left\{ \sum_{j\sim k}(u(j)-u(k))^2 \;\mid u(i)=0,\ \|u\|=2m \right\}. \tag{2}$$

We can solve this task approximately by reformulating the requirement that $\|u\| = 2m$ as a "soft constraint" with weight $c > 0$, and minimising the objective function

$$f(u) \quad = \quad \frac{1}{2}\sum_{j\sim k}\Big(u(j)-u(k)\Big)^2 + \frac{c}{2}\cdot\left(2m - \sum_j \deg(j)\cdot u(j)^2\right) \tag{3}$$

by gradient descent. Since the partial derivatives of $f$ have the simple form

[1] As is well known [2], the largest eigenvalue of a Markov chain transition matrix is always 1, with left eigenvector representing the stationary distribution $\pi$ and right eigenvector the potential $(1, 1, \ldots, 1)$, both up to normalisation. In the present case, $\pi = (0, \ldots, 0, 1, 0, \ldots, 0)$, with 1 in the $i$th position. The second largest eigenvalue dominates the convergence rate of the chain, and the corresponding left and right eigenvectors indicate the most slowly converging mode.

$$\frac{\partial f}{\partial u(j)} \quad = \quad -\sum_{k\sim j} u(k) + (1-c)\cdot \deg(j)\cdot u(j), \tag{4}$$

the descent step can be computed locally at each node, based on information about the $u$-estimates at the node itself and its neighbours:

$$\tilde{u}_{t+1}(j) \quad = \quad \tilde{u}_t(j) + \delta \cdot \big(\sum_{k\sim j} \tilde{u}(k) - (1-c)\cdot \deg(j)\cdot \tilde{u}(j)\big), \tag{5}$$

where $\delta > 0$ is a parameter determining the speed of the descent.

Assuming that the natural cluster of node $i$ is small compared to the size of the full network, the normalisation $\|u\| = 2m$ entails that most nodes $j$ in the network will have $u(j) \approx 1$. Thus the descent iterations (5) can be started from an initial vector $\tilde{u}_0$ that has $\tilde{u}_0(i) = 0$ for the source node $i$ and $\tilde{u}_0(k) = 1$ for all $k \neq i$. The estimates need then to be updated at time $t > 0$ only for those nodes $j$ that have neighbours $k \sim j$ such that $\tilde{u}_{t-1}(k) < 1$.

Balancing the constraint weight $c$ against the speed of gradient descent $\delta$ naturally requires some care. We have obtained reasonably stable results with the following heuristic: given an estimate $\bar{k}$ for the average degree of the nodes in the network, set $c = 1/\bar{k}^2$ and $\delta = c/10$. The gradient iterations (5) are then continued until all the changes in the $u$-estimates are below $\varepsilon = \delta/10$. The $\tilde{u(j)}$ values are thresholded at 1, so that if the right hand side of equation (5) suggests a value greater than this, then a value of 1 is used in the update instead. Occasionally equation (5) may suggest also negative $u$-estimates, but this we have taken as an indication of a too rapid descent, and have restarted the run with a smaller value of $\delta$.

The eventual (approximate) Fiedler values thus represent the degree of membership of each node $j$ in the cluster of node $i$. A fully automated clustering system needs to still determine a good cluster boundary for node $i$, based on these values. This is a simple one-dimensional two-classification task that can in principle be solved using any of the standard pattern classifiers, such as the $k$-means algorithm [6]. However since we wish to maintain the locality of our method also at this stage, the most obvious implementations of these algorithms are not acceptable to us. (We have not yet looked into the possibility of localising the standard classifiers.) One simple local approach would be to just threshold the potentials as in [25], but we prefer not to introduce any additional instance-specific parameters to the algorithm.

Rather, we choose to follow the approach of [22, 23] of defining a locally computable cluster quality measure and optimising it by some local process — currently by a simulated annealing computation that modifies (expands or contracts) a candidate cluster one node at a time, with a time-increasing preference towards modifications that improve cluster quality. Given a source node $i$ and a candidate cluster $S$ containing $i$, a natural family of quality measures is provided by the *weighted Cheeger ratios* [3, p. 35]:

$$h_w(S) \quad = \quad \frac{\sum_{j\in S}\sum_{k\sim j, k\notin S} w(j,k)}{\sum_{j\in S}\sum_{k\sim j} w(j,k)}, \tag{6}$$

where $w(j,k)$ is an appropriate nonnegative edge weight function. Clusters $S$ with low Cheeger ratios have low (weighted) extracluster connectivity, and high (weighted)

intracluster connectivity, as is to be intuitively expected of a good cluster. Thus, aiming to minimise this ratio seems like a reasonable thing to do, and is also justified by general isoperimetric principles. In our experiments, edge weights determined as $w(j,k) = (|u(j) - u(k)|)^{-1}$ seem to lead to natural clusters in different types of networks, and are also intuitively appealing.

## 3 Experiments

We report on tests of our local Fiedler clustering method on three types of networks: randomised "caveman graphs" with 144 and 1533 nodes, a "collaboration graph" representing the pairwise coauthorships of 503 mathematicians and computer scientists, and a small social interaction graph, so called "Zachary's karate club network" often used to illustrate also other network clustering algorithms (e.g. [16, 18, 25]).

The synthetic caveman graphs (cf. Figure 1) were generated according to a probabilistic variation of the deterministic construction given in [24, p. 103]. Whereas the recipe in [24] stipulates that a caveman graph of size $n = rk$ and cavesize $k$ consist of exactly $r$ copies of a $k$-clique connected together into a cycle in a specific way, our construction gives only probabilistic parameters for the expected size, number and connection densities of the caves, resulting in a somewhat more natural family of test graphs with nevertheless predictable clustering properties. (The precise graph generation method is given in [23, p. 94].)

Figure 1 represents the results of the approximate Fiedler vector calculations on a 144-node caveman graph, starting from three different source nodes. For visual effect, the nodes are colour-coded so that dark colours correspond to small approximated Fiedler potential values, with the source node in each case coloured black. The parameter values used in this case were the standard ones derived from $\bar{k} = 6.14$ (i.e. $c = 0.027$, $\delta = 0.003$, $\varepsilon = 0.0003$). As can be seen, the method discerns the natural clusters embedded in the graph quite distinctly. The nodes selected by the Cheeger ratio heuristic for the relevant clusters in each of the three cases are indicated by thickened node boundaries; also the clusters determined in this manner can be seen to correspond to the natural ones. (The smaller, 144-node graph was chosen here merely for illustra-

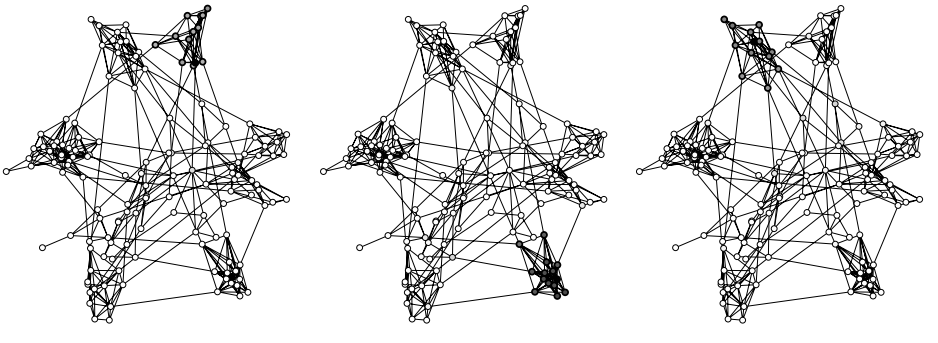


**Fig. 1.** Local Fiedler clusters in a 144-node caveman graph

tive purposes. The results on the bigger, 1533-node graph are qualitatively similar, but the graph is too large to be represented in a drawing.)

Another perspective on the data is provided by Figure 2, where the approximate Fiedler vector corresponding to the clustering in the middle panel of Figure 1 is presented numerically. On the left, the components of the Fiedler vector are ordered simply by node index, and on the right they are sorted in increasing order. The vertical lines indicate the nodes selected for the cluster of the source node (which in this case has index 95) by the Cheeger ratio heuristic. In our generating process for the synthetic caveman graphs, nodes deemed to belong to the same "cave" are assigned consequent indices, and hence good clusterings should exhibit the "band" structure observed on the left.

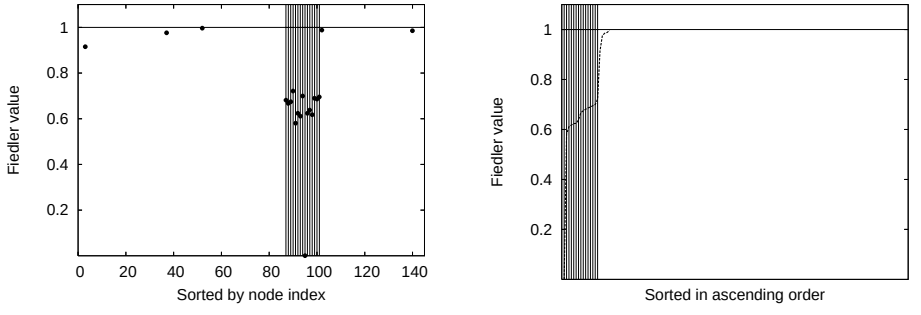



**Fig. 2.** Components of a Fiedler clustering vector

Our second test graph was extracted from the Mathematics section of the Karlsruhe Collection of Computer Science Bibliographies.[2] The raw coauthorship data was cleaned in various ways to eliminate nonperson authors such as institutes and committees, unify spellings of authors' names, etc. (details given in [23, p. 99]). The resulting 503-node graph is shown in the left panel of Figure 3. The right panel shows three small collaborative clusters identified by the local Fiedler clustering method, starting from three distinct source nodes; the clusters are non-overlapping in the sense that none of the source nodes gave values less than 1.0 for any of the members of the other two clusters.

Figure 4 presents a close-up view of the three clusters indicated in Figure 3, with distant and overlapping nodes rearranged to allow a better view of the structure of the induced subgraphs. Also in this instance, our standard parameter values based on $\bar{k} = 3.3$ were used.

The third example (Figure 5) represents the friendship relations among 34 members of a university karate club studied by the anthropologist Zachary in 1977 [26] (cited in [16]). Due to internal tensions, the club split into two during Zachary's two-year study period, and some of the members joined the former instructor of the club in establishing a new organisation. In the graphs of Figure 5, the actual partition of the club is indicated

[2] `http://liinwww.ira.uka.de/bibliography/Math/`, accessed 2 Dec 2002.

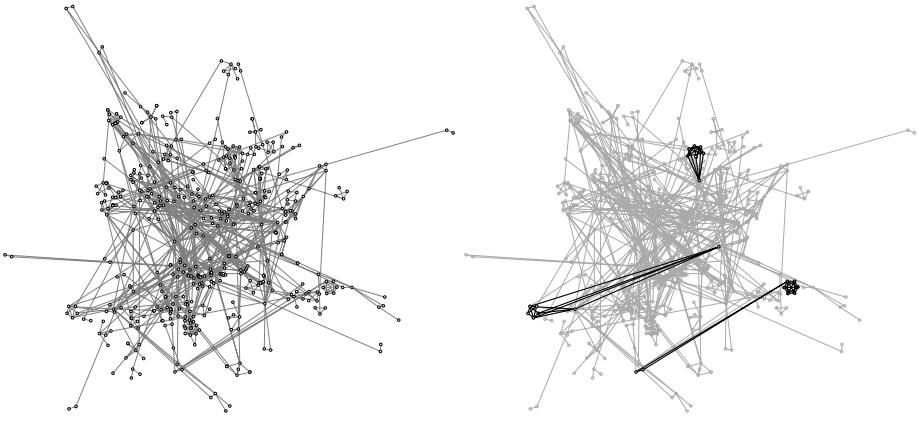


**Fig. 3.** Local Fiedler clusters in a 503-node collaboration graph

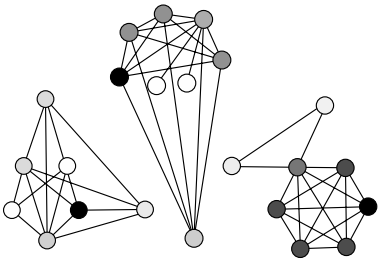


**Fig. 4.** A closeup view of the three clusters

by the shape of the nodes: square nodes correspond to the club members who stayed in the original club together with its administrator, and circular nodes correspond to the members who moved to the new club.

In the left panel of Figure 5, the nodes are coloured according to their approximate Fiedler values in the cluster of the original club's administrator, and in the right panel according to the club's instructor. As can be seen, the correlation between the shapes and colours of the nodes is quite good in both cases; only a few nodes in the middle are "undecided" as to which club they belong to, but this may correspond also to the actual social reality of the situation.

We wish to emphasize that the small size of our example graphs here is due to the requirements of illustration. The fact that our method is *local* means exactly that its running time scales relative to the size of the resulting *cluster*, and does *not* depend on the size of the ambient graph.

In fact, we have also implemented the method in such a manner that the $u$-estimates are updated according to equation (5) only as required by the optimisation process of the Cheeger clustering criterion (6). This means, firstly, that nodes that fall out of the single-edge neighbourhood of an evolving candidate cluster no longer need to be accessed, and secondly, that nodes that remain in the cluster have their $u$-estimates updated repeatedly

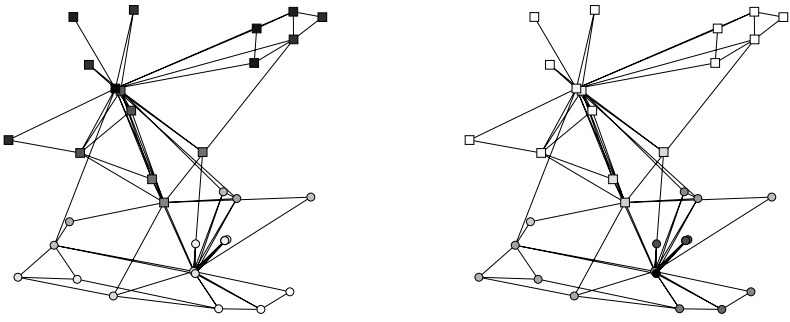


**Fig. 5.** Fiedler values in a 34-node karate club network

in connection with re-evaluations of the Cheeger criterion, thus implicitly focusing the gradient descent of the $u$-estimates to the part of the graph that is of interest for the clustering goal. This implementation saves considerably in both the working space and the running time requirements of the algorithm, without affecting the quality of the results.

## 4 Conclusions and Further Work

We presented a local method for clustering graphs based on computing their approximate Fiedler vectors and illustrated its behaviour on simple "caveman", "collaboration" and social interaction graphs. According to our experiments, the method behaves well and conforms to the intuition that arises from its analytical properties. The key characteristic of the method is that its resource requirements depend only on the size and connectivity of the resulting cluster, and *not* on the characteristics of the whole graph.

As future work, the algorithm should also be extended to work on directed graphs, in order to deal with interesting natural networks such as the WWW. Some interesting issues remain also in the area of localising standard clustering methods and comparing them to the presently used Cheeger criterion optimisation technique.

## Acknowledgments

We thank most appreciatively Mr. Kosti Rytkönen for providing us with his automatic graph drawing tool used to produce the diagrams in Figures 1, 3 and 5.

## References

1. U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. In *Proceedings of the 11th Annual European Symposium on Algorithms(ESA'03), Lecture Notes in Computer Science 2382*, pages 568–579, Berlin Heidelberg, 2003. Springer-Verlag.
2. P. Brémaud. *Markov Chains, Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer-Verlag, New York, NY, 1999.

3. F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, Providence, RI, 1997.
4. F. R. K. Chung and R. B. Ellis. A chip-firing game and Dirichlet eigenvalues. *Discrete Mathematics*, 257:341–355, 2002.
5. P. G. Doyle and J. L. Snell. *Random Walks and Electric Networks*. Mathematical Association of America, Washington, DC, 1984.
6. R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, New York, NY, 2001.
7. M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23:298–305, 1973.
8. M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–633, 1975.
9. G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee. Self-organization and identification of Web communities. *IEEE Computer*, 35(3):66–71, 2002.
10. C. Gkantsidis, M. Mihail, and E. Zegura. Spectral analysis of Internet topologies. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*, pages 364–374, New York, NY, 2003. IEEE.
11. S. Guattery and G. L. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, 1998.
12. X. He, H. Zha, C. H. Q. Ding, and H. Simon. Web document clustering using hyperlink structures. *Computational Statistics & Data Analysis*, 41(1):19–45, 2002.
13. J. Hopcroft, O. Khan, B. Kulis, and B. Selman. Natural communities in large linked networks. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 541–546, New York, NY, 2003. ACM.
14. R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):497–515, 2004.
15. D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC'04)*, New York, NY, 2004. ACM.
16. M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.
17. M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69:066113, 2004.
18. M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.
19. A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11:430–452, 1990.
20. S. E. Schaeffer. Stochastic local clustering for massive graphs. In *Proceedings of the Ninth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'05)*, Lecture Notes in Computer Science, Berlin Heidelberg, 2005. Springer-Verlag.
21. D. A. Spielman and S.-H. Teng. Spectral partitioning works: planar graphs and finite element meshes. In *Proceedings of the 37th IEEE Symposium on Foundations of Computing (FOCS'96)*, pages 96–105, Los Alamitos, CA, 1996. IEEE Computer Society.
22. S. E. Virtanen. Clustering the Chilean Web. In *Proceedings of the First Latin American Web Congress*, pages 229–231, Los Alamitos, CA, 2003. IEEE Computer Society.
23. S. E. Virtanen. Properties of nonuniform random graph models. Research Report A77, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, May 2003. URL: `http://www.tcs.hut.fi/Publications/info/bibdb.HUT-TCS-A77.shtml`.

24. D. J. Watts. *Small Worlds: The Dynamics of Networks between Ordeer and Randomness*. Princeton University Press, Princeton, RI, 1999.
25. F. Wu and B. A. Huberman. Finding communities in linear time: a physics approach. *The European Physics Journal B*, 38:331–338, 2004.
26. W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.

# Almost FPRAS for Lattice Models of Protein Folding*

## (Extended Abstract)

Anna Gambin and Damian Wójtowicz

Institute of Informatics, Warsaw University,ul. Banacha 2, 02-097 Warsaw, Poland
{aniag, dami}@mimuw.edu.pl

**Abstract.** The provably efficient randomized approximation scheme which evaluates the partition function for the wide class of lattice models of protein prediction is presented. We propose to apply the idea of self-testing algorithms introduced recently in [8]. We consider the protein folding process which is simplified to a self-avoiding walk on a lattice. The power of a simplified approach is in its ability to search the conformation space, to train the search parameters and to test basic assumptions about the nature of the protein folding process. Our main theoretical results are formulated in the general setting, i.e. we do not assume any specific lattice model. For the simulation study we have chosen the HP model on the FCC lattice.

## 1 Introduction

One approach to protein structure prediction is to simulate the backbone of $\alpha$-carbons of $n$-residue protein as a *self-avoiding walk* (SAW), where $n$ beads on a string occupy adjacent lattice sites. Lattice models of protein folding have provided valuable insights into the general complexity of protein structure prediction problem: for the variety of models it has been shown to be NP-hard and even MAX SNP hard (see e.g. [4]). These intractability results are complemented by efficient (heuristic or approximation) algorithms that construct the near-optimal protein structures.

A *self-avoiding walk* takes place on a graph, and it is a walk (i.e. a sequence of distinct graph vertices) that starts at a fixed origin. In this paper we are concerned with self-avoiding walks on some regular lattices. We call a lattice *regular* if any two lattice vertices can be mapped on the other by a translation of the lattice. Self-avoiding walks have been studied for many years and are the subject of an extensive literature, see e.g. the monograph [3].

Recently, Randall worked on uniform generating and approximate counting self-avoiding walks on lattices $\mathbb{Z}^d$, and wrote in [7]: *"[...] A generalization worth exploring is sampling self-avoiding walks with attractive or repulsive forces among*

* Research supported by Polish Research Council KBN grant 4 T11C 044 25.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 534–544, 2005.
© Springer-Verlag Berlin Heidelberg 2005

*the bonds. [...], this problems arises in computational biology in context of protein prediction. The sites of the self-avoiding walk represent carbon atoms of a long protein chain, and forces between carbon atoms which are adjacent in the lattice, but not along the protein, determine the structure of the protein. Biologists are interested in generating such 'potential' protein structures according to their likelihood in order to infer properties about actual proteins which are difficult to probe directly."* This quotation includes major motivations of our paper. Randall have posed also two problems from which we are solving here the latter.

**Notation.** We consider finite regular lattices $\mathcal{L}$ that satisfy the following conditions: (a) the number of outgoing edges from any vertex equals $\mathbf{D}$; (b)the lattice is strongly connected; (c) one of vertex is marked as an origin (zero point) and denoted by $\mathbf{0}$. Examples of such lattices are cubic $\mathbb{Z}^d$ ($\mathbf{D} = 2d$) and face-centered cubic (FCC, $\mathbf{D} = 12$).

The *length* $|w|$ of SAW $w$ is the number of its edges. The set of SAWs of length $N$ on $\mathcal{L}$ is denoted by $\mathcal{S}_N$ and the set of SAWs of length at most $N$ by $\mathcal{X}_N = \bigcup_{i=0}^{N} \mathcal{S}_i$ . The *concatenation* $w \circ v$ of two SAWs $w$ and $v$ is the walk formed by translating $v$ so that its origin coincides with the free endpoint of $w$ and appending the translated copy of $v$ to $w$. Note that $w \circ v$ need not to be self avoiding and length $|w \circ v|$ amounts $|w| + |v|$.

The *labeling of walks* $w \in \mathcal{X}_N$ is a mapping $s$ from the set $\{0, 1, \ldots, N\}$ to some finite alphabet $\Sigma$, and it labels $i$th site of walk $w$ by $s(i)$.

For $0 < \lambda \leq 1$ we associate with each SAW $w \in \mathcal{X}_N$ a weight $\lambda^{h(w)}$, where $h : \mathcal{X}_N \to \mathbb{R}_+$ is any nonnegative, superadditive function which can be evaluated for any walk from $\mathcal{X}_N$ in time polynomial in $N$ and $|\Sigma|$. The weight of a given conformation (SAW) corresponds to a free energy of a folded protein sequence.

Let $c_N$ denotes the sum of weighted SAWs of length $N$, i.e. $c_N = \sum_{w \in \mathcal{S}_N} \lambda^{h(w)}$. In this paper, we present Markov Chain Monte Carlo algorithms for two problems concerning weighted SAWs:
– sample a walk proportional to its weight (i.e., sample from Gibbs distribution);
– compute $c_N$, also known as a partition function; from this value one can derive different characteristics such as density of states, entropies and energies necessary to study the molecular properties of a model.

The running time of these algorithms is polynomial in the walk length and grows slowly with parameters controlling the accuracy and confidence levels of the estimates. Our algorithms are based on Randall approach presented in [7] and [8].

**Example.** Dill et. al. proposed ([1]) the HP lattice model for proteins. According to this model chains (SAWs) are configured on three-dimensional lattice, for example $\mathbb{Z}^3$. An alphabet $\Sigma$ consists of two letters H and P which correspond to hydrophobic and polar amino acids. In the HP model, the weight function reflects the fact that hydrophobic amino acids have (or not) a propensity to form a hydrophobic core and it depends on the number of HH contacts (i.e. non-adjacent hydrophobics that occupy adjacent grid points on the lattice).

**Organization of the paper.** The paper is organized as follows. Section 2 presents the concept of randomized approximation scheme and the ideas behind the self-testing approach. The main theoretical results are given here together with the outline of the algorithms. Next Section contains the numerical results obtained for the counting and generation problems. Section 4 is devoted to the discussion of possible extensions of this work.

## 2 Theoretical Results

We start this Section with a formal definition of efficient randomized approximation schemes. The algorithms are based on the Markov Chain Monte Carlo paradigm, i.e. on the simulation of appropriately defined Markov chain. Then we formulate the main results.

**Definition 1. A nearly Gibbs generator** *for weighted SAWs is a probabilistic algorithm which, on input* $N$ *and* $\varepsilon \in (0,1)$*, outputs a self-avoiding walk of length* $N$ *with probability at least* $1/q(N)$ *for a fixed polynomial* $q$*, such that the conditional probability distribution over walks of length* $N$ *has variation distance at most* $\varepsilon$ *from the Gibbs distribution. If this generator runs in time polynomial in* $N, \varepsilon^{-1}$ *and* $\ln \delta^{-1}$ *with probability at least* $1-\delta$*, where* $\delta \in (0,1)$ *is a part of input, then we call it* **almost fully polynomial**.

**Definition 2. A randomized approximation scheme** *for the weighted problem of counting SAWs is a probabilistic algorithm which, on input* $N$ *and* $\epsilon, \delta \in (0,1)$*, outputs a number* $\tilde{c}_N$ *such that*

$$\Pr\left(|\tilde{c}_N - c_N| \leq c_N \epsilon\right) \geq 1-\delta$$

*If this scheme runs in time* $\mathcal{T}(N)$ *such that*

$$\Pr\left(\mathcal{T}(N) \leq \ p(N, \epsilon^{-1}, \ln \delta^{-1})\right) \geq 1-\delta$$

*where* $p$ *is some polynomial, then we call it* **almost fully polynomial**.

Recall that we denote by $c_i$ the sum of weighted SAWs of length $i$ (i.e. $c_i = \sum_{w \in S_i} \lambda^{h(w)}$). The following quantity affects the running time of our algorithms:

$$\alpha_N := \min_{j,k \ : \ j+k \leq N} \frac{c_{j+k}}{c_j \cdot c_k}.$$

**Theorem 1 (Main Theorem).** *There exist:*

- *a nearly Gibbs generator for weighted self-avoiding walks,*
- *a randomized approximation scheme for the weighted problem of counting self-avoiding walks*

*that run in time polynomial in* $N, \ln \varepsilon^{-1}, \ln \delta^{-1}$ *and* $\alpha_N^{-1}$ *with probability* $1-\delta$.

We propose some generalization of widely believed conjecture about SAWs ([7]). This Conjecture is being tested simultaneously with running the algorithms — this is the main idea of *self-testing* approach.

**Conjecture 1.** *For any $j, k \in \mathbb{N}$ there exists a fixed polynomial $g$ such that:*

$$c_j c_k \leq g(j+k) c_{j+k}$$

**Corollary 1.** *Assuming Conjecture 1, there exist an almost fully polynomial nearly Gibbs generator and an almost fully polynomial randomized approximation scheme for the weighted SAWs.*

### 2.1 The Markov Chain $\mathcal{M}_n$

In this section we show how to incrementally construct Markov chains $\mathcal{M}_1, \mathcal{M}_2, \ldots$ such that the $n$th chain has as its state space the set of all SAWs of length at most $n$ and it depends on some parameters $\beta_1, \ldots \beta_n \in (0,1)$. First we define the chain $\mathcal{M}_n$ and deduce its basic properties, next we analyze its rate of convergence. The approximation schemes whose existence are postulated by Corollary 1 are based on the following three ideas:

(1) by simulating Markov chain $\mathcal{M}_{n-1}$ we can determine the mixing time of the chain $\mathcal{M}_n$ necessary for its simulation during the next step;
(2) the quantity $\beta_n$ (needed for the chain $\mathcal{M}_n$) is calculated in the previous step using the chain $\mathcal{M}_{n-1}$, so we do not require the knowledge of all $\beta_i$;
(3) the self-testing procedure ensures that the algorithm runs in the polynomial time; it tests whether Conjecture 1 is true for the assumed parameters' space.

Our chain explores the space of self-avoiding walks by expanding and contracting a walk randomly over time. Transitions in this chain are allowed only between self-avoiding walks (states) whose lengths differ at most 1. Given such a neighborhood structure, a Markov chain with the desired properties is immediately obtained using the *Metropolis method*. This give us the transition probabilities $P_n$ of the Markov chain $\mathcal{M}_n$ defined by:

$$P_n(v,v') = \begin{cases} \frac{1}{\mathbf{D}} \min\left\{\lambda^{h(v')-h(v)}\beta_{|v'|}, 1\right\} & \text{if } v \prec_1 v'; \\ \frac{1}{\mathbf{D}} \min\left\{\lambda^{h(v')-h(v)}\beta_{|v|}^{-1}, 1\right\} & \text{if } v' \prec_1 v; \\ r(v) & \text{if } v = v'; \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

where $r(v)$ is a normalizing constant, $v, v' \in \mathcal{X}_n$, and $\prec_1$is the ordering on the set of all self-avoiding walks such that $v \prec_1 v'$ if and only if $v'$ extends $v$ by one step. More precisely, $v \prec_1 v'$ ($v \prec v'$) if and only if $|v|+1 = |v'|$ ($|v| < |v'|$) and the first $|v|$ steps of $v'$ coincide with $v$.

**Proposition 1.** *The stationary distribution* $\pi_n : \mathcal{X}_n \longrightarrow \mathbf{R}$ *of the Markov chain* $\mathcal{M}_n$ *is given by:*

$$\pi_n(v) = \frac{\lambda^{h(v)}}{Z_n} \prod_{i=1}^{|v|} \beta_i, \quad \text{for all } v \in \mathcal{X}_n,$$

*where* $Z_n$ *is a normalizing factor.*

We assume that $\varepsilon \in (0,1)$ and $\left|\beta_i - \frac{c_{i-1}}{c_i}\right| \leq \frac{c_{i-1}}{c_i} \frac{\varepsilon}{2N}$.

**Theorem 2.** *For the Markov chain* $\mathcal{M}_n$ *(with self-loops probabilities* $\geq \frac{1}{2}$*), starting at the empty walk* $\mathbf{0}$*, we have*

$$\tau_n(\varepsilon) = \tau_{\mathbf{0}}^{(n)}(\varepsilon) \leq K n^2 \alpha_n^{-1} \ln \frac{n+1}{\varepsilon},$$

*where* $K$ *is some positive constant.*

### 2.2 The Algorithm

**Estimating $\alpha_n$ to determine the mixing time for $\mathcal{M}_n$.** We have determined the mixing time of the Markov chain $\mathcal{M}_n$ as a function of the unknown quantity $\alpha_n$, among other quantities. Thus we are interested in upper bounding the quantity $\alpha_n^{-1}$. Of course, if we accept Conjecture 1 then we could everywhere substitute $g(n)$ for $\alpha_n^{-1}$, but we want to be independent of any conjectures. To this aim, we present an algorithm for finding this upper bound with high probability.

The procedure approximating the quantity $\alpha_n$ is shown as Algorithm 1. The main part of this procedure is the estimation the quantity $\frac{c_n}{c_i c_{n-i}}$ within ratio $1+\varepsilon_\alpha$ (with probability at least $1 - \frac{\delta}{n^3}$) for which we find an unbiased estimator $a_{n,i} = \lambda^{h(u,v)} \mathbf{1}_{[u \circ v \in \mathcal{S}_n]}$, where $i < n$ and $u, v$ are random SAWs of length $i, n-i$, respectively. To draw the sample of walks of any given length $i < n$ we simulate the Markov chain $\mathcal{M}_{n-1}$ with the stationary distribution $\pi_{n-1}$. We assume also that we have previously calculated the approximated value $\tilde{\alpha}_{n-1}$ using the same procedure. Then the procedure approximates $\alpha_n$ within ratio $1+\varepsilon_\alpha$ with probability at least $1-\delta$, because in the procedure we have $\tilde{\alpha}_n = \min\{\tilde{\alpha}_{n-1}, \min_{i \leq n} a_{n,i}\}$. Lemma 1 states the main properties of the procedure `EstimateAlpha`.

**Lemma 1.** *Let* $\varepsilon_\alpha, \delta \in (0,1)$ *and* $t_n \geq c \alpha_n^{-1} \varepsilon_\alpha^{-2} (\ln n + \ln \delta^{-1})$ *(for some constant* $c$*). Then for the procedure* ***`EstimateAlpha`*** *we have:*

$$\Pr\left(|\tilde{\alpha}_n - \alpha_n| \leq \alpha_n \varepsilon_\alpha\right) \geq 1 - \delta.$$

*Moreover, if* $\mathcal{T}_{EA}$ *(random variable) denotes a running time of this procedure then we have:*

$$\Pr\left(\mathcal{T}_{EA} = \mathcal{O}\left(n^2 \left(t_n + n \ln \delta^{-1}\right) \tau_n\right)\right) \geq 1 - \delta,$$

*where* $\tau_n$ *is the mixing time of the Markov chain* $\mathcal{M}_n$.

**Algorithm 1** `EstimateAlpha`

**Require:** $\varepsilon_\alpha$, $\delta$, $\lambda$, $s[0 \dots N]$, $\tilde{\alpha}_{n-1}^{-1}$, $\tilde{\beta}_1^{-1}$, …, $\tilde{\beta}_{n-1}^{-1}$;
**Ensure:** $\tilde{\alpha}_n$;
1: $\tilde{\alpha}_n := \tilde{\alpha}_{n-1}$;
2: **for** $i = 1, 2, \dots, \lfloor n/2 \rfloor$ **do**
3: using $\mathcal{M}_{n-1}$, generate $t_n$ random SAWs $u_j \in \mathcal{S}_i$ and $t_n$ random SAWs $v_j \in \mathcal{S}_{n-i}$;
4: **for** $j = 1, 2, \dots, t_n$ **do**
5: $X_j := \begin{cases} \lambda^{h(u,v)} & \text{if } u_j \circ v_j \in \mathcal{S}_n; \\ 0 & \text{otherwise;} \end{cases}$
6: **end for**
7: $a_{n,i} := \frac{1}{t_n} \sum_{j=1}^{t_n} X_j$;
8: $\tilde{\alpha}_n := \min\{\tilde{\alpha}_n, a_{n,i}\}$;
9: **end for**
10: **return** $\tilde{\alpha}_n$;

As a simple consequence of Lemma 1 we obtain that the running time of the procedure `EstimateAlpha` is, with high probability, polynomial in $n$, $\alpha_n^{-1}$, $\varepsilon^{-1}$, $\ln \delta^{-1}$, namely $\mathcal{O}(n^4 \alpha_n^{-2} \varepsilon^{-2})$ if we suppress logarithmic factors and assume that $\varepsilon_\alpha \leq \varepsilon$. Because we want to know the quantity $\alpha_n$ to determine the simulation time for $\mathcal{M}_n$ and we are able to approximate it within ratio $1 + \varepsilon_\alpha$ (with high probability), so we could substitute $\tilde{\alpha}_n^{-1}(1 + \varepsilon_\alpha)$ for $\alpha_n^{-1}$. Notice that $\alpha_n^{-1} \leq \tilde{\alpha}_n^{-1}(1 + \varepsilon_\alpha)$ with high probability.

**Estimating $\beta_n$.** Recall that we need the parameter $\beta_n$ to simulate the Markov chain $\mathcal{M}_n$ and we do not need it for $\mathcal{M}_{n'}$, where $n' < n$. We show here how to estimate the necessary quantity $\beta_n$ using the chain $\mathcal{M}_{n-1}$, so in consequence how to sample SAWs with repulsive forces among the bond. The procedure is sketched in Algorithm 2, where we show also how to estimate the weighted sum of SAWs of a given length (line **6**).

**Algorithm 2** `EstimateBeta`

**Require:** $\varepsilon$, $\varepsilon_\alpha$, $\delta$, $\lambda$, $s[0 \dots N]$;
**Ensure:** $\tilde{\beta}_2^{-1}, \tilde{\beta}_3^{-1}, \dots, \tilde{\beta}_N^{-1}$ and $\tilde{c}_2, \tilde{c}_3, \dots, \tilde{c}_N$;
1: $\tilde{c}_1 :=$D; $\tilde{\beta}_1^{-1} :=$D; $\tilde{\alpha}_1 := 1$;
2: **for** $n = 2, 3, \dots, N$ **do**
3: using $\mathcal{M}_{n-1}$, generate a set $Y$ of $T_n$ random SAWs $w \in \mathcal{S}_{n-1}$;
4: $E := \sum_{w \in Y} \sum_{v \succ_1 w} \lambda^{h(v)-h(w)}$;
5: $\tilde{\beta}_n^{-1} := E / T_n$;
6: $\tilde{c}_n := \tilde{c}_{n-1} \cdot \tilde{\beta}_n^{-1}$;
7: **output** $\tilde{\beta}_n^{-1}$ and $\tilde{c}_n$;
8: $\tilde{\alpha}_n :=$`EstimateAlpha`$(\varepsilon_\alpha, \delta, \lambda, s, \tilde{\alpha}_{n-1}, \tilde{\beta}_1^{-1}, \dots, \tilde{\beta}_{n-1}^{-1})$;
9: **end for**

We explain in more details how this procedure works. Assume that we have already computed good approximations of the parameter $\beta_{n'}$ for $n' < n$, so

we could use the chain $\mathcal{M}_{n-1}$. To find a good approximation $\tilde{\beta}_n$ of $\beta_n = \frac{c_{n-1}}{c_n}$ we sample $T_n$ walks of length $n-1$ (using the Markov chain $\mathcal{M}_{n-1}$), then we extend every sampled walk by one step in all directions, and we estimate the average growth of weight for this walks. We can do this by checking all $\mathbf{D}-1$ possibilities. To obtain a good approximation of $\beta_n$ with high probability we do not need too large sample size $T_n$ for any $n$ what is precisely stated in Lemma 2.

**Lemma 2.** *Let $\varepsilon, \delta \in (0,1)$, $n \leq N$ and $T_n \geq cN^2\alpha_n^{-1}\varepsilon^{-2}(\ln N + \ln \delta^{-1})$ (for some constant c). Then the procedure* `EstimateBeta` *estimates $\beta_n^{-1}$ such that:*

$$\Pr\left(\left|\tilde{\beta}_n^{-1} - \beta_n^{-1}\right| \leq \beta_n^{-1}\frac{\varepsilon}{2N}\right) \geq 1 - \frac{\delta}{N}.$$

*Moreover, let $\mathcal{T}_{EB}$ (random variable) denotes a running time of this procedure (excluding line* ***8****, i.e. the estimation of $\alpha_n$) then $\mathcal{T}_{EB}$ satisfies:*

$$\Pr\left(\mathcal{T}_{EB} = \mathcal{O}\left(N^2 T_N \tau_N\right)\right) \geq 1 - \frac{\delta}{N},$$

*where $\tau_N$ is the mixing time of the Markov chain $\mathcal{M}_N$.*

From this Lemma we have that with high probability the running time of the procedure `EstimateBeta`, excluding line **8**, is polynomial in $N$,$\alpha_N^{-1}$,$\varepsilon^{-1}$, $\ln \delta^{-1}$, namely $\mathcal{O}(N^6\alpha_N^{-2}\varepsilon^{-2}\ln\frac{N}{\delta}\ln\frac{N}{\varepsilon})$. If follows from Lemma 1 that the running time of the whole procedure `EstimateBeta` is polynomial in $N$,$\alpha_N^{-1}$,$\varepsilon^{-1}$,$\ln\delta^{-1}$ and $\varepsilon_\alpha^{-1}$. Procedure `EstimateBeta` estimates also weighted sums of SAWs of given length. Notice that $c_n = \prod_{i=1}^{n}\beta_i^{-1} = c_{n-1}\beta_n^{-1}$, so it is easy to approximate $c_n$ when we have already computed $c_{n-1}$ and $\beta_n^{-1}$.

**Lemma 3.** *For the parameters taken from Lemma 2 the procedure* `EstimateBeta` *estimates $c_n$ such that:* $\Pr\left(|\tilde{c}_n - c_n| \leq c_n\varepsilon\right) \geq 1 - \delta$.

**Conjecture independence.** Till now we have designed the algorithm which is correct without any additional assumption, however to bound its running time we need to assume Conjecture 1. The algorithm `Selftest` below tests, whether this Conjecture holds. Using it we can guarantee almost fully polynomial time for generation and counting procedures.

Recall that Conjecture 1 postulates the existence of a fixed polynomial $g$ such that: $c_j c_k \leq g(j+k)c_{j+k}$. Hence, for all $n$ we have $\alpha_n^{-1} \leq g(n)$ and this inequality yields a polynomial mixing time for the Markov chain $\mathcal{M}_n$.

The idea of self-testing is to assure with high probability that $\alpha_n^{-1} \leq g(n)$ whenever we know that $\alpha_i^{-1}$ for $i < n$. The quantity $\alpha_n^{-1}$ is approximated by procedure `EstimateAlpha` with the relative error not greater than $\epsilon_\alpha$ with probability at least $1-\delta$. Thus, if $\alpha_n^{-1} \leq g(n)$ then for our estimator $\tilde{\alpha}_n^{-1}$ the inequality $\tilde{\alpha}_n^{-1}(1-\epsilon_\alpha) \leq g(n)$ holds with the same probability. This inequality yields $\alpha_n^{-1} \leq \frac{1+\epsilon_\alpha}{1-\epsilon_\alpha}$ and explains why in procedure `Selftest` we compare $\tilde{\alpha}_n^{-1}(1-\epsilon_\alpha)$ with $g(n)$.

**Theorem 3.** *Let $n \leq N$ and $\varepsilon, \varepsilon_\alpha, \delta \in (0,1)$ where $\varepsilon_a \leq \varepsilon$. Then the procedure* `EstimateBeta` *(Algorithm 2) with the self-tester incorporated (instead of line*

**Algorithm 3** `Selftest`

**Require:** $\epsilon_\alpha, \delta, \lambda, s[0 \dots N], \tilde{\alpha}_{n-1}, \tilde{\beta}_1^{-1}, \dots, \tilde{\beta}_{n-1}^{-1}$;
1: $\tilde{\alpha}_n :=$ `EstimateAlpha`$(\epsilon_\alpha, \delta, \lambda, s, \tilde{\alpha}_{n-1}, \tilde{\beta}_1, \dots, \tilde{\beta}_{n-1})$;
2: **if** $\tilde{\alpha}_n^{-1}(1-\epsilon_\alpha) > g(n)$ **then**
3: **output** "*Warning:* conjecture fails!";
4: **end if**

***8****) runs in time polynomial in* $N, \varepsilon^{-1}$ *and* $\ln \delta^{-1}$ *with probability at least* $1-\delta$*. Moreover, assuming that no warning message has been issued for any* $i < n$*, this procedure satisfies the following properties:*

*(i) if* $\alpha_n^{-1} \le g(n)$*, then the procedure outputs a reliable numerical answer (i.e.,* $\tilde{\beta}_n^{-1}$ *and* $\tilde{c}_n$ *such that* $\Pr\left(\left|\tilde{\beta}_n^{-1} - \beta_n^{-1}\right| \le \beta_n^{-1}\frac{\varepsilon}{2N}\right) \ge 1 - \frac{\delta}{N}$ *and* $\Pr\left(|\tilde{c}_n - c_n| \le c_n\varepsilon\right) \ge 1-\delta$*) with probability at least* $1-\delta$*;*

*(ii) if* $\alpha_n^{-1} > \frac{1+\varepsilon_\alpha}{1-\varepsilon_\alpha} g(n)$*, then the procedure outputs a warning message with probability at least* $1-\delta$*;*

*(iii) if* $g(n) < \alpha_n^{-1} \le \frac{1+\varepsilon_\alpha}{1-\varepsilon_\alpha} g(n)$*, then the procedure either outputs a warning message or a reliable numerical answer.*

## 3 Numerical Results

**HP model.** We have chosen the HP model for its simplicity. It is one of the most studied simple protein model. On the other hand this model is widely believed to capture many important properties of folding process. In HP model we restrict the space of conformations to self-avoiding walks on a lattice in which lattice points are labeled by the letters H and P. Depending on the parameter $\lambda$ the energy potential (i.e. the weight function) in HP model may reflect the fact that hydrophobic amino acids have a propensity to form a hydrophobic core (for $\lambda > 1$) or not (for $\lambda < 1$). For a walk $w$ its weight is equal to $\lambda^{h(w)}$, where $h(w)$ counts the number of hydrophobics that form a topological contact.

**Selecting a lattice.** HP simulations have typically followed Dill's original choice of square lattices. Unfortunately, there is one severe consequence of the structure of the square lattice: no two amino acids can be in adjacent lattice points if the string between them is of odd length. E.g. the sequence HPHPHPHPHPH... has no HH contacts, despite the fact that such a protein has many contacts in "real space". We have decided to work with the face-centered-cubic (FCC) lattice, since it is much more accurate in modeling real proteins and lacks the parity problem of the cubic lattice. The most promising fact is that structure prediction in the FCC HP model can be used for real protein structure prediction in hierarchical approaches [11].

The FCC lattice is a close pack structure. It has alternating layers shifted so its atoms are aligned to the gaps of the preceding layer. The FCC lattice, when

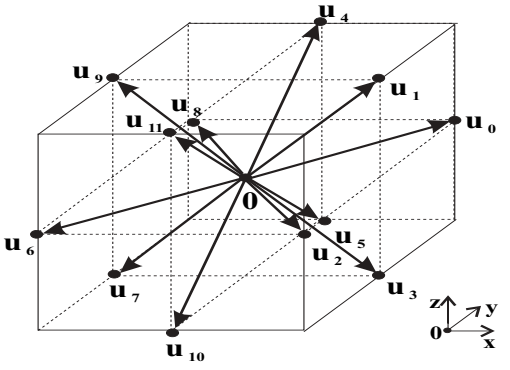



**Fig. 1.** The basis vectors for FCC lattice

viewed along the (1 1 1) direction, is composed of hexagonal layers stacked upon one another. The FCC layers cycle among the three equivalent shifted positions. The FCC lattice nodes can be viewed as the points in $\mathbb{R}^3$ whose all coordinates are integer and the sum of coordinates is even. FCC= $\{x \in \mathbb{Z}^3 : \sum_{i=1}^{3} x_i \text{ is even}\}$ (see Figure 1). **Results.** Our results are twofold: we have approximated the partition function $c_N$ for a wide range of parameters and we have tested the Conjecture 1. The experiments are performed mainly for two sequences: first one is somehow artificial sequence: $w_1$ = HPPHPHPHPHPH... and the second is the sequence: $w_2$ = PHPHPHPPPPHPHPHPPHPHPHPPPPHPHPHPP. The sequence $w_2$ has been analyzed in [6].

**Partition function.** In Table 1 we present the partition function calculated by the procedure `EstimateBeta` for sequences $w_1$ and $w_2$ and for $\lambda = 0.5$. The first column contains the approximate number of self-avoiding walks of given length. It is an interesting observation, that the value of $\beta_n^{-1}$ is very close to the

**Table 1.** The partition function estimated by the procedure `EstimateBeta`

| $n$ | $c_n$ | $c_n$ | $c_n$ | $\beta_n^{-1}$ | $\beta_n^{-1}$ |
|---|---|---|---|---|---|
| | $\lambda = 1$ | for $w_1$ | for $w_2$ | for $w_1$ | for $w_2$ |
| 2 | 132 | 123.901 | 132 | 10.3251 | 11 |
| 3 | 1403.55 | 1323.18 | 1326.17 | 10.6793 | 10.0467 |
| 4 | 14704.2 | 12897.9 | 13953.6 | 9.74764 | 10.5217 |
| 5 | 152489 | 134510 | 134065 | 10.4288 | 9.60794 |
| 6 | 1.57441e+06 | 1.28339e+06 | 1.39607e+06 | 9.54126 | 10.4134 |
| 7 | 1.61775e+07 | 1.3307e+07 | 1.43855e+07 | 10.3686 | 10.3042 |
| 8 | 1.65642e+08 | 1.25837e+08 | 1.47305e+08 | 9.45641 | 10.2399 |
| 9 | 1.69321e+09 | 1.29758e+09 | 1.50843e+09 | 10.3117 | 10.2401 |
| 10 | 1.72806e+10 | 1.22262e+10 | 1.50833e+10 | 9.422 | 9.99936 |
| 11 | 1.76098e+11 | 1.25874e+11 | 1.54251e+11 | 10.2955 | 10.2266 |
| 12 | 1.79192e+12 | 1.17901e+12 | 1.46706e+12 | 9.36657 | 9.51087 |

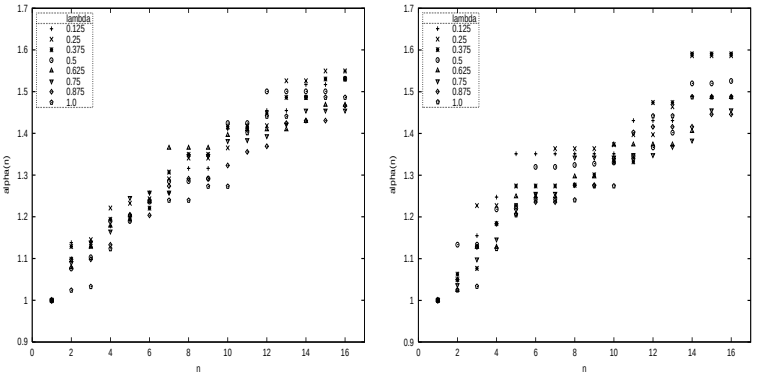



**Fig. 2.** $\alpha_n^{-1}$ for different $\lambda \in (0,1)$ and sequences $w_1$ and $w_2$

postulated value for its counterpart for $\lambda = 1$ (i.e. to the limit $\mu = \lim_{n\to\infty} c_n^{1/n}$ where $c_n$ stands for the number of self-avoiding walks of length $n$). This limit is believed [9] to be equal approximately to 10.0364.

**The Conjecture.** For the uniform case (i.e. $\lambda = 1$) the conjecture analogous to Conjecture 1 have been studied extensively in [8]. The quantity $\alpha_N := \min_{\substack{j,k \\ j+k\le N}} \frac{c_{j+k}}{c_j c_k}$. for unweighted walks has a natural interpretation: for fixed lengths $j$ and $k$, the fraction $\frac{c_{j+k}}{c_j c_k}$ is the probability that the concatenation of two walks chosen uniformly and independently forms a proper longer self-avoiding walk. In our case Conjecture 1 corresponds to the following property of the underlying lattice: the concatenation of two SAWs with lengths $j$ and $k$, with reasonable high probability forms a longer SAW and its weight does not differ much from the weight of the walk of length $j + k$ on average.

Concerning the Conjecture 1 we have obtained very promising results: the quantity $\alpha_n^{-1}$ seems to be almost independent on $\lambda$ (see Fig. 2) and the hydrophobicity pattern of the sequence. We also observe that this value increases quite slowly with $n$. This let us believe that the Conjecture 1 is true. Hence we can easily choose appropriate polynomial $g$ required in the procedure `Selftest`.

## 4 Conclusion and Further Work

We have presented provably efficient randomized approximation schemes for the problem of sampling weighted SAWs and computing the partition function. Our algorithms adopt the idea of self-testing from [8].

The natural question here is the verification of the Conjecture 1. However even in the uniform case (i.e. unweighted SAW) this conjecture remains challenging open problem.

Another interesting extension of our work is to apply much more efficient version of the Algorithm 1. Such a modification based on the idea of single

MC-trajectory simulation from [5] has been already developed by us. Preliminary results obtained for the uniform case show the essential speedup w.r.t. the standard algorithm.

## References


1. Dill, K. A., Bromerg, S., Yue, K., Fiebig, K. M., Yee, P. D., Thomas, P. D. and Chan, H. S. Principles of protein folding – A perspective from simple exact models. *Protein Science* **4** (1995), pp. 561–602.
2. Jerrum, M. R. and Sinclair, A. J. Approximating the permanent. *SIAM Journal on Computing* **18** (1989), pp. 1149–1178.
3. Madras, N. and Slade, G. "The Self-Avoiding Walk." Birkhäuser, Boston, 1993.
4. Nayak, A., Sinclair, A. and Zwick, U. Spatial Codes and the Hardness of String Folding. *Journal of Computational Biology*, **6(1)** (1999), pp. 13–36.
5. Niemiro, W. and Pokarowski, P. Faster MCMC Estimation Along One Walk. *Preprint*, 2001.
6. Pokarowski, P., Koliński, A. and Skolnik, J. A minimal physically realistic protein-like lattice model: Designing an enargy landscape that ensures all-or-one folding to a unique native state. *Biophysical Journal* (in press), 2003.
7. Randall, D. Counting in lattices: combinatorial problems from statistical mechanics. *PhD Thesis*, UC Berkeley, 1995.
8. Randall, D. and Sinclair, A. J. Self-Testing Algorithms for Self-Avoiding Walks. *Journal of Mathematical Physics* **41** (2000), pp. 1570–1584.
9. Schuster, P. and Stadler, P. F. "Discrete Models of Biopolymers." TBI Preprint, 1999.
10. Sinclair, A. J. "Algorithms for random generation and counting: a Markov chain approach." Birkhäuser, Boston, 1992.
11. Will, S. Constraint-based hydrophobic core construction for protein structure prediction in the face-centered-cubic lattice. *Proc. Pacific Symposium on Biocomputing*, 2002.

# Vertex Cover Approximations: Experiments and Observations

Eyjolfur Asgeirsson* and Cliff Stein**

Department of IEOR,
Columbia University, New York, NY
{ea367, cliff}@ieor.columbia.edu

**Abstract.** The vertex cover problem is a classic NP-complete problem for which the best worst-case approximation ratio is roughly 2. In this paper, we use a collection of simple reductions, each of which guarantees an approximation ratio of $\frac{3}{2}$, to find approximate vertex covers for a large collection of test graphs from various sources. We explain these reductions and explore the interaction between them. These reductions are extremely fast and even though they, by themselves are not guaranteed to find a vertex cover, we manage to find a $3/2$-approximate vertex cover for every single graph in our large collection of test examples.

## 1 Introduction

The vertex cover problem is a classic problem in computer science and one of the first NP-complete problems, [17, 11]. A vertex cover of a graph $G = (V, E)$ is a subset of the vertices, $C \subseteq V$, such that each edge $e \in E$ has at least one endpoint in $C$. The objective is to minimize the size of the vertex cover.

A simple greedy algorithm gives a 2-approximation for this problem [9]. In spite of many attempts to design improved approximation algorithms for vertex cover [7, 18, 3, 13] the best known approximation ratio is $2 - \theta(\frac{\log\log n}{\log n})$ for a graph with $n$ vertices [14]. Minimum vertex cover NP-hard to approximate within any factor smaller than 1.36 [8] and many people believe that there does not exist an algorithm with a fixed approximation ratio better than 2 [15, 10]. In recent years, much good work has been done on the fixed parameter version of vertex cover, where, given a fixed parameter $k$ and a graph $G$ with $n$ vertices, we can find a vertex cover of size at most $k$ in time $O(kn + 1.2832^k)$, if such a vertex cover exists [2, 5, 20].

In this paper, we look at the classic vertex cover problem and take a different approach. We use a collection of simple reductions and allow reductions that don't maintain optimality but only guarantee a worst case approximation ratio of $3/2$. Each reduction has unique properties and utilizes various specific graph structures. We will look both at the performance of specific reductions and at how they work in combinations. By combining reductions that use fundamentally different properties of the graph, we can get very beneficial interactions; these reductions create a scheme that is much more

* Research partially supported by NSF Grant DMI-9970063.
** Research partially supported by NSF Grant DMI-9970063.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 545–557, 2005.
© Springer-Verlag Berlin Heidelberg 2005

powerful than the sum of the individual methods. These interactions between different reductions create both possibilities and difficulties that we will explore further.

To test these reductions, we collected all graphs we could find on the internet and from previous works, and also generated graphs specially designed to be difficult for the vertex cover problem. In principle, our reductions do not guarantee that we will find an approximate vertex cover, however we managed to find a $3/2$-approximate vertex cover in several seconds for every single graph using only our reductions. For the graphs where we know the size of the minimum vertex cover, the actual approximation ratio was usually much lower than $3/2$; in many cases the vertex cover we found was either optimal or very close to optimal.

## 2 Graph Reductions

Our main approach is to use divide and conquer. The vertex cover problem is a hard problem, but instead of tackling the whole graph at once, we try to find chinks in its armor and solve it by breaking it into smaller and easier subproblems. For this approach to work, we need the following lemma which states that by partitioning the graph and carefully finding an approximate vertex cover for each part, we can combine them into a feasible vertex cover for the whole graph with an approximation ratio equal to the largest approximation ratio of the vertex covers for the subgraphs.

**Lemma 1.** *Assume we have a graph $G = (V, E)$ and a partition of the vertices $V = V_1 \cup V_2 \cup \ldots \cup V_k$. Let $G_i = (V_i, E_i)$ be the subgraph induced by $V_i$ and suppose that $\forall i$, $E_i \neq \emptyset$. Also assume that for each $G_i$ we have a vertex cover $VC_i^{approx}$ with the property that $VC^{approx} = \cup_i VC_i^{approx}$ is a vertex cover for $G$. Let $VC^{opt}$ be an optimal vertex cover for $G$. Then:*

$$\frac{VC^{approx}}{VC^{opt}} \leq \max_i \frac{|VC_i^{approx}|}{|VC^{opt} \cap V_i|}.$$

*Proof.* We have that $\mathrm{VC}^{\mathrm{approx}} = \cup_i \mathrm{VC}_i^{\mathrm{approx}}$ and since the vertex partition is disjoint, $|\mathrm{VC}^{\mathrm{approx}}| = |\cup_i \mathrm{VC}_i^{\mathrm{approx}}| = \sum_i |\mathrm{VC}^{\mathrm{approx}}|$. Since $E_i \neq \emptyset$, $|\mathrm{VC}^{\mathrm{opt}} \cap V_i| \geq 1$ for all $i$. Then

$$\frac{|\mathrm{VC}^{\mathrm{approx}}|}{|\mathrm{VC}^{\mathrm{opt}}|} = \frac{|\cup_i \mathrm{VC}_i^{\mathrm{approx}}|}{|\cup_i (\mathrm{VC}^{\mathrm{opt}} \cap V_i)|} = \frac{\sum_i |\mathrm{VC}_i^{\mathrm{approx}}|}{\sum_i |\mathrm{VC}^{\mathrm{opt}} \cap V_i|} \leq \max_i \frac{|\mathrm{VC}_i^{\mathrm{approx}}|}{|\mathrm{VC}^{\mathrm{opt}} \cap V_i|}$$

■

This lemma implies that we can find an approximate vertex cover for a graph by iteratively finding an approximate vertex cover for small sections of the original graph, until hopefully we have an approximate vertex cover for the whole graph. The way we will use this lemma is to iteratively break off small pieces of the graph. We are not claiming that finding a vertex cover for each subproblem implies that we've found a vertex cover of the whole graph. The lemma contains the additional restriction that the union of the vertex covers of the subgraphs is a vertex cover for the whole graph.

**Definition 1.** *An* **optimal graph reduction** *is a mapping from a graph $G = (V, E)$ to a graph $G' = (V', E')$ with the property that if we have an optimal vertex cover for $G'$ then we can find an optimal vertex cover for the original graph $G$.*

**Definition 2.** *A* $\rho$**-approximating graph reduction** *is a mapping from a graph* $G = (V, E)$ *to a graph* $G' = (V', E')$ *such that if we have an optimal vertex cover for* $G'$ *then we can find a* $\rho$*-approximate vertex cover for* $G$*.*

We will use optimal graph reductions and $\rho$-approximate graph reductions with $\rho \leq \frac{3}{2}$. An operation we will use extensively is a *vertex contraction*.

**Definition 3.** *The* **contraction of a set of vertices** $v_1, \ldots, v_k$ *to a new vertex* $v$ *is an operation where we replace the vertices* $v_1, \ldots, v_k$ *with a new vertex* $v$*, delete all edges between removed vertices and replace each edge* $(v_i, u)$ *with an edge* $(v, u)$*. Then set of vertices adjacent to* $v$ *is the union of the vertices that were adjacent to* $v_1, \ldots, v_k$*.*

When we perform a vertex contraction, we replace multiple edges that might appear with a single edge and encode information about the contracted vertices and adjacent edges so that we can recreate them later to get the original graph.

### 2.1 Optimal Graph Reductions

Almost all the optimal graph reductions that we present in this section are well known and have been used to simplify difficult graphs previously [1]. The exception to that rule is the Extended Network Flow method, which is a simple yet powerful idea that to our knowledge has not been used extensively before. For completeness we briefly explain these reductions.

#### Zero- and One-degree Vertices

*Claim.* A vertex of degree zero is not in an optimal vertex cover.

*Claim.* Let $u$ be a vertex of degree 1, and $w$ be its neighbor. Then there is an optimal vertex cover $C$ such that $w \in C$ and $u \notin C$.

#### Degree-two Vertices

*Claim.* If there is a degree-two vertex $u$ whose neighbors $v$ and $w$ are adjacent then there is an optimal vertex cover that includes both $v$ and $w$ and not $u$.

Let $u$ be a vertex of degree with $v$ and $w$ as adjacent neighbors. To cover the edge $(u, v)$, at least one of $v$ or $w$ must be in any vertex cover. By removing that vertex and all adjacent edges, $u$ becomes a degree-one vertex which means that there is an optimal vertex cover that includes $u$ and $w$ but does not include $u$.

*Claim.* If there is a degree-two vertex $u$ whose neighbors, $v$ and $w$ are non-adjacent, we can find a new graph $G'$ by contracting the vertices $u$,$v$ and $w$ to a new vertex $z$. Given a vertex cover for $G'$ with approximation ratio $\rho$, we can find a vertex cover for the original graph $G$ with the approximation ratio $\rho$. Specifically, if the vertex cover for $G'$ is optimal then we can find an optimal vertex cover for $G$.

This idea of eliminating degree-two vertices was proposed in [5]. Any optimal vertex cover of $G$ will have at least one of the vertices and at most two. If there is just one of them in the vertex cover then it must be vertex $u$. Otherwise if there are two vertices

in the vertex cover then we can select $v$ and $w$ to be in the cover by the same argument as before. Let $\text{VC}^{\text{opt}}(G)$ be the optimal vertex cover for $G$. Then $|\text{VC}^{\text{opt}}(G)| = |\text{VC}^{\text{opt}}(G')| + 1$. If $z$ is in the vertex cover for $G'$ then $v$ and $w$ will be in the vertex cover for $G$, and if $z$ is not in the vertex cover for $G'$ then only $u$ will be in the vertex cover for $G$. Since $|\text{VC}^{\text{opt}}(G)| = |\text{VC}^{\text{opt}}(G')| + 1$, any approximation ratio for a vertex cover of $G'$ holds for the vertex cover of $G$.

**Network Flow.** The vertex cover problem can be formulated as the following integer program.

$$\begin{aligned} \min \quad & \textstyle\sum_i x_i \\ \text{s.t. } & x_u + x_v \geq 1 \; \forall \; (u,v) \in E \\ & x_i \in \{0,1\} \;\; \forall \; i \in V \end{aligned}$$

By solving the linear programming relaxation of this problem we get a fractional solution. Nemhauser and Trotter [19] showed that the solution of this linear program can be used find a partial solution to the vertex cover problem. Given an optimal solution $x^*$ to the linear programming relaxation, define $P = \{u \in V | x_u > 0.5\}$, $Q = \{u \in V | x_u = 0.5\}$ and $R = \{u \in V | x_u < 0.5\}$. We can show that there is an optimal vertex cover that is a superset of $P$ and disjoint from $R$. Hence we can solve the linear programming relaxation of the vertex cover and remove all vertices that correspond to solution variables with values not equal to $1/2$. It is well known that this problem can be solved as a network flow problem [16].

**Extended Network Flow.** One of the problems with the network flow algorithm is that it tends to find solutions with many variables equal to $1/2$, even when other solutions exists with more variables either 0 or 1. In order to find as many non-half solution variables as possible, we first solve the network flow problem and remove all variables with values not equal to $1/2$. Then, using the optimal solution, we try to set each variable with value $1/2$ equal to 1 and resolve. If the objective value doesn't change we keep this new value and repeat for the remaining variables. Otherwise we set the value back to $1/2$ and try the next variable. The Extended Network Flow method works well in practice on sparse graphs where the size of the optimal vertex cover is close to half the number of vertices. To our knowledge, this approach has not been used previously. Our implementation is based on a bipartite matching and unit capacity flow algorithm from Andrew Goldberg's Network Optimization Library [12, 6].

### 2.2 Approximating Graph Reductions

While the optimal reductions can find solutions to simple examples, they are usually not enough to solve difficult graphs. We use approximating graph reductions to make further progress. The approximate reductions in this section use Lemma 1 extensively. We try to find a subgraph with certain properties, reduce this subgraph while ensuring that all edges between this subgraph and the remaining graph are covered by our reduction. By Lemma 1 we can do this repeatedly and get an approximate solution to the vertex cover problem with approximation ratio equal to the largest approximation ratio of these reductions.

**Definition 4.** *Let $G = (V, E)$ be a graph and let $G'$ be the graph obtained by contracting of a set of vertices $V_i \subset V$ to a new vertex $z$. If, given an optimal vertex cover of $G'$, we can use the mapping of $V_i$ to $z$ to find a $\rho$-approximate vertex cover for $G$, then we call $z$ a* **$\rho$-approximated vertex**. *(We will sometimes drop the $\rho$) A* **regular vertex** *is a vertex that is not an approximated vertex.*

Approximated vertices can be very useful in some situations but in others they can create difficulties. They are useful because they map a set of vertices to a single vertex and yield a simpler graph with fewer vertices and edges. The difficulty is that we cannot use them in further approximating reductions, although we can use approximated vertices in optimal reductions. Due to this, we try to use reductions that create approximated vertices only when necessary.

In Definition 4, we are using Lemma 1 when we find the $\rho$-approximate vertex cover of the original graph $G = (V, E)$ from the vertex cover of $G'$. Let $z$ be the $\rho$-approximated vertex in $G'$ and let $V_1$ be the set of vertices in $G$ that we contract to $z$, and $V_2 = V \setminus V_1$. Let $VC'$ be the vertex cover of $G'$ and $VC_2 = VC' \cap V_2$. Let $VC_1$ be the cover of $V_1$ that we can find by using the knowledge whether $z$ is in $VC'$ or not, while making sure that $VC_1$ covers all edges with one or more endpoint in $V_1$ that are not covered by $VC_2$. Then by Lemma 1 and Definition 4, the vertex cover $VC = VC_1 \cup VC_2$ is a $\rho$-approximate vertex cover of $G$.

**Degree-three Vertices.** The idea for degree-three vertices is similar to the degree-two vertices reduction. We find a degree-three vertex, $u$, with non-adjacent neighbors, $v, w$ and $z$, and contract them to a new vertex, $q$ to get a new graph $G'$. This new vertex, $q$, is a $\frac{3}{2}$-approximated vertex, so if we find an optimal vertex cover for this new graph, we can determine which of $u, v, w$ and $z$ are in an $\frac{3}{2}$-approximate vertex cover for our original graph.

**Lemma 2.** *Let $G = (V, E)$ be a graph and $u \in V$ be a degree-three vertex with $v, w$ and $z$ as neighbors, $v, w$ and $z$ are non-adjacent. Let $G'$ be a the graph we get by contracting $u, v, w$ and $z$ to a new vertex $q$. Then, if $VC^*$ is an optimal vertex cover for $G'$ then we can find a vertex cover $VC^{approx}$ for $G$ that is a $\frac{3}{2}$-approximation.*

*Proof.* Let $\text{VC}^*$ be an optimal vertex cover for $G'$ and set $\text{VC}^{\text{approx}}$ be the same as $\text{VC}^*$ for all vertices $v \in V \setminus \{u, v, w, z\}$. Now if $q$ is not in $\text{VC}^*$ then every vertex in the neighborhood of $q$ must be in the vertex cover. Hence, every vertex in the neighborhoods of $v, w$ and $z$ are in the vertex cover $\text{VC}^{\text{approx}}$ and none of $v, w$ or $z$ need to be in the vertex cover. However, we know that at least one of the vertices $u, v, w$ or $z$ must be in the vertex cover, hence we can select $u$ to be in the vertex cover $\text{VC}^{\text{approx}}$ and by Lemma 1 this is an optimal vertex cover for $G$. If $q$ is in $\text{VC}^*$ then at least one of $q$'s neighbors is not in $\text{VC}^*$ so at least one of $v, w$ or $u$ must be in the vertex cover $\text{VC}^{\text{approx}}$. In that case we would need at least two of $u, v, w$ and $z$ to be in $\text{VC}^{\text{approx}}$ to cover the subgraph induced by $u, v, w$ and $z$. We can select $v, w$ and $z$ in the vertex cover $\text{VC}^{\text{approx}}$ and get a $\frac{3}{2}$-approximation for this subgraph. Hence by Lemma 1, if $\text{VC}^*$ is an optimal vertex cover for $G'$ then $\text{VC}^{\text{approx}}$ is a $\frac{3}{2}$-approximation of the optimal vertex cover for $G$. ∎

This reduction removes four regular vertices and at least three edges from our graph while creating one $\frac{3}{2}$-approximated vertex.

**Triangles**

*Claim.* For any clique of size $k$, at least $k-1$ of the vertices are in any vertex cover.

*Claim.* If the vertices $v$,$u$ and $w$ form a triangle, then we can include all three vertices in a vertex cover for a $\frac{3}{2}$-approximation.
A triangle is a clique of size three, so at least two of the vertices must be in any vertex cover. Hence if we include all three vertices in the vertex cover we get a $\frac{3}{2}$-approximation. Removing a triangle removes three vertices from the graph along with all adjacent edges. This has the nice property that it does not create any approximated vertices.

**Four-cycles.** Assume we have a cordless cycle of length four with the vertices $v_1, v_2, v_3$ and $v_4$. For any cycle of length four, at least two of the vertices must be in a vertex cover and the only way to get exactly two of the vertices in the vertex cover is to select the opposite corners, i.e. either $v_1$ and $v_3$ or $v_2$ and $v_4$. Any other choice will give us at least three of the vertices in the vertex cover. We use this property to get a $\frac{4}{3}$-approximate reduction that removes four regular vertices and at least three edges from the graph, but at the same time creates two new $\frac{4}{3}$-approximated vertices.

**Lemma 3.** *Let $G = (V, E)$ be a graph with $v_1, v_2, v_3$ and $v_4$ as a cordless cycle of length four. Let $G'$ be a new graph where we contract $v_1$ and $v_3$ to a new vertex $z_1$ and contract $v_2$ and $v_4$ to a new vertex $z_2$. Then if $VC^*$ is an optimal vertex cover for $G'$ then we can find a $\frac{4}{3}$-approximate vertex cover for $G$.*

*Proof.* For any vertex cover of $G$, at least two of the vertices $v_1, \ldots, v_4$ must be in the cover. The only way to get exactly two of the vertices is to select either $v_1$ and $v_3$ or $v_2$ and $v_4$. Any other selection will have at least three of the vertices in the vertex cover. In the graph $G'$, there is an edge between $z_1$ and $z_2$ so at least one of these vertices must be in any vertex cover. If $z_1 \in \text{VC}^*$ and $z_2 \notin \text{VC}^*$ then that corresponds to $v_1$ and $v_3$ being in the vertex cover for $G$. Since the remaining graphs of $G'$ and $G$ are the same, this vertex cover is optimal for $G$. Similarly if $z_2 \in \text{VC}^*$ and $z_1 \notin \text{VC}^*$. Then $v_2$ and $v_4$ will be in the vertex cover for $G$ and we have an optimal vertex cover. However, if both $z_1$ and $z_2$ are in the optimal vertex cover for $G'$ then at least three of the vertices $v_1, \ldots, v_4$ must be in the optimal vertex cover for $G$. In that case, we take all four vertices in the cover and get a $\frac{4}{3}$-approximation. ∎

**Six-cycles.** The idea for cycles of length six is the same as for cycles of length four. If we have a cordless cycle of length six, we can replace it with only two vertices and get a $\frac{3}{2}$-approximation. This reduction removes six regular vertices and at least five edges from the graph, but creates two $\frac{3}{2}$-approximated vertices.

**Lemma 4.** *Let $v_1, v_2, \ldots, v_6$ be a cycle of length six in $G$. Let $G'$ be a graph where $v_1, v_3$ and $v_5$ in $G$ are contracted to a new vertex $z_1$ and $v_2, v_4$ and $v_6$ are contracted to $z_2$ while keeping all other vertices and edges the same. If $VC^*$ is an optimal vertex cover for $G'$ then we can find a vertex cover $VC^{approx}$ for $G$ that is a $\frac{3}{2}$-approximation of the optimal vertex cover.*

The proof for this is almost identical to the proof to Lemma 3..

### 2.3 Other Reductions

The crown reduction is an optimal reduction that was introduced by [1]. Here we try to find an independent set of vertices $S$ such that there exists a matching on the edges connecting $S$ and its neighborhood, $N(S)$, that matches all the vertices in $N(S)$. If we can find such a set then we can take the neighborhood set in the vertex cover and none of the vertices in the independent set. Abu Khzam et. al. [1] proved that this is an optimal reduction. However, it is easy to show that this reduction is captured by the Extended Network Flow method.

**Greedily Decreasing the Vertex Cover.** After we find an approximate vertex cover we run a simple greedy algorithm to eliminate vertices from the vertex cover. We look at each vertex in the cover and check if all its neighbors are also in the vertex cover. If that is the case then we remove this vertex from the cover.

## 3 Order of Reductions

The reductions in previous section are all simple and straightforward. We can use them in any order and given one input graph, applying them in different orders will give different results. This creates some difficulties when we try to automate the reduction, since the wrong choices can leave us in a dead end without any means of removing approximated vertices, while a different approach might have solved the problem. The greatest danger is in using 3-degree, 4-cycle and 6-cycle reductions since they leave approximated vertices that cannot be used in further approximations. The extended network flow and low degree reductions are optimal while the triangle elimination completely removes the vertices from the graph, so these methods are safe in the sense that they do not leave any approximated vertices.

The triangle elimination proved to be very effective on almost all of the graphs but it is also responsible for the largest approximation ratios. In many cases, just running triangle elimination followed by the extended network flow method was enough to find an approximate vertex cover, but often it was necessary to use multiple iterations of several reductions to get a solution. One example of this is shown in Table 1. This table shows the results of each iteration for the complement graph of the graph 's20.vc' which we created using Laura Sanchis' graph generator (see Section 4). We show the number of vertices and edges at the start of each iteration and how many vertices and edges each reduction removes from the graph. In this case, the extended network flow method and triangle elimination remove many edges which help create low degree vertices, which are eliminated with low-degree methods. During the low-degree methods, the 2-degree reduction creates more triangles, thus creating a cycle where we slowly but surely clear all the vertices.

For a few graphs we had to use the 3-degree reduction or 4-cycle and 6-cycle reduction to create a way into the graph for the optimal methods or the triangle elimination. One example of this is shown in Table 2. Here we are trying to find a vertex cover for the complement graph of the graph 'san400_0.9_1.clq' [4]. The table shows how many vertices and edges are at the start of each iteration, which reduction we used and how many vertices and edges this reduction removed from the graph. In this example, the

**Table 1.** Interaction between triangle elimination and low-degree reductions on the graph 's20.vc'. $|V|$ and $|E|$ are the number of vertices and edges at the start of each iteration while $\Delta V$ and $\Delta E$ show how many vertices and edges are removed from the graph

| Method | $\|V\|$ | $\|E\|$ | $\Delta V$ | $\Delta E$ |
|---|---|---|---|---|
| Network flow | 500 | 2000 | 3 | 17 |
| Triangles | 497 | 1983 | 147 | 1076 |
| Low Degree | 350 | 907 | 59 | 96 |
| Triangles | 291 | 811 | 24 | 205 |
| Low Degree | 267 | 606 | 54 | 67 |
| Triangles | 213 | 539 | 15 | 108 |
| Low Degree | 198 | 431 | 46 | 49 |
| Triangles | 152 | 382 | 21 | 139 |
| Low Degree | 131 | 243 | 50 | 57 |
| Triangles | 81 | 186 | 6 | 41 |
| Low Degree | 75 | 145 | 26 | 60 |
| Triangles | 49 | 85 | 3 | 11 |
| Low Degree | 46 | 74 | 33 | 50 |
| Triangles | 13 | 24 | 3 | 10 |
| Low Degree | 10 | 14 | 10 | 14 |
| Finished | 0 | 0 | - | - |

**Table 2.** The complement graph of 'san400_0.9_1.clq'. On the left we find a $3/2$-approximate vertex cover by using 3-degree reduction. On the right we use 4-cycle reduction instead. There we get stuck and cannot finish

| Method | $\|V\|$ | $\|E\|$ | $\Delta V$ | $\Delta E$ |
|---|---|---|---|---|
| Triangles | 400 | 7980 | 348 | 7864 |
| Low Degree | 52 | 116 | 12 | 21 |
| Triangles | 40 | 95 | 3 | 18 |
| Low Degree | 37 | 77 | 2 | 3 |
| 3-Degree | 35 | 74 | 15 | 27 |
| Network Flow | 20 | 47 | 20 | 47 |
| Finished | 0 | 0 | - | - |

| Method | $\|V\|$ | $\|E\|$ | $\Delta V$ | $\Delta E$ |
|---|---|---|---|---|
| Triangles | 400 | 7980 | 348 | 7864 |
| Low Degree | 52 | 116 | 12 | 21 |
| Triangles | 40 | 95 | 3 | 18 |
| Low Degree | 37 | 77 | 2 | 3 |
| 4-cycle | 35 | 74 | 10 | 17 |
| 6-cycle | 25 | 57 | 4 | 5 |
| Stuck | 21 | 52 | - | - |

three major reductions are not enough and we must use the 3-degree reduction to find a way into the graph, if we use 4-cycle reduction instead we get stuck.

After much experimentation, we settled on using the following order of reductions to automate the approximation process. We ran the extended network flow method, triangle elimination and low-degree in a loop until we had found a solution or no improvements were made during an iteration. Then we ran 3-degree, 4-cycle and 6-cycle reductions, stopping as soon as any one of them made some progress and returning to the original loop. The stopping criteria is either having processed all the vertices from the graph which gives us a vertex cover, or running 3-degree, 4-cycle and 6-cycle without any improvements. In that case we stop and must use some other methods, such as branch-and-bound, to get a solution. If we find a solution then the final step in the al-

gorithm is to use a simple greedy algorithm to eliminate unnecessary vertices from the cover. In our experiments we never had to resort to branch-and-bound, our algorithm managed to solve every single graph we found.

## 4 Experiments and Results

We did an extensive search for datasets we could use for vertex cover and also gathered all datasets we could find from previous works.

1. From the DIMACS website we took 78 graphs that were used as a challenge for the MaxClique problem [4]. We also used the complement graphs for these graphs, where we find the vertex cover for the complement graphs. These graphs are of special interest since there is a direct connection between the optimal vertex cover in a graph and the maximum clique in the complement graph.
2. From the DIMACS challenges we also obtained 60 graphs used as a benchmark for the MinColor problem, along with the complement graphs.
3. Also from the DIMACS challenges, we found 5 additional benchmark graphs. We also used the complement graphs.
4. sh2-3.dim and sh2-10.dim are graphs used in [1]. These graphs were obtained from the biological data repositories NCBI and SWISS-PROT. We also used the complement graphs.
5. From [22] we found 4 small graphs used as a benchmark for vertex cover algorithms. These graphs are of special interest because the optimal vertex cover is known. We also tried the complement graphs.
6. We generated 32 graphs using Laura Sanchis' graph generator [21]. These graphs have 500 vertices each, with number of edges ranging from 2000 to $110,000$. We split these graphs into three groups, with maximum clique sizes of 2,4 and 10. The reason we focused more on graphs with small cliques is that the triangle elimination is just too powerful on graphs with large cliques, leaving at most two vertices from a clique of size greater than two. And not surprisingly, we also tried the complement graphs of these generated graphs for additional 32 graphs.

**We solved every one of these 362 graphs, finding a $3/2$-approximate vertex cover in under 5 minutes for each one.** The running time for most of these graphs was less than a second. Moreover, in only one case did we need to use any reduction other than the extended network flow method, triangle elimination or low degree reduction. In that case, a simple 3-degree reduction finished off the graph. The triangle elimination is very powerful, on average it removed $88.78\%$ of the vertices and $93.34\%$ of the edges from each graph. The extended network flow method removed on average $5.09\%$ of the vertices and $5.56\%$ of the edges from each graph while low degree reductions averaged $6.12\%$ of the vertices and $1.10\%$ of the edges from each graph.

The experiments were run on a machine with 1.6GHz Intel Pentium 4 and 512MB RAM. The largest running time we saw for the extended network flow method was just under 4 minutes on the graph 'MANN_a81.clq' with $3,321$ vertices and $5,506,380$ edges, even though it didn't manage to remove anything from that graph. The largest running time for the triangle elimination was 56 seconds on the same graph, removing

every single vertex and all the edges. These running times are however largely related to paging, for smaller graphs with less than $500,000$ edges, the running time for each reduction was just a fraction of a second.

The MaxClique-Complement graphs are of special interest since many of them have a known optimal solution. The vertices not in the vertex cover form an independent set, which is a clique in the complement graph. Since we are trying to minimize the size of the vertex cover, it's equivalent to maximizing the clique size. Of the $75$ MaxClique-Complement graphs, we had optimal solutions to $48$ of them. The average approximation ratio was $1.043$ and the largest ratio was $1.459$.

Since our reductions perform so well and we manage to find an approximate vertex cover for every graph, we will only go into details about the most difficult and interesting problems.

**Worst approximation ratio:** The graphs that had the worst approximation ratio were the complement graphs of the MANN series. These graphs are made by Carlo Mannino and they are a part of the Maximum Clique challenge on the DIMACS webpage. These graphs are a clique formulation of the Steiner Triple Problem and they show how easy it is to create graphs with approximation ratios close to 3/2. They include a large set of independent triangles and the optimal vertex cover includes only two vertices from each triangle while the triangle elimination includes all three vertices from each triangle.

The method of greedily decreasing the vertex cover after we have found a feasible cover is very inconsistent. In the worst cases, it does not help at all while in one of the best cases we manage to decrease the size of the vertex cover on a graph with $200$ vertices from $198$ vertices down to the optimal vertex cover of $142$ vertices.

**Most challenging problems:** Some complement graphs of the 'san200' and 'san400' series from the DIMACS challenges were the most challenging, forcing us to use low degree, triangle elimination and extended network flow to get a result while still having approximation ratio over $1.2$. The only graph where we had to use 3-degree reduction is the complement graph of 'san400_0.9_1.clq'. This is shown in Table 2.

Some graphs we generated using Laura Sanches' graph generator showed very interesting behavior. The triangle elimination and the extended network flow method removed about half the vertices and then low degree elimination removed a few more. We seemed to be stuck, the graph had a few 4 and 6-cycles, but if we reduced them we couldn't finish the graph completely since we then couldn't eliminate the approximated vertices that these reductions created. However, if we went back and forth between 2-degree elimination and triangle elimination then slowly but surely we finished off the whole graph. This is shown in Table 1.

**Special bad case:** Even though we managed to find an approximate vertex cover for every graph we tested using only these simple methods, it's easy to construct graphs where our algorithm gets stuck. One example of such a graph, consisting of connected 5-cycles is shown in Figure 1. The only thing we can do in this case is to use 3-degree reduction to decrease the size of this graph down from $45$ vertices and $75$ edges to a reduced graph of $18$ vertices and $48$ edges. After that, we are stuck and must turn to branch-and-bound or some other methods to get a solution. The graph in Figure 1 is a

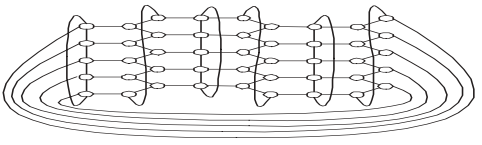


**Fig. 1.** A simple bad case

**Table 3.** Detailed performance on a small subset of the graphs

| Name | n | m | VCdec | $\lvert VC \rvert$ | ENF n | ENF m | ENF time | LD n | LD m | LD time | T n | T m | T time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Max-Clique** | | | | | | | | | | | | | |
| brock800_3.clq | 800 | 207333 | 4 | 794 | 0 | 0 | 0.161 | 2 | 1 | 0 | 798 | 207332 | 0.171 |
| C4000.5.clq | 4000 | 4000268 | 8 | 3989 | 0 | 0 | 13.734 | 4 | 3 | 0 | 3996 | 4000265 | 10.392 |
| c-fat500-2.clq | 500 | 9139 | 0 | 481 | 0 | 0 | 0.016 | 44 | 59 | 0 | 456 | 9080 | 0.005 |
| johnson32-2-4.clq | 496 | 107880 | 10 | 465 | 30 | 190 | 0.056 | 1 | 0 | 0 | 465 | 107690 | 0.101 |
| MANN_a81.clq | 3321 | 5506380 | 2 | 3318 | 0 | 0 | 229.387 | 0 | 0 | 0 | 3321 | 5506380 | 55.991 |
| p_hat1500-2.clq | 1500 | 568960 | 25 | 1473 | 0 | 0 | 0.639 | 3 | 1 | 0 | 1497 | 568959 | 0.722 |
| **Max-Clique Complement** | | | | | | | | | | | | | |
| C1000.9.clq(comp) | 1000 | 49421 | 27 | 952 | 12 | 16 | 0.165 | 28 | 49 | 0 | 960 | 49356 | 0.045 |
| hamming10-2.clq(comp) | 1024 | 5120 | 0 | 512 | 1024 | 5120 | 0.004 | 0 | 0 | 0 | 0 | 0 | 0 |
| keller6.clq(comp) | 3361 | 1026582 | 29 | 3330 | 0 | 0 | 1.536 | 1 | 0 | 0 | 3360 | 1026582 | 0.318 |
| MANN_a45.clq(comp) | 1035 | 1980 | 0 | 990 | 0 | 0 | 0.04 | 45 | 0 | 0 | 990 | 1980 | 0.002 |
| **Min-Color** | | | | | | | | | | | | | |
| DSJC1000.1.col | 1000 | 49629 | 26 | 956 | 0 | 0 | 0.172 | 28 | 28 | 0 | 972 | 49601 | 0.081 |
| flat1000_50_0.col | 1000 | 245000 | 6 | 980 | 0 | 0 | 0.272 | 22 | 56 | 0 | 978 | 244944 | 0.294 |
| le450_25a.col | 450 | 8260 | 12 | 370 | 0 | 0 | 0.034 | 123 | 332 | 0 | 327 | 7928 | 0.01 |
| R250.1.col | 250 | 867 | 8 | 190 | 10 | 41 | 0.007 | 90 | 125 | 0 | 150 | 701 | 0 |
| zeroin.i.2.col | 211 | 3541 | 0 | 84 | 108 | 3342 | 0.003 | 79 | 1 | 0 | 24 | 198 | 0 |
| **Min-Color Complement** | | | | | | | | | | | | | |
| DSJR500.1c.col(comp) | 500 | 3475 | 10 | 438 | 0 | 0 | 0.014 | 98 | 134 | 0 | 402 | 3341 | 0.001 |
| R1000.5.col(comp) | 1000 | 261233 | 186 | 808 | 0 | 0 | 0.343 | 10 | 12 | 0 | 990 | 261221 | 0.216 |
| **Sanchis' generator** | | | | | | | | | | | | | |
| s17.vc | 500 | 70000 | 124 | 375 | 0 | 0 | 0.059 | 2 | 1 | 0 | 498 | 69999 | 0.025 |
| s20.vc | 500 | 2000 | 18 | 337 | 3 | 17 | 0.07 | 278 | 393 | 0 | 219 | 1590 | 0.008 |
| s22.vc | 500 | 10000 | 16 | 454 | 0 | 0 | 0.04 | 56 | 82 | 0 | 444 | 9918 | 0.009 |
| **sh2 problems** | | | | | | | | | | | | | |
| sh2-3.dim.sh | 839 | 5860 | 0 | 246 | 839 | 5860 | 0.006 | 0 | 0 | 0 | 0 | 0 | 0 |
| sh2-10.dim.sh | 839 | 129697 | 92 | 644 | 44 | 167 | 0.244 | 78 | 135 | 0 | 717 | 129395 | 0.149 |
| sh2-10.dim.sh.pp | 726 | 69982 | 54 | 529 | 101 | 11222 | 0.213 | 124 | 384 | 0.001 | 501 | 58376 | 0.091 |
| **VC benchmarks** | | | | | | | | | | | | | |
| vtx_cov_3.gph | 100 | 200 | 3 | 56 | 8 | 40 | 0.002 | 74 | 98 | 0.001 | 18 | 62 | 0 |

simple bad case, by combining graphs similar to this and making them larger we can easily construct large graphs where the methods used here have little or no effect.

Table 3 shows how the reductions performed on a selected subset of the graphs. The first column is the name of the graph, then we have the number of vertices and number of edges. The next column shows by how much a simple greedy approach managed to decrease the size of our vertex cover. Next is the size of the vertex cover we found. Then we have the performance of the extended network flow method, triangle elimination and low degree reductions. We show how many vertices and edges each method removed from each graph, and the total time it took in seconds. This is only a small selection of the results we have, due to space constraints we cannot show all our results here.

Looking at these results it is clear that for dense graphs ('C4000.5.clq', 'MANN_a81.clq'), the triangle elimination is very powerful, while the extended network flow method works well on sparser graphs ('hamming10-2.clq(comp)', 'zeroin.i.2.col', 'sh2-3.dim.sh'). The three largest graphs show an extreme case of how the running time increases when the size of the graphs increases, the triangle elimination takes about $0.3$ seconds on a graph with just over $1,000,000$ edges while for a graph with $4,000,000$ edges this takes over $10$ seconds and then up to $56$ seconds

on a graph with around $5,500,000$ edges. Most of this time increase is due to the fact that these largest graphs do not fit into memory so the performance of the algorithms takes a hit. The time it takes to do low degree elimination is so small that it is hardly measurable even though it is very helpful in some cases ('s20.vc').

## 5 Conclusions

simple reductions where we allowed reductions that have a worst case approximation ratio of $3/2$. Even though these reductions do not guarantee that we will find a solution, we ran these reductions on a wide collection of test problems from every source we could find and by combining them we managed to find an approximate vertex cover for every single graph. Moreover, the reductions are extremely fast and easily applied, and since the bad examples have a very restrictive structure, these reductions should work well in practice.

To our knowledge, applying reductions that maintain a worst case guarantee has not been widely studied. This approach should be applicable to other problems.

## References

1. F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. *Proceedings, ACM-SIAM Workshop on Algorithm Engineering and Experiments*, 2004.
2. R. Balasubramanian, M. R. Fellows, and V. Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65:163–168, 1998.
3. R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Disc. Math.*, 25:27–46, 1985.
4. DIMACS Implementation Challenges. Center for Discrete Mathematics & Theoretical Computer Science. ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/.
5. J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.
6. B.V. Cherkassky, A.V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or Push? A computational study of Bipartite Matching and Unit Capacity Flow Algorithms. Technical Report 98-036R, NEC Research Institute, Inc., 1998.
7. P. Crescenzi and V. Kann. A compendium of NP optimization problems. http://www.nada.kth.se/theory/problemlist.html.
8. I. Dinur and S. Safra. The importance of being biased. *Proc. 34th Ann. ACM Symp. on Theory of Comp.*, pages 33–42, 2002.
9. P. Erdös and T. Gallai. On the minimal number of vertices representing the edges of a graph. *Publ. Math. Inst. Hungar. Acad. Sci.*, 6:181–202, 1961.
10. U. Feige. Vertex cover is hardest to approximate on regular graphs. Technical Report MCS03-15, Computer Science and Applied Mathematics, The Weizmann Institute of Science, 2003.
11. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Fransisco, 1979.
12. A. Goldberg. Andrew Goldberg's Network Optimization Library. http://www.avglab.com/andrew/soft.html.

13. M. M. Halldórsson. Approximating discrete collections via local improvements. *Proc. 6th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 160–169, 1995.
14. E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. *Proc. 11th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 329–337, 2000.
15. D. S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6:243–254, 1983.
16. D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS, 1997.
17. R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.
18. B. Monien and E. Speckenmeyer. Ramsey Numbers and an Approximation Algorithm for the Vertex Cover Problem. *Acta Inf.*, 22:115–123, 1985.
19. G. L. Nemhauser and L. E. Trotter. Vertex packing: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.
20. R. Niedermeier and P. Rossmanith. On Efficient Fixed Parameter Algorithms for Weighted Vertex Cover. *Journal of Algorithms*, 47:63–77, 2003.
21. L.A Sanchis. Test Case Construction for the Vertex Cover Problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15, 1994.
22. M. Truszczynski, N. Leone, and I. Niemela. Benchmark problems for answer set programming systems. http://www.cs.uky.edu/ai/benchmarks.html.

# GRASP with Path-Relinking for the Maximum Diversity Problem

Marcos R.Q. de Andrade, Paulo M.F. de Andrade, Simone L. Martins, and Alexandre Plastino*

Universidade Federal Fluminense, Departamento de Ciência da Computação
Rua Passo da Pátria, 156 – Bloco E – 3º andar – Boa Viagem
24210-240, Niterói, RJ, Brazil
{mandrade, pandrade, simone, plastino}@ic.uff.br

**Abstract.** The Maximum Diversity Problem (MDP) consists in identifying, in a population, a subset of elements, characterized by a set of attributes, that present the most diverse characteristics between themselves. The identification of such solution is an NP-hard problem. This paper presents a GRASP heuristic associated with the path-relinking technique developed to obtain high-quality solutions for this problem in a competitive computational time. Experimental results illustrate the effectiveness of using the path-relinking method to improve results generated by pure GRASP.

## 1 Introduction

Consider $P = \{p_1, \ldots, p_n\}$ a set of elements and $p_{ik}$, $k \in L = \{1, \ldots, l\}$, the $l$ attributes associated with each element $p_i$. The maximum diversity problem (MDP) [4, 6, 7] consists in identifying a subset $M$ from the population $P$, so that the $m$ elements from $M$ present the maximum possible diversity among them. The measure of diversity $d_{ij}$ between a pair of elements $(i, j)$ is calculated by a function applied on their attributes. This problem can be formulated as: Maximize $z = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$, subject to $\sum_{i=1}^{n} x_i = m$, where $x_i$ is a binary variable indicating if an element $i$ is selected to be a member of the subset $M$.

Many applications [8] can be solved using the resolution of this problem, such as human resource management, measure of biodiversity, and VLSI design.

Glover et al. [6] presented mixed integer zero-one formulation for this problem, that can be used to solve small instances by exact methods. They also show that this problem belongs to the class of NP-hard problems.

Some heuristics are available to obtain approximate solutions. Weitz and Lakshminarayanan [12] developed five heuristics to find groups of students with the most possible diverse characteristics, such as nationality, age and graduation level. They tested the heuristics using instances based on real data and implemented an exact algorithm for solving them.

* Work sponsored by CNPq research grant 300879/00-8.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 558–569, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Constructive and destructive heuristics were presented by Glover et al. [7], who created instances with different sizes of population (maximum value was 30) and showed that the proposed heuristics obtained results close (2%) to the ones obtained by the exact algorithm, but much faster.

Ghosh [4] proposed a GRASP (Greedy Randomized Adaptive Search Procedure) that obtained good results for small instances of the problem. Andrade et al. [2] developed a new GRASP and showed results for instances randomly created with a maximum population of 250 individuals. This algorithm was able to find some solutions better than the ones found by Ghosh's algorithm. Silva et al. [11] elaborated construction and local search heuristics and combined them to generate several GRASP algorithms, which were tested for instances created by the authors for populations of maximum size equal to 500 individuals. They compared their algorithms with the ones developed by Ghosh and Andrade et al. and showed that better results were obtained.

In this paper, we present GRASP heuristics for the MDP, which use path-relinking as an intensification mechanism. In Sect. 2 we describe the path-relinking techniques adopted and present the combinations of GRASP and path-relinking developed. In Sect. 3 we show computational results for these different versions of GRASP proposed. Concluding remarks are presented in Sect. 4.

## 2 GRASP and Path-Relinking

GRASP [3] is an iterative process, where each iteration consists of two phases: construction and local search. In the construction phase a feasible solution is built, and its neighborhood is explored by a local search. The result is the best solution found over all iterations.

Path-relinking is a technique proposed by Glover [5] to explore possible trajectories connecting high quality solutions, obtained by heuristics like tabu search and scatter search. A survey of GRASP with path-relinking is given in Resende and Ribeiro [9]. The pure GRASP metaheuristic is a memoryless method, because all iterations are independent and no information about the solutions is passed from one iteration to another. The objective of introducing path-relinking to a pure GRASP algorithm is to retain previous good solutions and use them as guides in the search of new good solutions.

The main objective of this paper is to evidence that the introduction of the path-relinking technique into GRASP heuristic, as an intensification strategy, may improve the results obtained for the MDP. Although, nowadays, the pure GRASP heuristics developed by Silva et al. [11] present the best results found in the literature for the MDP, we introduced path-relinking strategies to the GRASP heuristic developed by Andrade et al. [2], since the work of Silva et al. was not concluded at the time we started this research. The pseudocode for the pure GRASP of Andrade et al. is given in Fig. 1.

In line 1, the size of population $n$, the subset size $m$, the diversity matrix $DivMat$ and the maximum number of iterations $MaxIter$ are obtained. $DivMat$ contains the diversity $d_{ij}$ between each pair of elements $(i, j)$ of population $P$.

```
procedure Pure_GRASP
01.   Read_Input_Data(n, m, DivMat, MaxIter);
02.   z_best ← −∞;
03.   for i = 1, ..., MaxIter do
04.     Sol ← Build_Random_Greedy_Solution(n, m, DivMat);
05.     Sol ← Local_Search(Sol);
06.     if z(Sol) > z_best do
07.       z_best ← z(Sol);
08.       Best_Sol ← Sol;
09.     end if;
10.   end for;
11.   return Best_Sol;
```

**Fig. 1.** Algorithm for pure GRASP

The cost of the best solution found is initialized in line 2. From lines 3 to line 10, the iterations of the GRASP are executed. In line 4, a feasible solution is constructed inserting one element at each construction iteration. Initially, the sum of diversities $SD(i)$ between an element $i$ and the other elements is calculated. The initial element of the constructed solution is selected randomly from the $m$ individuals that present larger values of $SD$. Then, for each construction iteration, a restricted candidate list (RCL) is created and an element from this list is randomly selected. To build the RCL, candidates are sorted in decrescent order by a function, which evaluates the benefit of inserting an element in a partial constructed solution. Details about this construction phase may be found in [2].

After a solution is constructed, a local search phase should be executed to attempt to improve the initial solution. The neighborhood of a solution used by Andrade et al. [2] is the set of all solutions obtained by replacing an element in the solution by another that does not belong to it. In line 5, a local search is performed starting with the incumbent solution $Sol$ obtained by the construction phase. For each $i \in Sol$ and $j \in P \setminus Sol$, the improvement due to exchanging $i$ by $j$, $\Delta z(i,j) = \sum_{u \in Sol}(d_{ju} - d_{iu})$, is computed. If, for all $i$ and $j$, $\Delta z(i,j) \leq 0$, the local search is terminated, because no exchange will improve $z$. Otherwise, the elements of the pair $(i,j)$ that provides the maximum $\Delta z(i,j)$ are interchanged, a new incumbent solution $Sol$ is created, and the local search is performed again. In lines 7 and 8, the best solution found is updated if an improved solution is generated after the local search.

In order to minimize the execution time, we insert a procedure to avoid executing the local search on duplicate solutions in this original GRASP. All solutions generated after the construction phase are stored in a repository. After a solution is constructed, a search is performed in this repository and, in case that the solution is found, then the local search is not performed.

The new GRASP presented in this paper uses path-relinking as an intensification strategy. An elite set $E$ is maintained, in which good solutions found in GRASP iterations are stored to be combined with solutions created by other GRASP iterations. The path-relinking is activated only after the elite set reaches

its size. For each solution $Sol$ generated in a GRASP iteration, one of the solutions $e \in E$ is selected and a path-relinking is applied between them. There are two ways to choose element $e$: randomly selection (E1) or picking up the one with the best cost (E2). A path-relinking is performed by starting from an initial solution $s_i$ and gradually incorporating attributes from a guide solution $s_g$ to it, until $s_i$ becomes equal to $s_g$. Several ways to select $s_i$ and $s_g$ have been studied in the literature [9]. We have explored the following strategies:

- Backward relinking (T1): the best cost solution is set to $s_i$ and the worst one to $s_g$.
- Forward relinking (T2): the worst cost solution is set to $s_i$ and the best one to $s_g$.
- Mixed relinking (T3): both trajectories are explored in alternate way. At each iteration of path-relinking, the roles of initial and guide solutions are inverted.

Figure 2 shows the GRASP with path-relinking developed in this work. In line 1, the instance parameters are obtained and also the parameters $EliteSel$ to specify the way to select a solution from the elite set and $RelinkType$ used to define the strategy of path-relinking. The elite set and the solutions repository are initialized in lines 3 and 4. For each GRASP iteration a solution is constructed in line 6 and, in line 7, a search is performed in the repository to verify if this

```
procedure GRASP_PR
01.  Read_Input_Data(n, m, DivMat, MaxIter, EliteSel, RelinkType);
02.  z_best ← −∞;
03.  E ← {};
04.  Repository ← {};
05.  for i = 1, ..., MaxIter do
06.    Sol ← Build_Random_Greedy_Solution(n, m, DivMat);
07.    Search_Sol_Repository(Repository, Sol, SolNotFound);
08.    if SolNotFound then do
09.      Repository ← Repository ∪ {Sol};
10.      Sol ← Local_Search(Sol);
11.    end if;
12.    if E is full then do
13.      SolElite ← Select_Sol_Elite(E, EliteSel);
14.      Sol ← PathRelinking(Sol, SolElite, RelinkType);
15.    end if;
16.    Update_Elite(E, Sol);
17.    if z(Sol) > z_best do
18.      z_best ← z(Sol);
19.      Best_Sol ← Sol;
20.    end if;
21.  end for;
22.  return Best_Sol;
```

**Fig. 2.** Algorithm for GRASP with path-relinking

solution has already been produced. If it has not been found, the repository is updated in line 9, and a local search is executed in line 10. If the elite set is full, the path-relinking should be performed. In line 13, the element of the elite set is selected according to *EliteSel* parameter and, in line 14, the path-relinking is performed using the strategy defined by parameter *RelinkType*. In line 16, either the solution generated by the GRASP iteration or by path-relinking is evaluated to verify if it should be inserted in the elite set. If the elite set is not full and the solution is not already in the elite set, then it is inserted into it. If the elite set is full and the solution cost is better than the worst-cost elite set solution, then this new solution replaces the worst one in the elite set. In lines 18 and 19 the best solution and best cost are updated.

We show in Fig. 3 the details on our implementation of path-relinking for the MDP. In line 1, the solution generated by the GRASP iteration *Sol* and the solution from the elite size *SolElite* are compared to determine which has the best cost ($BS$) and which has the worst one ($WS$). From lines 3 to 9, the initial and guide solutions are set. If either strategy `backward relinking` (T1) or `mixed relinking` (T3) is used, the initial solution is set as the best-cost solution among *Sol* and *SolElite*. Otherwise, if strategy `forward relinking` (T2) is adopted, the initial solution is set as the worst one. From line 10 to 22, the steps of path-relinking are performed until the initial solution reaches

```
procedure Path_Relinking(Sol, SolElite, RelinkType)
01.   Find_Better_Worst(Sol, SolElite, BS, WS);
02.   BetterSolPR ← BS;
03.   if RelinkType equal to T1 or T3 then do
04.     Initial ← BS;
05.     Guide ← WS;
06.   else do
07.     Initial ← WS;
08.     Guide ← BS;
09.   end if;
10.   while Initial not equal to Guide do
11.     DifEl ← Obtain_Different_Elements(Initial, Guide);
12.     IntSol ← Obtain_Int_Sol(Initial, DifEl);
13.     if z(IntSol) > z(BetterSolPR) then do
14.       BetterSolPR ← IntSol;
15.     end if;
16.     if RelinkType is equal to T1 or T2 then do
17.       Initial ← IntSol;
18.     else do
19.       Initial ← Guide;
20.       Guide ← IntSol;
21.     end if;
22.   end while;
23.   return BetterSolPR;
```

**Fig. 3.** Algorithm for path-relinking

the guide solution. In line 11, the elements that are in the guide solution and are not in the initial solution are found. In line 12, a step of the path-relinking is performed. Intermediate solutions are obtained by replacing an element that belongs to the initial solution and do not belong to the guide solution with all other elements that belong to the guide solution and do not belong to the initial solution. The solution which presents the best cost among all these intermediate solutions is selected as the result of an iteration of path-relinking. This solution ($IntSol$) is then more similar to the guide solution because one element from the initial solution was replaced by another one from the guide solution. In line 14, if this new intermediate solution ($IntSol$) has a better cost than the current best intermediate solution ($BetterSolPR$), then the latter is updated. From lines 16 to 21, the solutions used as initial and guide for the next iteration of path-relinking are selected according to the strategy adopted.

We developed six GRASP heuristics combining the two parameters used for path-relinking: the way to select an element from the elite set and the strategy used for walking from one solution to another. The computational experiments implemented to evaluate the performance of these heuristics are presented in the next Section.

## 3 Computational Results

The computational experiments were performed on five sets (A, B, C, D, and E) of test problems with different characteristics [11]. An instance of a set consists of a population $P$ with size $n$ and a diversity matrix $DivMat$, which contains the diversity $d_{ij}$ between elements $i$ and $j$ of $P$. The instances of a specific set are obtained from the same base diversity matrix, differing among themselves by the population size. Different base matrices are used to generate instances for each set. All sets A, B, C, and D contain instances with population sizes $n = 50, 100, 150, 200, 250$. The instances for set E have populations sizes $n = 300, 400, 500$. For all instances, tests were performed to find subsets $M$ of sizes 20% and 40% of the population size.

The six GRASP procedures evaluated were created by combining a walking strategy (T1, T2, and T3) with the way to select an element from the elite set (E1, E2).

The algorithms were implemented in C and compiled with `gcc 2.96`. The tests for sets A, B, C, and D were performed on a 550 MHz Intel Pentium III PC with 384 Mbytes of RAM, and for set E on a 1.3 GHz AMD Athlon with 256 Mbytes of RAM.

In Tables 1 and 2, we show the results of computing 1000 iterations for the pure GRASP and for each GRASP with path-relinking. The first column identifies each instance by the set it belongs to (A, B, C, D, or E) and the size of the population. Bold values indicate that a new GRASP heuristic outperforms the pure GRASP, and the best result found is underlined. The results obtained for selecting a subset of size equal to 20% of the population size are in Table 1 and for 40% can be found in Table 2. The elite set size was 5 and the path-relinking

**Table 1.** Diversity values for subset of size 20% of the population size

| Inst. | GRASP | T1E1 | T2E1 | T3E1 | T1E2 | T2E2 | T3E2 | Best |
|---|---|---|---|---|---|---|---|---|
| A050 | 491.9 | 491.9 | 491.9 | 491.9 | 491.9 | 491.9 | 491.9 | 491.9 |
| A100 | 2007.1 | 2007.1 | 2007.1 | 2007.1 | 2007.1 | 2007.1 | 2007.1 | 2007.1 |
| A150 | 4552.1 | 4552.1 | 4552.1 | 4552.1 | 4552.1 | 4552.1 | 4552.1 | 4552.1 |
| A200 | 8132.1 | 8132.1 | 8132.1 | 8132.1 | 8132.1 | 8132.1 | 8132.1 | 8132.1 |
| A250 | 12653 | 12653 | 12653 | 12653 | 12653 | 12653 | 12653 | 12653 |
| B050 | 334976 | 334976 | 334976 | 334976 | 334976 | 334976 | 334976 | 334976 |
| B100 | 1267277 | 1267277 | 1267277 | 1267277 | 1267277 | 1267277 | 1267277 | 1267277 |
| B150 | 2758381 | 2758381 | 2758381 | 2758381 | 2758381 | 2758381 | 2758381 | 2758381 |
| B200 | 4787819 | **4788086** | **4788086** | **4788086** | **4788086** | **4788086** | **4788086** | 4788086 |
| B250 | 7378534 | **7388307** | **7388471** | **7388501** | **7388471** | **7388471** | **7388501** | 7388997 |
| C050 | 316409 | 316409 | 316409 | 316409 | 316409 | 316409 | 316409 | 316409 |
| C100 | 1205722 | 1205722 | 1205722 | 1205722 | 1205722 | 1205722 | 1205722 | 1205722 |
| C150 | 2613286 | 2613286 | 2613286 | 2613286 | 2613286 | 2613286 | 2613286 | 2613286 |
| C200 | 4627942 | **4630545** | **4630545** | **4630545** | **4630545** | **4630545** | **4630545** | 4630545 |
| C250 | 7177365 | **7178043** | **7178043** | **7178043** | **7178043** | **7178043** | **7178043** | 7178043 |
| D050 | 381379 | 381379 | 381379 | 381379 | 381379 | 381379 | 381379 | 381379 |
| D100 | 1570800 | 1570800 | 1570800 | 1570800 | 1570800 | 1570800 | 1570800 | 1570800 |
| D150 | 3498551 | **3500593** | **3500593** | **3500593** | **3502215** | **3502215** | **3502215** | 3502567 |
| D200 | 6201319 | **6202900** | **6201603** | **6202900** | **6202900** | **6202900** | **6202900** | 6207580 |
| D250 | 9673201 | **9679220** | **9676047** | **9677761** | **9680718** | **9680718** | **9680718** | 9685430 |
| E300 | 9653 | **9689** | **9689** | **9689** | **9684** | **9684** | **9684** | 9689 |
| E400 | 16870 | **16906** | **16906** | **16906** | **16906** | **16906** | **16906** | 16956 |
| E500 | 26158 | **26179** | **26179** | **26179** | **26197** | **26197** | **26197** | 26254 |

**Table 2.** Diversity values for subset of size 40% of the population size

| Inst. | GRASP | T1E1 | T2E1 | T3E1 | T1E2 | T2E2 | T3E2 | Best |
|---|---|---|---|---|---|---|---|---|
| A050 | 1931.5 | 1931.5 | 1931.5 | 1931.5 | 1931.5 | 1931.5 | 1931.5 | 1931.5 |
| A100 | 7730 | 7730 | 7730 | 7730 | 7730 | 7730 | 7730 | 7730 |
| A150 | 17482.4 | 17482.4 | 17482.4 | 17482.4 | 17482.4 | 17482.4 | 17482.4 | 17482.4 |
| A200 | 31048.6 | 31048.6 | 31048.6 | 31048.6 | 31048.6 | 31048.6 | 31048.6 | 31048.6 |
| A250 | 48384.3 | 48384.3 | 48384.3 | 48384.3 | 48384.3 | 48384.3 | 48384.3 | 48384.3 |
| B050 | 1171416 | 1171416 | 1171416 | 1171416 | 1171416 | 1171416 | 1171416 | 1171416 |
| B100 | 4544642 | 4544642 | 4544642 | 4544642 | 4544642 | 4544642 | 4544642 | 4544642 |
| B150 | 9956281 | **9960461** | **9960461** | **9960461** | **9956937** | **9956717** | **9956937** | 9960461 |
| B200 | 17544447 | 17544447 | 17544447 | 17544447 | 17544447 | 17544447 | 17544447 | 17544448 |
| B250 | 27133488 | **27153046** | **27151127** | **27153046** | **27153694** | **27153694** | **27153694** | 27162906 |
| C050 | 1094343 | 1094343 | 1094343 | 1094343 | 1094343 | 1094343 | 1094343 | 1094343 |
| C100 | 4219476 | 4219476 | 4219476 | 4219476 | 4219476 | 4219476 | 4219476 | 4219476 |
| C150 | 9374611 | 9374611 | 9374611 | 9374611 | 9374611 | 9374611 | 9374611 | 9374611 |
| C200 | 16759895 | 16759895 | 16759895 | 16759895 | 16759895 | 16759895 | 16759895 | 16759895 |
| C250 | 26047022 | 26047022 | 26047022 | 26047022 | 26047022 | 26047022 | 26047022 | 26047022 |
| D050 | 1502908 | 1502908 | 1502908 | 1502908 | 1502908 | 1502908 | 1502908 | 1502908 |
| D100 | 6067776 | 6067776 | 6067776 | 6067776 | 6067776 | 6067776 | 6067776 | 6067776 |
| D150 | 13609151 | 13609151 | 13609151 | 13609151 | 13609151 | 13609151 | 13609151 | 13611261 |
| D200 | 24127123 | **24131660** | **24131340** | **24131660** | **24131660** | **24131340** | **24131660** | 24133320 |
| D250 | 37718854 | **37735642** | **37735642** | **37735642** | **37742328** | **37737210** | **37742328** | 37753120 |
| E300 | 35864 | **35874** | **35874** | **35874** | **35874** | **35874** | **35874** | 35881 |
| E400 | 62340 | **62417** | **62417** | **62417** | **62429** | **62429** | **62429** | 62483 |
| E500 | 97171 | **97254** | **97254** | **97254** | **97268** | **97268** | **97268** | 97344 |

strategy starts to be performed only after this set set is full, i.e, during the first five iterations, path-relinking is not applied. The best known values, obtained by several different algorithms [2, 4, 11] under distinct environment conditions, are shown in the last column.

For the 46 tests, the proposed GRASP heuristics found 17 better solutions than the pure GRASP. For instances of population size 50, the optimal solution is known [2] and the pure GRASP and the GRASP with path-relinking heuristics were able to find them. All procedures obtained the same result for population size 100. The new GRASP heuristics show better results than pure GRASP for larger population sizes. The strategies that use path-relinking found better solutions for almost all instances of population size larger or equal to 250.

To evaluate the performance of the path-relinking concerning the strategies used for defining the initial and guide solutions (T1, T2, and T3), we observed the number of times that one of these strategies found the best (underlined) solution. The values obtained for T1, T2, and T3 are 23, 19, and 25, respectively, indicating that strategies T1 and T3 performed better. Other works [9, 10] also show that usually T1 and T3 outperforms T2. The explanation for this behavior is that these two strategies start the trajectory from a good cost solution, and the neighborhood of the initial solution is much more carefully explored than that of the guiding one. So there is a better chance to investigate in more detail the neighborhood of the most promising solution.

As these two strategies presented better results, more tests were performed using them. The instances used have population sizes 150, 200, 250, 300, 400, and 500. We discarded instances from set A and with population size 100, because the new GRASP heuristics obtained the same results of the pure GRASP.

Each instance was run three times with different random seeds. Tables 3 and 4 show the average and best diversity values and the average computational times (in seconds) achieved for subsets of size 20% and 40% of the population size. Bold values indicate that a new GRASP heuristic outperforms the pure GRASP, and the best result found is underlined. Considering the average diversity values, the GRASP with path-relinking procedures found 19 better solutions in 24 tests. Strategies T1 and T3 behave similarly, generating almost the same number of better results. For instances with subset size of 20%, selecting a random element from the elite set (E1) generated better results, while for instances with subset size of 40%, choosing the better solution of the elite set (E2) performed better.

The new GRASP heuristics generate better results than pure GRASP but it demands more computational time, as we can see from these tables. To perform a more fair comparison, the pure GRASP was allowed to execute until achieve a fixed limit on run time. This limit was set to the highest time used by the new GRASP heuristics with path-relinking. The average results obtained for three independent runs are presented in Table 5. The second and sixth columns present the diversity values obtained when executing pure GRASP for 1000 iterations using subset sizes of 20% and 40%, respectively. The third and seventh columns show the values generated by the pure GRASP executing until the run time limit was achieved. In the remaining columns, the worst and best values produced by the GRASP with path-relinking procedures are exhibited. The bold values, in the third and seventh columns, indicate that the pure GRASP with more time outperforms the pure GRASP. But it never gives better results than the worst

**Table 3.** Diversity values and computational times (in seconds) for subset of size 20% of the population size

| | GRASP | | | T1E1 | | | T3E1 | | | T1E2 | | | T3E2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inst. | average | best | Tavg | average | best | Tavg | average | best | Tavg | average | best | Tavg | average | best | Tavg |
| B150 | 2756581 | 2758381 | 57 | **2758381** | 2758381 | 109 | **2758381** | 2758381 | 112 | **2758003** | 2758381 | 111 | **2758003** | 2758381 | 106 |
| B200 | 4787900 | 4788062 | 184 | **4788086** | **4788086** | 287 | **4788086** | **4788086** | 283 | **4788086** | **4788086** | 289 | **4788086** | **4788086** | 286 |
| B250 | 7365771 | 7378534 | 468 | **7376110** | **7388307** | 850 | **7376175** | **7388501** | 847 | **7375944** | **7388471** | 870 | **7375954** | **7388501** | 859 |
| C150 | 2613286 | 2613286 | 52 | 2613286 | 2613286 | 67 | 2613286 | 2613286 | 72 | 2613286 | 2613286 | 73 | 2613286 | 2613286 | 65 |
| C200 | 4626056 | 4627942 | 174 | **4630545** | **4630545** | 211 | **4630545** | **4630545** | 229 | **4628450** | **4630545** | 225 | **4629318** | **4630545** | 210 |
| C250 | 7176140 | 7177365 | 454 | **7178043** | **7178043** | 547 | **7178043** | **7178043** | 572 | **7177973** | **7178043** | 572 | **7177973** | **7178043** | 534 |
| D150 | 3498547 | 3498824 | 53 | **3500283** | **3500593** | 71 | **3500283** | **3500593** | 72 | **3502060** | **3502215** | 71 | **3502060** | **3502215** | 68 |
| D200 | 6203391 | 6205869 | 174 | **6205505** | **6206807** | 202 | **6205505** | **6206807** | 201 | **6204650** | **6206807** | 200 | **6204650** | **6206807** | 201 |
| D250 | 9671279 | 9674389 | 451 | **9680980** | **9682832** | 524 | **9680494** | **9682832** | 514 | **9680945** | **9682832** | 513 | **9680189** | **9682832** | 515 |
| E300 | 9652 | 9653 | 376 | **9679** | **9689** | 472 | **9679** | **9689** | 473 | **9678** | **9684** | 473 | **9678** | **9684** | 478 |
| E400 | 16874 | 16896 | 1437 | **16904** | **16910** | 1754 | **16904** | **16910** | 1752 | **16899** | **16906** | 1765 | **16899** | **16906** | 1767 |
| E500 | 26151 | 26165 | 2875 | **26198** | **26211** | 3269 | **26198** | **26211** | 3263 | **26198** | **26212** | 3249 | **26198** | **26212** | 3252 |

**Table 4.** Diversity values and computational times (in seconds) for subset of size 40% of the population size

| | GRASP | | | T1E1 | | | T3E1 | | | T1E2 | | | T3E2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inst. | average | best | Tavg | average | best | Tavg | average | best | Tavg | average | best | Tavg | average | best | Tavg |
| B150 | 9952386 | 9956281 | 352 | **9957365** | **9960461** | 439 | **9957365** | **9960461** | 445 | 9952386 | **9956937** | 449 | **9952949** | **9956937** | 441 |
| B200 | 17526848 | 17544447 | 1109 | **17543566** | 17544447 | 1234 | **17543566** | 17544447 | 1257 | **17544061** | 17544447 | 1284 | **17544061** | 17544447 | 1228 |
| B250 | 27134803 | 27141894 | 2722 | **27153692** | **27154446** | 3297 | **27153829** | **27154446** | 3306 | **27154524** | **27157209** | 3305 | **27154943** | **27157437** | 3173 |
| C150 | 9374611 | 9374611 | 218 | 9374611 | 9374611 | 232 | 9374611 | 9374611 | 241 | 9374611 | 9374611 | 233 | 9374611 | 9374611 | 231 |
| C200 | 16759895 | 16759895 | 533 | 16759895 | 16759895 | 659 | 16759895 | 16759895 | 666 | 16759895 | 16759895 | 668 | 16759895 | 16759895 | 684 |
| C250 | 26047022 | 26047022 | 1937 | 26047022 | 26047022 | 2063 | 26047022 | 26047022 | 2050 | 26047022 | 26047022 | 2189 | 26047022 | 26047022 | 2258 |
| D150 | 13609448 | 13609596 | 229 | 13609448 | 13609596 | 250 | 13609448 | 13609596 | 258 | 13609448 | 13609596 | 253 | 13609448 | 13609596 | 253 |
| D200 | 24127046 | 24127807 | 874 | **24130476** | **24131660** | 932 | **24130476** | **24131660** | 967 | **24131581** | **24131660** | 949 | **24131581** | **24131660** | 952 |
| D250 | 37720373 | 37722360 | 2322 | **37733021** | **37735642** | 2481 | **37733021** | **37735642** | 2552 | **37735250** | **37742328** | 2664 | **37735250** | **37742328** | 2689 |
| E300 | 35853 | 35864 | 1619 | **35868** | **35874** | 1887 | **35868** | **35874** | 1887 | **35868** | **35874** | 1928 | **35868** | **35874** | 1916 |
| E400 | 62343 | 62354 | 10334 | **62415** | **62417** | 12924 | **62415** | **62417** | 12803 | **62424** | **62429** | 12861 | **62424** | **62429** | 12798 |
| E500 | 97170 | 97171 | 44686 | **97252** | **97254** | 46144 | **97252** | **97254** | 46675 | **97269** | **97269** | 47191 | **97277** | **97292** | 46216 |

**Table 5.** Average diversity obtained by pure GRASP executing more time

| | 20% | | | | 40% | | | |
|---|---|---|---|---|---|---|---|---|
| Inst. | GRASP | GRASPt | Worst PR | Best PR | GRASP | GRASPt | Worst PR | Best PR |
| B150 | 2756581 | 2756581 | 2758003 | 2758381 | 9952386 | 9952386 | 9952386 | 9957365 |
| B200 | 4787900 | **4787908** | 4788086 | 4788086 | 17526848 | 17526848 | 17543566 | 17544061 |
| B250 | 7365771 | **7370551** | 7375944 | 7376175 | 27134803 | **27137262** | 27153692 | 27154943 |
| C150 | 2613286 | 2613286 | 2613286 | 2613286 | 9374611 | 9374611 | 9374611 | 9374611 |
| C200 | 4626056 | **4626057** | 4628450 | 4630545 | 16759895 | 16759895 | 16759895 | 16759895 |
| C250 | 7176140 | **7176141** | 7177937 | 7178043 | 26047022 | 26047022 | 26047022 | 26047022 |
| D150 | 3498547 | **3498642** | 3500283 | 3502060 | 13609448 | 13609448 | 13609448 | 13609448 |
| D200 | 6203391 | 6203391 | 6204650 | 6205505 | 24127046 | 24127046 | 24130476 | 24131581 |
| D250 | 9671279 | 9671279 | 9680189 | 9680980 | 37720373 | 37720373 | 37733021 | 37735250 |
| E300 | 9652 | 9652 | 9678 | 9679 | 35853 | 35853 | 35868 | 35868 |
| E400 | 16874 | **16881** | 16899 | 16904 | 62343 | **62345** | 62415 | 62424 |
| E500 | 26151 | **26154** | 26198 | 26198 | 97170 | 97170 | 97252 | 97277 |

value generated by a GRASP with path-relinking. These results reinforce the contribution of inserting the path-relinking strategy into a pure GRASP.

We made a deeper performance analysis for the new GRASP heuristics T1E1, T1E2, T3E1, and T3E2. We selected three instances B150, C150, and D150 and executed each GRASP heuristic until a solution was found with a greater or equal cost compared to a target value. Two target values were used for each instance: the average and the best value obtained by the pure GRASP. Empirical probability distributions for the time to achieve a target value are plotted in Fig. 4. We only show results for the instance D150 because we have similar behavior for the two other instances. To plot the empirical distribution, we executed each GRASP heuristic 200 times using 200 different random seeds. In each execution, we measured the time to achieve a solution whose cost was greater or equal to the target cost. The execution times were sorted in ascending order so that $t_i$ represents the $i^{th}$ lower time. For $i = 1, \ldots, 200$, a probability $p_i = (i - 0.5)/200$ was associated for each time $t_i$ and the points $z_i = (t_i, p_i)$ were plotted [1].

Figure 4 illustrates that when the target becomes more difficult to be reached, the path-relinking strategy can find the solutions faster than pure GRASP. We can observe that all path-relinking strategies behave similarly, so it is not possible from this analysis to determine which of them performs better.

To show the evolution in quality of the elite set in the path-relinking strategy, T1E1 was run 200 times with 200 different seeds for instances C150 and D150. The stopping criterion for each run was to execute 1000 iterations. For each iteration of each run, we evaluate if the path-relinking improved the solution generated after the local search. In Fig. 5, a point $(x, y)$ of the curve represents the number of times $y$ that the T1E1 strategy found a better solution than pure GRASP in the $x^{th}$ iteration. For example, in the $10^{th}$ iteration for instance C150, we can see that, for the 200 executions, path-relinking was able to improve the solution generated after local search in approximately 170 times. The results show that after the elite set is full (iteration 5) and the path-relinking starts to be used, it contributes to generate better cost solutions. For the final iterations, path-relinking can improve many more solutions, because the elite set was improved with good solutions of earlier iterations.

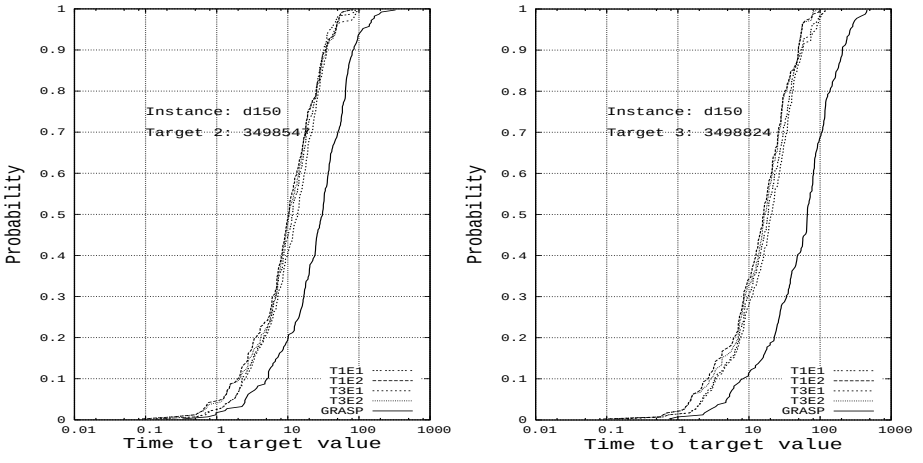



**Fig. 4.** Empirical probability distributions for the time to achieve a target value

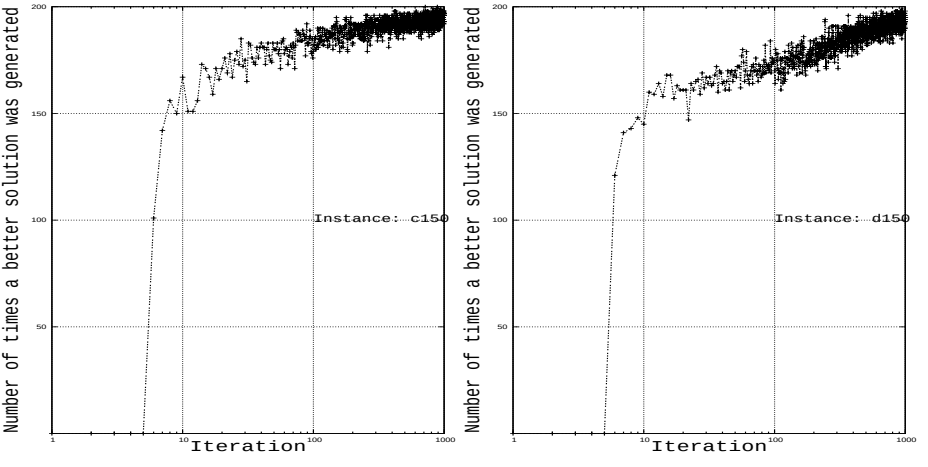



**Fig. 5.** Number of solutions improved by path-relinking

We also executed some tests to evaluate the influence of the elite set size in the performance of the path-relinking strategies. The instances of Tables 3 and 4 were run three times using an elite set size of 10 and 20. Comparing the average diversity values obtained using the elite set size 10 to the previous results, which were obtained using an elite set size 5, we noticed that, in 20 instances, it was able to generate better results, while, in another 20, the results were worse. When using the elite size 20, 23 better and 25 worse results were obtained. These results give evidence that having larger elite set size does not benefit the path-relinking strategy for solving the MDP problem.

## 4 Concluding Remarks

This paper presented some versions of GRASP heuristics to solve the maximum diversity problem (MDP). The main goal of this work was to analyse the influence of inserting path-relinking strategies to a pure GRASP.

Experimental results show that the versions that use path-relinking significantly improve the average quality of solutions generated by a pure GRASP approach proposed in the literature. Our experiments also show that path-relinking speeds up convergence to target values.

Although the new GRASP heuristics presented here were able to improve the results obtained by a pure GRASP, they were not able to find all best known solutions for these instances. As a future work, we believe that inserting the path-relinking technique into pure GRASP presented in [11], which has obtained good results, may contribute to achieve better solutions for the MDP.

## References

1. Aiex, R. M., Resende, M. G. C., Ribeiro, C. C.: Probability distribution of solution time in GRASP: An experimental investigation, J. of Heuristics **8** (2002), 343–373.
2. Andrade, P. M. F., Plastino, A., Ochi, L. S., Martins, S. L.: GRASP for the Maximum Diversity Problem, Procs. of MIC 2003, CD-ROM Paper: MIC03_15, (2003).
3. Feo, T. A., Resende, M. G. C.: Greedy randomized adaptive search procedures, J. of Global Optimization **6** (1995), 109–133.
4. Ghosh, J. B.: Computational aspects of the maximum diversity problem, Operations Research Letters **19** (1996), 175–181.
5. Glover, F., Laguna, M., Marti, R.: Fundamentals of scatter search and path-relinking, Control and Cybernetics **19** (1977), 653–684.
6. Glover, F., Hersh, G., McMillan, C.: Selecting subsets of maximum diversity, MS/IS Report No. 77-9, University of Colorado at Boulder, (1977).
7. Glover, F., Kuo, C-C., Dhir, K. S.: Integer programming and heuristic approaches to the minimum diversity problem, J. of Bus. and Management **4**, (1996), 93–111.
8. Kochenberger, G., Glover, F.: Diversity data mining, Working Paper, The University of Mississipi, (1999).
9. Resende, M. G. C., Ribeiro, C. C.: GRASP with path-relinking: Recent advances and applications, Metaheuristics: Progress as Real Problem Solvers (T. Ibaraki, K. Nonobe, and M. Yagiura, editors), (2005), 29–63.
10. Ribeiro, C. C., Uchoa, E., Werneck, R. F.: A hybrid GRASP with perturbations for the Steiner problem in graphs, INFORMS J. on Computing 14 (2002), 228-246.
11. Silva, G. C., Ochi, L. S., Martins, S. L.: Experimental comparison of greedy randomized adaptive search procedures for the maximum diversity problem, Lecture Notes on Computer Science **3059** (2004), 498–512.
12. Weitz, R., Lakshminarayanan, S.: An empirical comparison of heuristic methods for creating maximally diverse groups, J. of the Op. Res. Soc. **49** (1998), 635–646.

# How to Splay for loglog*N*-Competitiveness

George F. Georgakopoulos

Dept. of Computer Science, University of Crete,
Knossou Av., GR-71409 , Heraklion, Crete, Greece
ggeo@csd.uoc.gr

**Abstract.** We present an extension of the splay technique, the *chain-splay*. Chain-splay trees splay the accessed element to the root exactly as classic splay trees do, but also perform some local 'house-keeping' splay operations below the accessed element. We prove that chain-splay is loglog*N*-competitive to any off-line searching algorithm. This result is the nearest point to dynamic optimality of splay trees reached since 1983.

## 1 Introduction

A fundamental problem in computer science is SEARCHING: how can we organize $N$ elements into a data structure so that the accession of one by one of the elements of a given sequence is performed optimally? For many decades this problem was addressed using any of a large variety of 'balanced' trees [1, 5, 15, 17] through which an optimal cost $O(\log N)$ cost *per operation* is obtained.

Around 1983–85 a radical turn was made by Sleator and Tarjan (see [19], but also [3]) based on new ideas like *self-adjustment*, *amortized cost analysis* and *competitiveness* [22]. Sleator and Tarjan introduced the well known *splay-trees*, proved a very interesting set of properties for them ('static optimality', 'working set' and 'static finger' properties), and conjectured that splay trees are *dynamically optimal*, in other words, competitive with respect to any off-line searching algorithm that uses a binary search tree. (The reader can trace the story of splay trees and several related issues in [2, 4, 6–14, 16, 18–21, 23].)

The dynamic optimality conjecture remains open since 1983—the most significant relative progress so far being the proof of the 'dynamic finger' conjecture by R. Cole *et al*. (announced in 1989, published in [7, 8]). Yet in 2004 E. Demaine *et al*. [9] offered a data structure (the *tango tree*) which is $O(\log\log N)$-competitive to the optimal searching algorithm. The authors of [9] use a variety of trees consisting of smaller red-black trees that they cut-and-link to maintain a convenient, quite relaxed, balance condition. The structure they obtain—although it is self-organizing—it is not splay-like in the following sense: splay-like search trees do not maintain (explicitly) any balanced condition. (This seems to be a necessary condition for optimality. Tango trees, as their inventors already comment in [9], are not optimal. See also [11] where it is shown that even a minimum balance of an otherwise self-adjusting tree, can destroy optimality.)

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 570 – 579, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Thus we have to ask whether splay trees or splay-like trees can be as good as tango trees. In this work we show that the ideas leading to tango trees, also lead to a splay-like technique (*chain-splay*) that does achieve $O(\text{loglog}N)$-competitiveness. Tango trees and chain-splay stand as strong reminders, if any is needed, of the dynamic optimality issue.

## 2 The On-line Searching Problem

Let $U$ be a set of $N$ elements linearly ordered by $\leq$. A sequence of accesses $H$ (or *history*) of length $M$ is a function $H \in [0..M-1] \to U$. At the $i^{\text{th}}$ step, $i = 0, ..., M-1$, we have to access element $\sigma_i = H(i) \in U$. To do so, we maintain a binary search tree $S$, self-adjusted by some algorithm $A$ that is constrained to navigate in $S$ starting at its root, following one edge at a time and transforming $S$ through rotations. Traversing or rotating an edge costs one unit. Let $S_0$ be the initial tree, $S_{i+1}$ be the tree after completing $access(\sigma_i)$, and $c(\sigma_i)$ be the cost if the $i^{\text{th}}$ access. The total cost of performing all $M$ accessed is $C_A(H) \triangleq \sum_{i=0}^{M-1} c(s_i)$. Algorithm $A$ is said to be an *on-line* algorithm iff the way it performs $access(\sigma_i)$ depends only on 'the past', i.e., only on $S_i$ and $\sigma_0, ..., \sigma_i$. Otherwise it is called *off-line*. Obviously, there exists an optimal offline algorithm OPT: just check all possible sequences of trees and select the best. (See [6] for a more clever, yet still highly inefficient, optimum algorithm).

An algorithm $A$ serving $H$ with total cost $C_A(H)$, is said to be *k-competitive* (to the optimal) if $C_A(H) \leq k\, C_{\text{OPT}}(H)$ for all sequences of accesses $H$. We shall present in the next sections a version of splay achieving $k = O(\text{loglog}N)$ competitiveness.

## 3 Splay Templates, Progress Factors and Amortized-Cost Analysis

The splay technique for a binary tree $S$ may be described by the 'template set' [13, 20] given in Fig. 1. Let the internal node $x$ be a descendant of $y$ in $S$. The splay operation $splay[x \to y]$ is defined as follows: put a cursor ➔ on $x$ and relink its path to $y$ applying repeatedly the '*zig-zag*' or '*zig-zig*' templates and finally the '*zig*' template (if needed), until $x$ takes the place of $y$.

To obtain an amortized cost analysis for splay we assign to each node $x$ in $S$ (even external ones) an atomic weight $w(x) \geq 0$, and define its *total* weight $W(x)$ as the sum of the atomic weights of its descendant nodes, itself included.

For an edge $e = (parent(x), x)$ we define its *progress factor* $\varphi(e)$ by $\Phi(e) \triangleq \log\big(W(parent(x))/W(x)\big)$. For a path $x \to y$ we set $\varphi(x \to y) \triangleq \sum_{e \in (x \to y)} \varphi(e)$. The potential $\Phi$ of tree $S$ —w.r.t. to $w(\cdot)$ —is defined as the sum of all progress factors: $\Phi(S) \triangleq \sum_{e \in S} \varphi(e)$.

**Lemma 1 ('progress lemma'):** Let $splay[x \to y]$ be applied to $S$, obtaining tree $S'$. Then, $length(x \to y) - 1 \leq 3\,\varphi(x \to y) + [\Phi(S) - \Phi(S')]$, i.e., the *amortized cost* of $splay[x \to y]$ is $O(\varphi(x \to y)) = O(\log(W(y)/W(x)))$.

**Proof:** As in [13], or by a direct calculation similar to that in [19]. ■

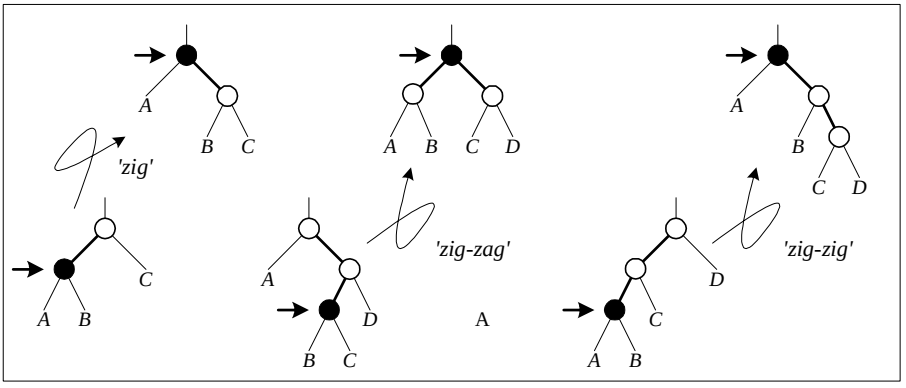



**Fig. 1.** Classic splay templates

## 4 Chain-Partitions and a Lower Bound for $C_{\mathrm{OPT}}(H)$

Simplifying things, let us suppose that $N = |U| = 2^m - 1$ for some integer $m > 0$. Let $B$ be a fully balanced binary tree of $N$ internal and $N+1$ external nodes. For each internal node $x$ in $B$ define *follow*($x$) as either its left child *left*($x$) or its right child *right*($x$), and define the *HEAD* nodes as the set of nodes that are the *follow* of no other one. (We shall see below how to define *follow*(·) w.r.t. a sequence of operations $H$.)

The *chain*($x$) will be the set of nodes reachable from $x$ along edges of the form ($y$, *follow*($y$)). The set of chains $P = \{chain(x): x \in HEAD\}$, is turned to a tree, in fact a binomial one, by defining the parent of a chain with head $x$, as that chain which contains the parent of $x$. Notice that the size of every chain is at most $\log2(N+1)$.

Every sequence of operations $H$ partitions the tree $B$ into chains: for each $x$ in $B$ we set *follow*($x$) ← *left*($x$) if the most recently accessed descendant of $x$ is either $x$, or some node in the left subtree of $x$, or no descendant of $x$ has been accessed so far. Otherwise, we set *follow*($x$) ← *right*($x$). For each step $i$ let $P_i(H)$ be the partition of $B$ into chains induced by the partial history $\sigma_0, ..., \sigma_{i-1}$. For every node $x$ we define its level at step $i$, $level_i(x)$, as the number of *HEAD* nodes of $P_i$ on the path $x \to root(B)$, (i.e., the depth of its chain in the tree of chains.) The following result is due to R. Wilber [23]:

**Lemma 2 ('interleave bound'):** Any algorithm $A$ serving $H$ by accessing a binary search tree through rotations, performs at least $C_{\mathrm{OPT}}(H) \geq 1/2 \sum_{i=0}^{M-1} level_i(\sigma_i) - O(N)$ operations.

(A proof of Lemma 2, rewritten to conform with the framework of this paper, is included in the Appendix for the sake of the reader, since it may provide helpful insights for the results presented subsequently.)

## 5 Chain-Splay: Achieving the Lower Bound Within $O(\text{loglog}N)$

To serve $H$ we have to access at each step $i$ the element $\sigma_i$ and adjust tree $S_i$ to tree $S_{i+1}$. For an amortized cost analysis of these accesses we define the atomic weight $w_i(x)$ of $x$ in $S_i$ by:

$$w_i(x) = \left(\log 2(N+1)\right)^{-level_i(x)} \tag{1}$$

its $level_i$ being derived from its corresponding node in $B$ and the current partition $P_i$.

**Lemma 3:** Let tree $S$ with root $r$ be weighed according to $w_i(\cdot)$. Then the amortized cost for accessing any $x$ in $S$ is $O(\varphi(x \to r)) = O(\text{loglog2}(N+1)) \cdot level(x)$.

**Proof:** Let us denote by $K_{m,l}$ the number of nodes of level $l$ in a fully balanced binary tree of height $m$. Suppose w.l.o.g. that $follow(root(B))$ is $left(root(B))$. A node of level $l$ in the left subtree of the root retains its level in the full tree while an element of level $l$–1 in the right subtree acquires level $l$ in the full tree. Thus $K_{m,l} = K_{m-1,l} + K_{m-1,l-1}$, and since $K_{m,1} = m$ and $K_{m,m} = 1$, we obtain $K_{m,l} = \binom{m}{l}$. Setting $m = \log 2(N+1)$, and calculating the total weight of the root $r$ by,

$$W(r) = \sum_{x \in S} m^{-level(x)} = \sum_{l=1}^{m} K_{m,l} \left(\frac{1}{m}\right)^l = \sum_{l=1}^{m} \binom{m}{l} \left(\frac{1}{m}\right)^l = \left(1+\frac{1}{m}\right)^m - 1 \le e - 1, \tag{2}$$

we obtain:

$$\varphi(x \to r) = \log \frac{W(r)}{W(x)} \le \log \frac{e}{w(x)} \le 2 + \log(m^{level(x)}) = O\left(\log m\right) \cdot level(x). \tag{3}$$

■

By Lemma 1 and Eq. 3, classic splay would suffice for our purposes if the weights were constant. Since each access modifies the partition of $B$ and the weights of nodes, we must *reweigh* the tree $S$. To do so, at a (very) low potential cost, we shall keep the following condition invariant:

**Chain-Respect Condition:**

1. For each step $i = 0, ..., M-1$, every chain $C$ in the partition $P_i$ induced on $B$ by the history up to step $i$, forms a single subtree in $S_i$ (i.e. its nodes are connected by $S_i$), and the parent node in $S_i$ of the top (highest) node of $C$ in $S_i$ belongs to the parent-chain of $C$ w.r.t. to $P_i$.
2. In every node $x$ in $S$ we keep one variable bit denoting whether $x$ is the top node of the chain to which it belongs. It is very easy to maintain this variable bit during rotations. We also keep the depth, $d(x)$, *in* $B$ of the corresponding node in $B$. For this we need just $\text{loglog2}(N+1)$ *fixed* bits. (Thus chain-splay trees use a few variable bits less than tango trees [9].)

**Theorem 4:** The following algorithm—explained in the proof—maintains the chain-respect condition and reweighs $S_i$ without increasing the potential by more than $O(\text{loglog2}(N+1))$ $level_i(\sigma_i)$.

```
Chain-Splay( tree S, element σ):
{ Splay σ to next higher 'top' node - unless σ is already one
  while σ ≠ root(S) do
  { Splay σ to the next higher 'top' node
    x_L = left(σ), x_R = right(σ)
    If d(x_L)<d(x_R) then LeftGroup(x_L) else RightGroup(x_R)
    // ReWeigh S - now! } }

Procedure LeftGroup(x):    // RightGroup(x) is symmetrical
{ c_J ← NIL; z ← left(x)
  while (z ≠ NIL) and (z not a 'top' node) do
  { c_J ← z; if d(z)<d(x) then z ← right(z) else z ← left(z)}
  Let p_J be the predecessor of c_J along the traversed path
  Splay c_J to below p_J , Splay p_J to below x
  Mark right(p_J) as 'top' }
```

**Proof:** Since after step $i$ element $\sigma_i$ is the most recently accessed one, the chain of $\sigma_i$ will become the new maximum chain of $P_{i+1}$, and the partition $P_i$ should be updated accordingly. We perform this update from child-chain to parent-chain as $\sigma_i$ ascends to the root.

Let $C$ be the chain $\sigma_i$ currently belongs to, and let the parent-chain of $C$ be $p(C)$. All elements in $C$ and all the descendants chains of $C$ belong to an interval defined by two elements $x_L$ and $x_R$ which are $\leq$-consecutive *in* $p(C)$. After splaying $\sigma_i$ to the top-node in $S_i$ of $p(C)$ these two elements will appear as $\sigma_i$'s left and right children, and the subtree of $C$ (with all its descendants) will split into two subtrees $L$ and $R$, appearing as right and left subtrees of $x_L$ and $x_R$ (Fig. 2). Chains that are descendants of $C$ will be moved *en block* into either tree $L$ or tree $R$.

We consider two cases (the second being symmetrical to the first):
*Case* 1: If the depth $d(x_R)$—in $B$—of $x_R$ is less than $d(x_L)$ then $x_R$ appears in $B$ above $x_L$ so $\sigma_i$ is a right descendant of $x_R$, and the chain $p(C)$ proceeds in $B$ through the left edge of $x_L$ : $follow(x_L) = left(x_L)$. To prepare the next partition we must set $follow(x_L) \leftarrow right(x_L)$. To update, so far, $P_i$, we should turn $chain(left(x_L))$ into a separate subtree and modify $p(C)$ by incorporating $L$ and $R$ into it.

The nodes in $chain(left(x_L))$ are exactly the nodes in the left subtree of $x_L$ that belong to $p(C)$ and have greater depth in $B$ than $x_L$. These nodes form a $\leq$-interval $J$ *within* chain $p(C)$, hence they can be grouped into one subtree by the procedure `LeftGroup`: starting with the left child of $x_L$, if a node $z$ has depth greater than the depth of $x_L$ then $[z \,..\, x_L] \subseteq J$ thus $z$'s right subtree is included in $J$ and we may proceed to $left(z)$. Otherwise $z$ and its left subtree are disjoint from $J$ and we can proceed to $right(z)$. We stop at NIL nodes or top-nodes (demarking the subtree of $p(C)$). Splaying $c_J$ up to below its predecessor $p_J$ in $p(C)$, and this predecessor just below $x_L$, groups $J$

to a single subtree, restoring our invariant condition: the update $follow(x_L) \leftarrow right(x_L)$ can be done by marking the highest node of $J$ as 'top'. (We splay nodes $p_J$ and $c_J$ so that their access-cost is paid in an amortized sense.)

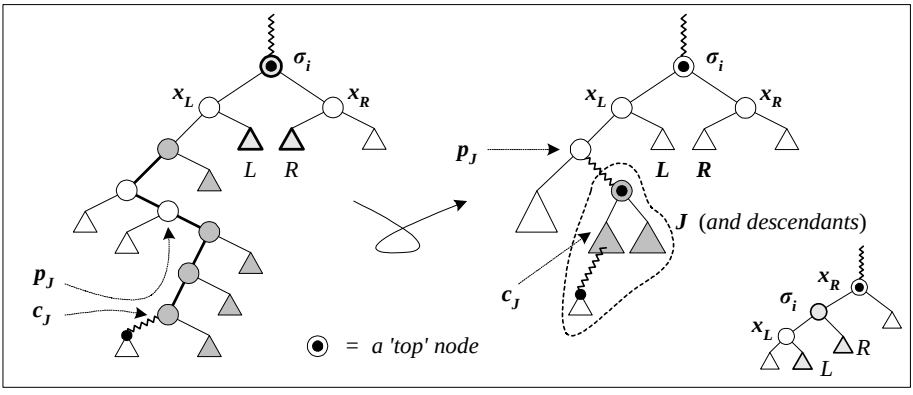



**Fig. 2.** Chain splay – the `LeftGroup` operation

The crucial fact is that after this suitable grouping we have the opportunity to reweigh $S_i$ in the appropriate manner with a minimal cost. Consider the sets, $J^*$ (consisting of nodes in $J$ and all their descendants) and $C^*$ (consisting of nodes in $C$ and all their descendants). The level of nodes in $J^*$ is increased by one so their weights must be divided by $\log2(N+1)$. The level of elements in $C^*$ is decreased by one, hence their weights must be multiplied by $\log2(N+1)$. This is all the reweighing needed, so far (since further updates are dangling). Two observations are in order:

1. The distributions of node-levels *inside* $C^*$ and $J^*$ are identical: these sets are just the subtrees of $x_L$ in $B$. Therefore their total weights *in B* differ just by a factor of $\log2(N+1)$. The weights in $C^*$ are multiplied by, and weights in $J^*$ are divided by, $\log2(N+1)$, hence the total weight of $\sigma_i$ and all its ancestors remains *exactly* the same. In particular $\varphi(\sigma_i \rightarrow root(S_i))$ remains the same.
2. The weights of all the nodes (even external ones) in subtrees $J^*$, $L$, $R$ are multiplied or divided by the same factor, therefore the progress factors $\varphi(e)$ of edges $e$ inside $J^*$, $L$ or $R$ are not modified. (Exactly this effect is more difficult to handle with a 'sum-of-logs' potential on $S$.)

Hence we have to consider only an $O(1)$ number of edges linking $J^*$, $L$, $R$ up to $\sigma_i$, the progress factors of which are easily seen to be either decreased, or increased by less than $O(\log\log2(N+1))$. Since the progress factors of all other edges remain the same, reweighing of $S_i$ increases the potential by an $O(\log\log2(N+1))$ amount at most.
*Case* 2: If $d(x_R)$ is greater than $d(x_L)$ we proceed symmetrically with a right-grouping.

The operations `LeftGroup` and `RighGroup` are performed locally, within the subtree that the relevant chain forms in $S_i$, and their amortized cost is easily calculated

(analogously to Lemma 3) to $O(\text{loglog2}(N+1))$. The while-loop is repeated at most $level_i(\sigma_i)$ times: tree $S_i$ respects the chains of $P_i$ thus we meet at most $level_i(\sigma_i)$ chains along the path $\sigma_i \rightarrow root(S_i)$. ■

Notice that splaying of $\sigma_i$ up to the top node of $p(C)$ may be performed *avoiding* the 'zig' rule of splay. In this way $\sigma_i$ may not become the top node of $p(C)$, but only its left or right child (Fig. 2). In this case our analysis holds as well, with only trivial modifications. Thus the accessed element can be splayed to the root of $S_i$ in exactly the same way as classic splay would do.

**Theorem 5:** `Chain-Splay` is $O(\text{loglog2}(N+1))$-competitive to the optimum searching algorithm, for $M \geq N$.

**Proof:** The splay of $\sigma_i$ to the root has by Lemma 3 an amortized cost of $O(\text{loglog2}(N+1))\cdot level_i(\sigma_i)$ and reweighing has by Theorem 4 an amortized cost of $O(\text{loglog2}(N+1))\cdot level_i(\sigma_i)$. Using Lemma 2 we obtain:

$$C_{\text{C-SPLAY}}(H) \leq O\left(\log m\right) \cdot \sum_{i=0}^{M-1} level_i(\sigma_i) \leq O(\text{loglog2}(N+1))\, C_{\text{OPT}}(H) + \Phi(S_0). \quad \blacksquare$$

## 6 Epilogue: A Comparison with Tango Trees and Some Comments

'Tango' trees [9] are a clever and very efficient variety of search trees. Yet, as far as the dynamic-optimality conjecture is concerned, they present few theoretically undesirable properties: (*a*) If the lower bound of Lemma 2 is tight then they are not optimal, as pointed out in [9]; (*b*) They are not splay-like since they maintain various subtrees explicitly height-balanced; (*c*) They do not use only comparisons—if this is of any theoretical significance; (*d*) They are naturally described through cutting and linking of subtrees, while the optimal opponent is allowed to perform only rotations.

Chain-splay avoids all the above, except, so far, the 1st one: possibly chain-splay is not optimal. Notice however in chain-splay each accessed element is splayed to the root in exactly the same way as ordinary splay would do. The extra work is done to keep the chains *S*-connected. At the time of writing we could not indicate a reason forcing us to do so, other than reweighing *S* effectively. Conceivably extra splaying arises as an artifact of our analysis. Asserting this would be a very strong statement, which—to our opinion—could even be equivalent to the dynamic optimality conjecture. Obviously, O(loglog*N*)-competitiveness of ordinary splay is the major next issue left open by this work.

Finally, we would like to draw the reader's attention to [11] where it is proved that splay trees are competitive to *parametrically balanced* self-adjusting trees—a class including the self-adjusting versions of almost all balanced trees designed so far. Can we apply or extend [11] to prove that chain-splay trees, or better splay trees, are competitive to tangos? Is ordinary splay competitive to chain-splay?

## References


[1] G.M. Adel'son-Vel'ski, E.M. Landis (1962), *"An Algorithm for the Organization of Information"*, Soviet Mathematical Doklady 146, 1259–1263.

[2] S. Albers, J. Westbrook (1998), *"Self-Organizing Data-Structures"* in *"Online Algorithms- The State of the Art"*, Lecture Notes in Computer Science 1442, Springer-Verlag, Chapter 2.

[3] B. Allen, I.J. Munro (1978), *"Self Organizing Binary Search Trees"*, J. of the ACM 25(4), 526–535.

[4] R. Balasubramanian, V. Raman (1995), *"Path Balance Heuristic For Self-Adjusting Binary Search Trees"*, Foundations of Software Technology and Theoretical Computer Science, 15th, Bangalore, India, Dec. 1995, LNCS 1026, Springer-Verlag, 338–348.

[5] R. Bayer, E. McCreight (1972), *"Organization and Maintenance of Large Ordered Indexes"*, Acta Informatica 1(-), 173–189.

[6] A. Blum, S. Chawla, A. Kalai (2002), *"Static Optimality and Dynamic Search-Optimality in Lists and Trees"*, ACM-SIAM Symp. on Discrete Algorithms, 13th, San Francisco, USA, Jan. 6–8, 1–8.

[7] R.J. Cole (2000) *"On the Dynamic Finger Conjecture for Splay Trees, Part II: The Proof"*, SIAM J. on Computing 30(1), 44–85.

[8] R.J. Cole, B. Mishra, J. P. Schmidt, A. Siegel (2000), *"On the Dynamic Finger Conjecture for Splay Trees, Part I: Splaying Sorting in log n-Block Sequences"*, SIAM J. on Computing 30(1), 1–43.

[9] D.E. Demaine, D. Harmon, J. Iacono, M. Pătraşcu (2004), *"Dynamic Optimality—Almost"*, IEEE Symp. on the Foundations of Computer Science, 45th, Rome, Italy, Oct. 17–19, 484–490.

[10] M. Fürer (1999), *"Randomized Splay Trees"*, ACM-SIAM Symp. on Discrete Algorithms, 10th, Maryland, USA, Jan. 17–19, 903–904.

[11] G.F. Georgakopoulos (2004), *"Splay Trees: a Reweighing Lemma and a Proof of Competitiveness vs. Dynamic Balanced Trees"*, J. of Algorithms 51, 64–76.

[12] G.F. Georgakopoulos, D.J. McClurkin (2001), *"Sphendamnoe: A Proof that k-Splay Falls to Achieve* $\log_k N$ *Behaviour"*, 8th Panhellenic Conf. on Informatics, Nicosia, Cyprus, Nov. 2001, Lecture Notes in Computer Science 2563, Springer-Verlag , 480–496.

[13] G.F. Georgakopoulos, D.J. McClurkin, (2004), *"General Template Splay: A Basic Theory and Calculus"*, The Computer Journal 41(1), 10–19.

[14] J. Iacono (2002), *"Key Independent Optimality"*, Inter. Symp. on Algorithms and Complexity, 13th, Lecture Notes in Computer Science 2518, Springer-Verlag, 25–31.

[15] D.E. Knuth D.E. (1971), *"Optimum Binary Search Trees"*, Acta Informatica 1, 14–25.

[16] C. Martel (1991), *"Self-Adjusting Multi-Way Search Trees"*, Information Processing Letters 38(3), 135–141.

[17] M.H. Overmars (1983), *"The Design of Dynamic Data Structures"*, Lecture Notes in Computer Science 156, Springer-Verlag, Heidelberg, 1983.

[18] M. Sherk M. (1995), *"Self Adjusting k-ary Search Trees"*, J. of Algorithms 19, 25–44.

[19] D.D. Sleator, R.E. Tarjan (1985), *"Self Adjusting Binary Search Trees"*, J. of the ACM 32(3), 652–686.

[20] A. Subramanian (1996), "An Explanation of Splaying", J. of Algorithms 20, 512–525.

[21] R. Sundar (1989), *"Twists, Turns, Cascades, Dequeue Conjecture and Scanning Theorem"*, IEEE Symp. on the Foundations of Computer Science, 30th, North Carolina, USA, Oct. 30 – Nov. 1, 555–559.

[22] R.E. Tarjan (1985), *"Amortized Computational Complexity"*, J. of Applied and Discrete Mathematics 6(2), 306–318.
[23] R. Wilber (1989), *"Lower Bounds for Accessing Binary Search Trees with Rotations"*, SIAM J. on Computing 18(1), 56–67.

## Appendix

**Lemma 2:** Any algorithm $A$ serving $H$ by accessing a binary search tree through rotations, performs at least $C_{\mathrm{OPT}}(H) \geq {}^{1}\!/_{2} \sum_{i=0}^{M-1} level_i(\sigma_i) - O(N)$ operations [23].

**Proof:** If $y$ is a *HEAD* node on the path $\sigma_i \rightarrow root(B)$ then during the access of $\sigma_i$ the $follow(\cdot)$ node of $y$'s parent changes from the sibling of $y$ to $y$ itself. The sum $\sum_{i=0}^{M-1} level_i(\sigma_i)$ equals the total number of such changes (including $M$ accesses to the root) and obviously half of them (excluding at most $N$) change the $follow(\cdot)$ of a node from its *left* to its *right* child. We shall put pebbles on tree $S$ and force algorithm $A$ to gather as many pebbles as the total number of left-to-right changes of $follow(\cdot)$.

Let us say that a node $z$ *covers* a set $J \subseteq U$ if $z$ is an ancestor in $S$ of every element in $J$. Denote the highest node of $J$ by $top(J)$. It is easy to see that, since $S$ is a binary search tree, if $J$ is an interval of $U$ then $top(J)$ covers $J$. For every $x$ in $B$ we define the *left subinterval* $\mathrm{LI}(x)$ of $x$ as $\{x\}$ plus all elements in the left subtree of $x$, and the *right subinterval* $\mathrm{RI}(x)$ as all elements in the right subtree of $x$. Since the subintervals of $x$ are not separated by any element, nodes $top(\mathrm{LI}(x))$ and $top(\mathrm{RI}(x))$ lie on a path in $S$. We define the *transition point* of $x$, $tp(x)$, as the *lowest* of them. Since the subintervals of every $x$ are disjoint and the transition point of $x$ belongs to the subinterval it covers, it covers that subinterval of $x$ to which it belongs. Moreover being the lowest of $top(\mathrm{LI}(x))$ and $top(\mathrm{RI}(x))$ it does *not* cover both. Notice that $A$ has to 'touch' (i.e. visit and/or rotate) $tp(x)$ in order to access in $S$ some element in the subinterval covered by $tp(x)$. Below we exploit this fact.

By the definition of $follow(x)$ in $B$, a left-to-right change occurring at node $x$, corresponds to two steps $a, b \in [0..M-1]$ for which $\sigma_\alpha \in \mathrm{LI}(x)$, $\sigma_b \in \mathrm{RI}(x)$ and for each $k \in (a, b)$ we have $\sigma_k \notin \mathrm{I}(x)$, i.e., the most recent access before step $b$ of an element in $\mathrm{I}(x)$ occurred at step $a$. (Recall that $x$ belongs to $\mathrm{LI}(x)$). After $access(\sigma_a)$ we put a pebble on the transition point $tp_a(x)$ in $S$. When algorithm $A$ 'touches' a node in $S$ it gathers all pebbles on it. The following facts prove our Lemma:

1. Every left-to-right change of $follow(\cdot)$ puts exactly one pebble on $S$. (It puts one on $tp_a(x)$ when it occurs and cannot not occur again unless this one has been gathered.)
2. The transition point of $x$ remains the same during steps $k \in [a..b]$, unless it is touched by $A$. (Unless $tp_a(x)$ is touched no other node can become, or cease to be, a descendant of it. Thus $tp_a(x)$ continues to cover the subinterval of $x$ to which it belongs and not to cover the other one.)
3. Algorithm $A$ touches the transition point $tp_a(x)$ at some step in $(a..b]$. (For accessing $\sigma_b$ algorithm $A$ must touch $tp_b(x)$, and this equals $tp_a(x)$ if the latter has been untouched so far.)
4. No two nodes in $B$ have the same transition point in $S$. (Let $x \neq y$ be two nodes with $z$ as their common transition point. Every node is an ancestor in $B$ of its transition

point, thus $x$ and $y$ are ancestors in $B$ of $z$. W.l.o.g. suppose that $x$, $y$, $z$ appear on their path in the given order. Node $z$ being the transition point of $x$ covers that subinterval of $x$ to which it belongs, i.e. the same subinterval which contains $y$. Thus it covers both subintervals of $y$. Yet $z$ is also the transition point of $y$, while the transition point of no node covers both its subintervals—a contradiction.)

5. Algorithm $A$ has to gather at most one pebble for touching $tp_a(x)$, i.e., it is not overcharged. (Node $tp_a(x)$, up to be touched by $A$, remains the transition node of $x$, hence by 4. no other node will put a pebble on it: every node carries at most on pebble.)
6. At most $N$ pebbles may be left ungathered at the end. ■

# Distilling Router Data Analysis for Faster and Simpler Dynamic IP Lookup Algorithms

Filippo Geraci[1] and Roberto Grossi[2]

[1] IIT-CNR, Pisa, and Dipartimento di Ingegneria dell'Informazione, U. Siena, Italy
filippo.geraci@iit.cnr.it
[2] Dipartimento di Informatica, Università di Pisa, 56127 Pisa, Italy
grossi@di.unipi.it

**Abstract.** We consider the problem of fast IP address lookup in the forwarding engines of Internet routers. We analyze over 2400 public snapshots of routing tables collected over five years, discovering what we call the *middle-class effect*. We exploit this effect for tailoring a simple solution to the IP lookup scheme, taking advantage of the skewed distribution of Internet addresses in routing tables. Our algorithmic solution is easy to implement as it is tantamount to performing an indirect memory access. Its performance can be bounded tightly and has very low memory dependence (e.g. just one memory access to off-chip memory in the hardware implementation). It can quickly handle route announcements and withdrawals on the fly, with a small cost which scales well with the number of routes. Concurrent access is permitted during these updates.

## 1 Introduction

The IP lookup problem is a recurrent problem in the literature for packet forwarding in the Internet [16]. Routers have to forward lots of packets from input interfaces to output interfaces (*next hops*) based on packet's destination Internet address, called an *IP address*. Forwarding a packet requires an IP address *lookup* at the routing table to select the next hop corresponding to the packet. (We will use the term "routing table" to denote what is more properly called a "forwarding table.") As routers have to deal with links whose speed constantly improves, the address lookup is considered one of the major bottlenecks [6, 16]. For networks with a link speed of 10 gigabits per second (OC-192), they need to forward up to 33 million packets per second, assuming that each packet is 40 bytes long. Other bottlenecks, such as those involved by fair queuing policy and IP switching technology, are well understood and handled [11].

In IPv4 [13] the prefixes are binary strings of variable length using the syntax $X.Y.W.Z/L$ to represent the first $L$ bits of the 4-byte word $X.Y.W.Z$, where $8 \leq L \leq 32$. Prefixes can be up to 128 bits in IPv6 [5] (but then have a different syntax). The use of prefixes increases the complexity of the IP address lookup problem. For each packet, more than one prefix in the routing table can match the packet's IP address. In this case, the adopted rule is to take the *longest*

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 580–592, 2005.
© Springer-Verlag Berlin Heidelberg 2005

**Table 1.** A routing table

| prefix | hop |
|---|---|
| 65.10.10.0/24 | 1 |
| 192.168.0.0/17 | 2 |
| 192.168.0.0/18 | 3 |
| 192.168.64.0/18 | 2 |
| 192.168.0.0/32 | 4 |
| 192.168.0.0/29 | 5 |

**Table 2.** Its two-layer organization

| layer 1 | | layer 2 | |
|---|---|---|---|
| 65.10.10.0/24 | 1 | 192.168.0.0/24 | 3 |
| 192.168.0.0/17 | 2 | 192.168.0.0/32 | 4 |
| 192.168.0.0/18 | 3 | 192.168.0.0/29 | 5 |
| 192.168.64.0/18 | 2 | | |
| 192.168.0.0/24 | 255 | | |

*matching prefix*. Given prefixes $p_1, p_2, \ldots, p_n$, for any binary string $x$ we want to identify the longest $p_i$ that equals the first bits of $x$, where $1 \leq i \leq n$. For example, let us consider the prefixes in Table 1. Both prefixes 192.168.0.0/17 and 192.168.0.0/18 match the IP address 192.168.32.125; hence, the packet is forwarded to next hop 3. We will only consider situations arising with single hops, since dealing with multi-hops is very similar. No-route-to-host is the special next hop 0 associated with the empty prefix $\epsilon$.

In this paper we stress the importance of data analysis on real routing tables *before* designing IP lookup algorithms. We begin with the experimental analysis performed on public databases of nearly 2400 snapshots of routing tables collected over five years. We identify some new parameters characterizing the (skewed) distribution of prefixes in routing tables. Based upon our findings, we provide a new and simple solution to the IP address lookup problem.

Our starting point is the result based on full expansion and compression of routing tables by Crescenzi, Dardini and Grossi [4]. (It was later referred to as CDG in [3].) Let us illustrate CDG conceptually with Table 1 for IPv4, considering all possible $2^{32}$ IP addresses that can be queried. For each such address, we associate with it the corresponding next hop according to its longest prefix match in Table 1. Now, let us organize the $2^{32}$ next hops thus computed in a $2^{16} \times 2^{16}$ matrix. Lookup time is now a direct access to this matrix; however, its size does not fit current capacity of L2 caches. Observing that many rows and columns contain repeated values, CDG considers only the distinct rows (as individual sequences of $2^{16}$ next hops each); they are further compressed using run length encoding (RLE) on their values. The lookup requires three accesses but the size reduces to very few megabytes.

**Table 3.** Dataset description

| router | #tables | from | to |
|---|---|---|---|
| aads | 538 | 10-01-00 | 05-15-02 |
| mae-east | 230 | 10-01-00 | 06-01-01 |
| mae-west | 618 | 10-01-99 | 04-12-02 |
| paix | 78 | 10-01-01 | 03-10-02 |
| pacbell | 576 | 12-09-98 | 05-15-02 |
| ripe-ncc | 365 | 01-01-03 | 12-01-03 |
| ripe-ncc | 19 | 10-10-99 | 04-01-04 |

| router | date | router | date |
|---|---|---|---|
| aads | 05-30-01 | oregon-03 | 07-10-03 |
| att | 07-10-03 | pacbell | 05-30-03 |
| east.attcanada | 07-10-03 | paix | 05-30-01 |
| funet | 10-30-97 | telstra | 03-31-01 |
| mae-west | 05-30-01 | telus | 07-10-03 |
| west.attcanada | 07-10-03 | oregon-01 | 03-31-01 |

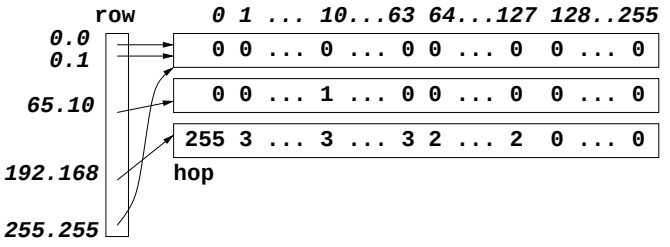



**Fig. 1.** The arrays `row` and `hop` for layer 1 in Table 2

To our knowledge CDG is the first to describe a lookup scheme whose design is fully driven by data analysis. A frequently cited survey [16] published in 2001 shows that CDG is almost an order of magnitude faster than its state-of-the-art competitors at that time (see Table 3 in [16]). Even in the worst case, the frequency of lookups with small response time is impressively high and does not depend on the traffic through the router (see Fig. 22 in [16]). Unfortunately, CDG has some drawbacks. The survey reports that "Schemes using multibit tries and compression give very fast search times. However compression and the leaf pushing technique used do not allow incremental updates. Rebuilding the whole structure is the only solution." Moreover, some authors [3, 7, 15] pointed out some cases in which the space requirement of CDG is too high, possibly causing its performance to suffer in the worst case.

**Our scheme.** We present a lookup scheme that exploits the original idea of CDG in a novel and even simpler way. Going on in our illustrative Table 1, let us truncate the prefixes in the table that are longer than 24, thus retaining just 24 bits and associating with them a dummy next hop (e.g. 255). We obtain *layer 1*, as shown in the left column of Table 2. The prefixes longer than 24 constitute *layer 2*, in the right column of Table 2, which is scarcely populated according to our data analysis. (Note that `192.168.0.0/24` in layer 2 is pushed from layer 1 to deal with IP addresses matching `192.168.0.0/`$L$, where $24 \leq L < 29$.) We can now revisit the approach described for CDG and apply it to layer 1. With each 24-bit address, we associate its corresponding next hop according to its longest prefix match in layer 1. Organizing the resulting next hops into a $2^{16} \times 2^8$ matrix, we keep only the distinct rows (and do not compress them with RLE) as shown in Fig. 1. It suffices to perform a lookup in two accesses in layer 1 by looking at the first 24 bits in the given IP address. For example, the next hop for IP address 192.168.32.125 can be retrieved by following the pointer in `row`[192.168] to a row of 256 entries, in which entry 32 contains the result, next hop 3. Sometimes we get the dummy next hop in layer 1 and so we also need to perform the lookup in layer 2 (this happens very rarely according to our data analysis). Access time and space occupancy are definitively improved over CDG in this way.

Our method exploits some properties that allow us to avoid the drawbacks of CDG. The main discovery is what we call the *middle-class effect* in real routing tables: even though the majority of prefixes have lengths ranging from 16 to 24, they tend to follow regular patterns. In other words, we have a good chance

to store the mapping from all the $2^{32}$ IP addresses to the next hops into a compact table, so that lookup and update are able to access the table very quickly using indirection. Our contributions can be summarized as follows. First, we save space significantly over CDG since we have a much more stable space occupancy that scales linearly with the table size (Fig. 3, left). We no longer need the run-length encoding (RLE) adopted in CDG, because we organize suitably the prefixes. Second, we improve lookup time by nearly 30% (Fig. 3, right). Third, we can dynamize the table, performing updates quickly without rebuilding the whole structure as previously required. Our update algorithm is robust since we can efficiently bound the worst case, which is important for unauthenticated announcements [9]. Concurrent access is also permitted while updating. Our method compares favorably with previous work [16]. We plan to extend our experimental investigation to the interesting method recently proposed in [3].

## 2 Data Analysis of Routing Tables

In this section, we describe our data analysis on routing tables to highlight a useful property for prefixes of length from 16 to 24, called the *middle-class effect*. We first describe the large data set of public routing tables for IPv4 in Section 2.1. We illustrate the middle-class effect in Section 2.2, showing how to exploit it for a two-layer organization of IP lookup tables in Section 2.3.

### 2.1 Databases and Experimental Platforms

We base our analysis on an extensive data set of more than 2400 snapshots of routing tables available from public databases, collected over a period ranging from 1998 to 2004. The major source is the IPMA project [10]. We also collected all daily data for year 2003, plus some monthly snapshots, from the RIPE NCC [14]. Some authors singled out individual snapshots that cause the worst-case behavior of CDG in terms of space occupancy; hence, they are good benchmarks for our method as well. Most of these tables have been employed in the experiments [3, 12, 15]. We report the figures in Table 3.

As for the updates, we collected *all* the announcements and withdrawals available for the entire year 2003 on RIPE NCC. In Fig. 2(left), we plot their number in millions on a daily basis. As we can see, the number of withdrawals is an order of magnitude smaller than the number of announcements. On the average, there is approximately one announcement per second; clearly, they arrive in bursts. For example, note the peak of more than 20 million updates on Oct. 25–26, 2003 (around 300 on the x-axis). We will use this particular peak for intense benchmarking in Section 4. As for the lookups, we could not find publicly available traffic traces (for privacy reasons). We instead use random data from previous work [3], as well as synthetic data. We obtain the latter by extending the approach in [2], adopted in the network community, to generate traffic data according to the distribution of the prefixes of any given routing table $T$ (details given in the technical report [8]).

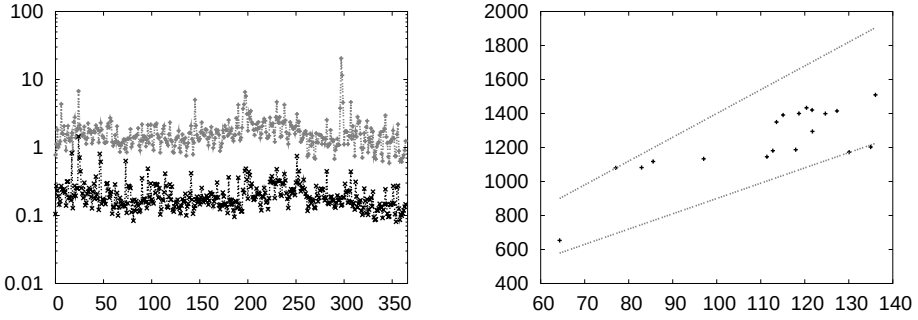



**Fig. 2.** *Left picture:* Millions of daily announcements (top) and of daily withdrawals (bottom) for RIPE NCC, in logarithmic scale on the y-axis. The x-axis reports the 365 days in year 2003 as numbers in the interval $0 \ldots 364$. *Right picture:* Space occupancy of our scheme scales linearly with table size. The x-axis reports the thousands of prefixes and the y-axis the number of kilobytes taken

For our experiments we employed an AMD Athlon XP 1900+ (1.6GHz), 256Mb RAM DDR at 133Mhz, 256Kb L2 cache, 128Kb L1 cache (64 Kb data and 64Kb instructions) running Linux kernel 2.4.22. We used `gettimeofday` for timings. We plan to extend the experimentation to more platforms (e.g. those based on the PowerPC).

### 2.2 Distilling the Middle-Class Effect in Routing Tables

In order to illustrate our ideas, let us consider any routing table $T$; in our case, the snapshot of the RIPE NCC router taken on April 1st, 2004, containing 138201 prefixes. (Note that analogous properties hold for the other router snapshots mentioned in Section 2.1.) What is widely known is the skewed distribution of prefixes from length 1 to 32 in $T$. Indeed, 98% of the prefix lengths are in the interval $16 \ldots 24$, which we call middle-class prefixes. This skew is typically a good sign for compressing data.

We can get further insight on $T$ by examining the trie storing all the prefixes in $T$ since an IP lookup is a traversal of a path from the root of the trie. The nodes of the tries are labeled with the next hops according to prefixes in $T$. Some nodes $u$ are also marked to record the fact that the path from the root to $u$ stores a prefix of the table. We can draw two cutlines on the trie, on levels 16 and 24. We obtain a set of at most $2^{16}$ sub-tries of height no more than $h = 8$. (The height is the numbering of levels, starting from 0 for the root.) For random data, we do not expect to find isomorphic sub-tries. Indeed, the probability that *no* two sub-tries are isomorphic is very near to one, i.e., $(1-p)^{2^{16}} \approx 1$ (see [1]). We instead consider a weaker notion which is more relevant in our case. Given a trie of height $h$, let us expand it to its complete form (also called prefix expansion) so that all the leaves are on the same level. Nodes are still labeled and marked according to the prefixes in $T$, except that they are now part of a complete trie

(with expanded leaves that explicitly represent all possible $2^h$ binary strings of length $h$). Note that each string is associated with its correct next hop when seen as part of an IP address.

We say that two tries are *equivalent*, if the sequence of $2^h$ next hops in the expanded leaves on level $h$ of the former is identical to that of the latter, when scanned in left-to-right order. In other words, when a lookup with $h$ bits is performed on two equivalent tries, the next hops thus returned make them indistinguishable. Note that two isomorphic tries are equivalent while the reverse is not necessarily true, since different combinations of shapes and labels/marks can yield the same sequence of next hops. We are therefore interested in selecting one representative for each class of equivalent tries. In our case, we apply this selection to the sub-tries of height at most 8 obtained from the cutlines on levels 16 and 24 (corresponding to the middle-class prefixes). How many of them are equivalent? For random data, we still expect that there are very few equivalent sub-tries. Fortunately, we observe what we call the **middle-class effect** in real routing tables $T$ when we build the trie on the prefixes in $T$:

> *Many sub-tries of height* $\leq 8$ *on level* 16 *are equivalent with lots of repetitions, and their nodes store the great majority of prefixes in* $T$.

So there is a good chance to store fewer than $2^{16}$ sub-tries by keeping just one representative for each equivalence class. Even though the majority of prefixes are middle-class (98% in our $T$), they do follow regular patterns in the routing table. In our example, table $T$ gives 13834 nonempty sub-tries of height at most 8 on level 16. Among these, we are left with 3241 representatives of equivalence classes. These are not random data at all!

### 2.3 Two-Layer Lookup Tables Exploiting the Middle-Class Effect

We now present a simple, but powerful, lookup scheme based on the middle-class effect described in Section 2.2. To be more concrete, we illustrate our approach by referring to $T$, shown in Table 1. Following what claimed in the middle-class effect, we can conceptually cut the trie built on the router prefixes, on level 24. We transform the resulting trie into a direct acyclic graph (DAG), in which equivalent sub-tries (of height at most 8) on level 16 are collapsed. This DAG can be represented by the data structure in Fig. 1, consisting of two components:
`hop`: This is the two-dimensional array of $\hat{\alpha} \times 256$ next hops, where $\hat{\alpha}$ is the number of non-equivalent sub-tries on level 16 of the DAG ($\hat{\alpha} = 3$ in our example); each such sub-trie is represented by its sequence of $2^8 = 256$ next hops *without* RLE compression.
`row`: This is the array of $2^{16}$ entries mapping the first 16 bits of IP addresses to the suitable row of `hop`. (Equivalently, they represent the children pointers of DAG nodes on level 16.)

In other words, we expand the upper part of the DAG that corresponds to the first 16 levels into `row`, and store in each row of `hop` the sequence of 256 next hops derived from each class of equivalent (collapsed) sub-tries. The pointers in `row` keep track of the corresponding sub-tries thus represented in `hop`. For any

IPv4 address $x = x_1.x_2.x_3.x_4$, the next hop obtained by searching for $x$ into the trie (compactly represented by the DAG) is that stored in $\texttt{hop}\big[\,\texttt{row}[x_1.x_2],\, x_3\,\big]$.

The above data structure forms what we call *layer 1*, which allows us to answer IP lookups by examining the first 24 bits (which is mostly the case in our collected data). The set of remaining prefixes (longer than 24 bits) in $T$ form *layer 2*, which is augmented by taking their first 24 bits. (Recall that we associate with these bits the dummy hop, e.g. 255, in layer 1.) Table 2 shows an example. Dummy prefixes of length 24 in layer 1 correspond to prefixes of length 24 with the correct next hop in layer 2. The number of such dummy prefixes cannot exceed that of prefixes longer than 24. At this stage, we do not need to choose any specific implementation of the lookup table for layer 2.

Before discussing the experimental analysis on the lookup in Section 3, we first assess the space occupancy of our scheme.

**Fact 1.** *Layer 1 occupies* $\hat{\alpha} \times 256 + 2^{16} \cdot \mathit{size}(\mathit{ptr})$ *bytes, where* $\hat{\alpha} \leq 2^{16}$ *is the number of non-equivalent sub-tries of height at most 8 on level 16, and* $\mathit{size}(\mathit{ptr}) \geq (\log_2 \hat{\alpha})/8$ *is the number of bytes encoding a pointer to* `hop`*'s rows.*

In the worst case, `hop` occupies no more than 16 Mb and `row` needs 256 Kb (using 4-byte pointers) by Fact 1. This is actually a pessimistic estimate, since we only keep the sub-tries that are *not* equivalent each other. In order to have a fair comparison of our scheme with CDG, we must add the space taken by the lookup table adopted for layer 2:

| | lookup | space (Kb) | | | | lookup | space (Kb) | | |
|---|---|---|---|---|---|---|---|---|---|
| | time | total | layer 1 | layer 2 | | time | total | layer 1 | layer 2 |
| CDG | 7012 | 2022 | 1521 | 501 | N-Way Srch | 5211 | 1608 | 1521 | 87 |
| Binary Search | 5221 | 1556 | 1521 | 35 | Binary Trie | 5758 | 1649 | 1521 | 128 |
| K-Way Search | 5274 | 1556 | 1521 | 35 | Hybrid Trie | 5297 | 1649 | 1521 | 128 |

In the above table, we report the figures for several choices with router west.attcanada, where we compare several methods for storing the prefixes in layer 2: CDG, array with binary search, $k$-way search (with $k = 8$ and $k = 2n$ where $n$ is the number of prefixes), binary tries, and hybrid tries in which the first three levels are indexed by individual bytes and the next 8 levels (at most) are indexed by individual bits. Indeed, a lookup in layer 2 surely matches at least the first 24 bits by construction. Lookup times measure the number of microseconds for running 100,000 lookups.

We computed similar tables for the other snapshots: it turns out that hybrid tries are the best trade-off between space and lookup time. Choosing hybrid tries for storing prefixes in layer 2, we report in Table 4 the space improvement with respect to CDG for the 13 benchmark tables listed in Section 2.1. As we can see, the column corresponding to our scheme gives a quite stable occupancy in space with respect to the routing table size (#prefixes). This is better highlighted if we consider the entire year 2003 of RIPE NCC, with the results for our scheme being plotted on the bottom part of Fig. 3(left).

**Table 4.** Space occupancy and time performance of our method versus CDG for the 13 benchmarks described in Section 2.1. Space is measured in kilobytes; time is measured in microseconds. The data for the columns should be read as follows. Columns 1–2 are the benchmark names and their numbers of prefixes. Columns 3–4 report the space occupancy of CDG and our method. Columns 5–6 measure their running time for 100,000 lookups on random traffic. Column 7 (hit-2) counts how many hits our lookups made in layer 2. Columns 8–10 are similar to 5–7 but refer to synthetic traffic. The figures in italic correspond to random data for the experiments in [3, 12]

| router | #prefixes | space (Kb) | | random (ms) | | | synthetic (ms) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | CDG | ours | CDG | ours | hit-2 | CDG | ours | hit-2 |
| aads | 32505 | 3706 | 1084 | *7276* | *5903* | 5463 | 7452 | 4775 | 4820 |
| att | 121711 | 2188 | 1822 | 12605 | 7351 | 15 | 7872 | 4941 | 16 |
| east.attcanada | 127561 | 16418 | 1661 | 15096 | 8429 | 3220 | 9164 | 5450 | 3116 |
| funet | 41328 | 666 | 540 | *3130* | *2461* | 88 | 5036 | 2783 | 67 |
| mae-west | 71319 | 4643 | 1290 | *7217* | *5916* | 2385 | 7425 | 4565 | 2401 |
| oregon-01 | 118190 | 9897 | 1596 | *7740* | *9933* | 11693 | 7265 | 6654 | 10651 |
| oregon-03 | 142883 | 9026 | 2164 | 14262 | 9529 | 3565 | 8790 | 6023 | 3525 |
| pacbell | 45184 | 3170 | 982 | 6126 | 5078 | 3899 | 6584 | 4233 | 3458 |
| paix | 17766 | 2745 | 875 | *6306* | *5522* | 9683 | 6934 | 4682 | 8703 |
| telstra | 104096 | 8896 | 1490 | *8468* | *7544* | 3899 | 7966 | 5317 | 3690 |
| telus | 126687 | 11390 | 1724 | 14011 | 8177 | 2095 | 8630 | 5279 | 2228 |
| west.attcanada | 127576 | 16749 | 1664 | 15071 | 8353 | 3277 | 9167 | 5350 | 3050 |
| RIPE NCC | 138201 | 5132 | 1202 | 10936 | 5922 | 1136 | 6970 | 4106 | 1074 |

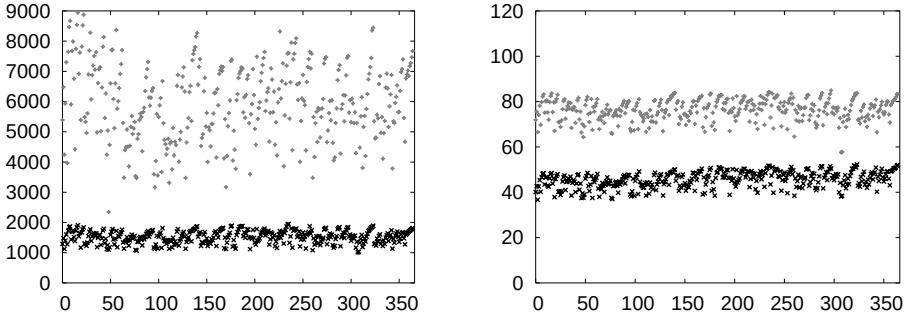



**Fig. 3.** *Left picture:* Space occupancy of our scheme vs. CDG for RIPE NCC. The x-axis reports the 365 daily snapshots of year 2003 numbered as 0...364; the y-axis the occupied space in kilobytes. *Right picture:* Number of milliseconds (on the y-axis) required by 1 million lookups in CDG (top) and in our scheme (bottom) using synthetic traffic. The x-axis reports the 365 daily snapshots of year 2003 numbered as 0...364

The net result for our scheme is a lookup table whose space occupancy scales linearly with the number of prefixes. (Clearly, layer 1 alone scales as well; moreover, its maximum size is 16Mb.) Fig. 2(right) illustrates this behavior for the available monthly snapshots of RIPE NNC, from October 1999 to April 2004,

with a number of prefixes ranging from 65841 (yielding $\hat{\alpha} = 1404$) to 138201 (yielding $\hat{\alpha} = 3241$). Here, layer 1 has a size ranging in $9n \ldots 14n$ bytes for $n$ prefixes. For the sake of comparison, a straightforward storage of these prefixes alone in a routing table would require $6n$ bytes, assuming that each prefix requires a 4-byte word of memory, and its prefix length and its next hop need one further byte each. We also computed statistics for all daily snapshots of 2003 of RIPE NCC (see Section 2.1). The total size of our lookup table (using a hybrid trie for layer 2) is in the range $7n \ldots 16n$, thus confirming the linearity of space even in this case.

## 3 Performing Lookups

The improved space bounds described in Section 2 makes our scheme more stable to use with respect to CDG. What about lookup time in IPv4? For any given IP address $x = x_1.x_2.x_3.x_4$, we keep the variable $\texttt{lx} = x_1.x_2$ storing the first 16 bits of $x$ and $\texttt{rx} = x_3.x_4$ storing the last 16 bits, so that $x = \texttt{lx.rx}$. We use the right shift operator on `rx` to get byte $x_3$ and to perform a lookup on $\texttt{lx}.x_3$. If we get the dummy value 255 in layer 1, we also need to perform a lookup in layer 2:

```
#define DUMMY  255
if ( (h1 = hop[ row[lx], rx>>8 ]) != DUMMY )
   return h1;
return lookup_layer2( lx.rx );
```

For example, an IP lookup for $x =$ `192.168.32.125` successfully stops at layer 1 by returning the next hop 3, which is located at $\texttt{hop}\big[\,\texttt{row}[\texttt{192.168}],\ \texttt{32}\,\big]$. Instead, $x =$ `192.168.0.27` requires a lookup in layer 2, since it returns the dummy value 255 stored in $\texttt{hop}\big[\,\texttt{row}[\texttt{192.168}],\ \texttt{0}\,\big]$. We measured the running time of our method and of CDG on the daily snapshots of RIPE NCC for the year 2003. We employed the synthetic traffic data for each individual snapshot as explained in Section 2.1. As can be noted in Fig. 3(right), our lookups are definitively faster than those in CDG by 30%. This is consistent with the fact that we reduce CDG's number of memory accesses from 3 to 2. It turns out that the role played by the data structures in layer 2 is rather limited in our data set. We refer the reader to columns hit-2 in the experimental data reported in Table 4, for the number of hits to layer 2 out of 100,000 lookups.

## 4 Performing Updates

We now describe one of the main effects of our simplification of the lookup scheme. We show how to efficiently handle the updates of the lookup table when announcements (i.e. insertions) and withdrawals (i.e. deletions) of routes arrive on the fly. We do not need to rebuild the lookup table from scratch. We describe how to use our method by assuming that some reasonably efficient method has been adopted for layer 2 (e.g. tries, multi-level hashing, TCAMs, etc.). Again,

we base our method on real data analysis to show that the great majority of updates involves layer 1, consistent with what was observed in the middle-class effect. We also make our scheme more robust by providing a good, exact upper bound on the number of entries changed in the lookup table in the worst case.

As described in Section 2.3, we employ `hop` and `row` for layer 1. It is crucial to observe that `hop` is stored in *row-major* order. Since we adopt the maximum number of columns, 256, the only admissible size change in `hop` is to add or remove rows. Performing this change on the columns would result in a disaster, as the whole `hop` would need to be re-allocated dynamically, which can have a cost analogous to that of rebuilding. Here is why we opt for keeping all the 256 columns. This also guarantees a high level of concurrent access to our lookup table during its lifetime.

We assume (realistically speaking) that the prefixes in route announcements and withdrawals are of length at least 8. (They can be shorter in case of heavy network failures, but then updating the routing table is a minor problem.) We also assume that there are at most 127 distinct next hop values in layer 1. We reserve the most significant bit in each entry of `hop` to mark it as a dummy. (Note that we do not use the dummy value of 255 anymore as in Section 2.3.) Masking this bit yields the correct next hop value.

We performed data analysis on the update traces for RIPE NCC. We collected all the announcements and withdrawals available for year 2003 (see Section 2.1). We discovered that for 336 days the percentage of daily updates involving layer 2 is less then 0.1%, for 360 days that percentage does not exceed the threshold of 0.2% and that the maximum percentage is less then 0.7% This confirms once again the middle-class effect, motivating our choice to build layer 1 on the first 24 bits. We suggest to use a well-tuned trie in layer 2, so that its update cost does not significantly influence the overall performance of announcements and withdrawals in a router.

### 4.1 Handling Announcements and Withdrawals

We show how to efficiently process announcements and withdrawals that are produced during the execution of the border gateway protocol (BGP). When an announcements arrives, we have to insert a certain prefix $p$ with its associated prefix length $l_p$ and next hop $h_p$, into layer 1. Recall that $8 \leq l_p \leq 32$ by our assumptions. We distinguish among three main cases for describing the worst-case effect of this insertion on `row` and `hop`:

1) Case $l_p < 16$: Since $l_p \geq 8$, we have to change no more than 256 entries in `row`. However, each of them could change up to 256 entries in `hop`. The worst case is therefore that of changing $256 + 2^{16}$ entries. In practice, the number of entries is much smaller.

2) Case $16 \leq l_p \leq 24$: This is the most frequent case according to the middle-class effect. We can change one entry of `row` to point from one row of `hop` to another, since the insertion of $p$ needs to change some entries of the row previously pointed in that entry of `row`. We may need to add a new row when

none of the existing ones match this change. In the worst case, we change no more than $1 + 256$ entries.

3) Case $l_p > 24$: We can change one entry in `row` and one in `hop`; the latter change may cause the creation of a new row in `hop` as discussed in case 2.

We use the most significant bit to mark the next hop of a truncated prefix so as to avoid that an update falling into cases 1–2 does not propagate to 24-bit prefixes in layer 2 as a side effect. Consequently, we need to make a slight change to lookup, as shown next.

```
#define BITMASK 0x80
#define NO_ROUTE_TO_HOST  0
if ( ! ((h1 = hop[ row[lx], rx>>8 ]) & BITMASK) )
   return h1;
if ( (h2 = lookup_layer2( lx.rx )) != NO_ROUTE_TO_HOST )
   return h2;
return h1 & ~BITMASK;
```

If a lookup in layer 2 returns no-route-to-host, then we must return the next hop value (with its most significant bit cleaned) previously computed in layer 1. Although it may appear that we are harming the performance of the original lookup algorithm in Section 3, we observe that the hit ratio for the first if-statement is very high and determines the real lookup cost, which stays unchanged according to the experimental evaluation discussed in Section 3.

Withdrawals have an effect on `row` and `hop` similar to announcements, except that we have to handle "hidden" prefixes. When we delete a prefix, we should find the "parent" of that prefix and propagate its next hop downward to replace that of the deleted prefix. For example, the withdrawal of route `192.168.0.0/18` from layer 1 in Table 2 causes the propagation of the next hop 2 (associated with `192.168.0.0/17`) to `192.168.0.0/24` (replacing its next hop 3) in layer 2.

As a result we add or remove one row at most in `hop`. Removed rows are linked in a free list that can be reused for adding rows. This does not change the lookup procedure and its cost.

Since the main cost is given by the number of entries changed in `row` and `hop`, we computed statistics to account for this cost, classifying it according to cases 1–3 (both for announcements and withdrawals). We processed the peak of Oct. 25–26, 2003, in router RIPE NCC:

| date | #announcements | #withdrawals | case 1 | case 2 | case 3 |
|---|---|---|---|---|---|
| 10-25-04 | 20459780 | 139787 | 0.68% | 99.31% | 0.01% |
| 10-26-04 | 11538757 | 144937 | 0.67% | 99.30% | 0.03% |

The above table shows that nearly 99.3% of the updates fall into case 2. Roughly half of them involve a prefix length $l_p = 24$, so they change just one entry in `hop`. Actually, the average number of changed entries in `row` and `hop` is nearly 1. For case 1, the most expensive one, the variance is high for a small number of updates while the rest of updates does not change any row of `hop`.

On Oct. 25, 1495 updates changed entries `row` and `hop`; on Oct. 26, there were 1889. These few updates changed between 100 and 1000 entries; we found a single example in which there were 20,985 changed entries, approaching the worst case.

The net result of the case analysis discussed so far is that updates are of bounded cost in layer 1, even in the worst case. This cost scales well with the number of updates and prefixes stored in layer 1.

**Fact 2.** *In the worst case, the announcement or withdrawal of an IPv4 route changes at most 256 entries in* `row` *and at most* $2^{16}$ *entries in* `hop` *in case 1. The number of changed entries in* `hop` *becomes* 256 *in cases 2 and 3. In all cases, the empirical average number of changed entries is nearly 1.*

## 5 Construction of the Lookup Table

The construction of our table consists in building a trie and then obtaining its DAG. It is worth noting that we insert the prefixes (truncated at 24 bits) into the trie in order of *nondecreasing prefix length*. If we do not follow this pattern, we have to propagate the next hop of the currently inserted prefix downward. In other words, we change the next hop to an already created set of nodes. If we follow the above pattern instead, we have to assign the next hop only to newly created nodes and this can happen once per node. This pattern gives a better performance in the worst case. For our tables, the most time-consuming construction was for oregon-03, in 365 milliseconds. Note that, since we can quickly handle updates, the construction time is less important than in static lookup tables.

## References

1. A.V. Aho and N.J.A. Sloane. Some doubly exponential sequences. *Fibonacci Quarterly*, 429–437, 1973.
2. M. Aida and T. Abe. Pseudo-address generation algorithm of packet destinations for internet performance simulation. In *IEEE INFOCOM*, 1425–1433, 2001.
3. A. L. Buchsbaum, G. S. Fowler, B. Kirishnamurthy, K.-P. Vo, and J. Wang. Fast prefix matching of bounded strings. *J. Exp. Algorithmics*, 8:1–3, 2003.
4. P. Crescenzi, L. Dardini, and R. Grossi. IP address lookup made fast and simple. *Proce. 7th Annual European Symposium on Algorithms*, 65–76, 1999.
5. S. Deering and R. Hinden. Internet protocol, version 6 (IPv6). RFC 1883, 1995.
6. S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *IEEE INFOCOM*, 201–212, 2003.
7. W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.
8. F. Geraci and R. Grossi. Distilling router data analysis for faster and simpler dynamic IP lookup algorithms. Tech. Report TR-05-01, Università di Pisa, January 2005.

9. G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. *Network and Distributed System Security Symposium*, 2003.
10. C. Labovitz, F. Jahanian, S. Johnson, R. Malan, S.R. Harris, J. Wan, M. Agrawal, D. Zhu, A. Ahuja, and J. Poland. Internet Performance Measurement and Analysis (IPMA) statistics. `http://www.merit.edu/ipma`, 1999.
11. B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking*, 7(3):324–334, 1999.
12. M. Pellegrini and G. Fusco. Efficient IP table lookup via adaptive stratified trees with selective reconstructions. *12th European Symp. on Algorithms*, 24–35, 2004.
13. J. Postel. Internet protocol. RFC 791, 1981.
14. Network Coordination Centre of the Réseaux IP Européens (RIPE NCC). Routing information service project (Amsterdam router). `http://www.ripe.net/ris/index.html`, 2003.
15. L. Rizzo. Personal communication, 2003.
16. M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. In *IEEE Network*, 8–23, 2001.

# Optimal Competitive Online Ray Search with an Error-Prone Robot

Tom Kamphans and Elmar Langetepe

University of Bonn, Institute of Computer Science I
Römerstraße 164, 53117 Bonn, Germany
{kamphans,langetep}@cs.uni-bonn.de

**Abstract.** We consider the problem of finding a door along a wall with a blind robot that neither knows the distance to the door nor the direction towards of the door. This problem can be solved with the well-known doubling strategy yielding an optimal competitive factor of 9 with the assumption that the robot does not make any errors during its movements. We study the case that the robot's movement is erroneous. In this case the doubling strategy is no longer optimal. We present optimal competitive strategies that take the error assumption into account.

**Keywords:** Online algorithms, motion planning, ray search, errors.

## 1 Introduction

Motion planning in unknown environments is theoretically well-understood and also practically solved in many settings. During the last decade many different objectives where discussed under several robot models. For a general overview on online motion planning problems see e. g. [3, 15, 9, 17].

Theoretical correctness results and performance guarantees often suffer from idealistic assumptions so that in the worst case a correct implementation is impossible. On the other hand, practioners analyze correctness and performance mainly statistically or empirically. Therefore it is useful to investigate, how theoretic online algorithms with idealistic assumptions behave if those assumptions cannot be fulfilled. Can we incorporate assumptions of errors in sensors and motion into the analysis?

The task of finding a point on a line by a blind agent without knowing the location of the goal was considered by Gal [6, 1] and independently reconsidered by Baeza-Yates et al. [2]. Both introduced the so-called doubling strategy, which is a basic paradigma for searching algorithms, e. g., approximating the optimal search path, see [5]. Searching on the line was generalized to searching on $m$ concurrent rays, see [8, 13, 14, 4, 7, 12].

In this paper we investigate how an error in the movement influences the correctness and the corresponding competitive factor of a strategy. The error range, denoted by a parameter $\delta$, may be known or unknown to the strategy.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 593–596, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Due to space limitations, we give only brief sketches of the proofs and refer the interested reader to [11] where we also consider a second error model.

## 2 The Standard Problem and the Error Model

The task is to find a point, $t$, on a line. Both the distance from the start position $s$ to $t$, as well as the position of $t$ (left hand or right hand to $s$) is unknown. A strategy can be described by a sequence $F = (f_i)_{i\in\mathbb{N}}$. $f_i$ denotes the distance the robot walks in the $i$-th iteration. If $i$ is even (odd), the robot moves $f_i$ steps from the start to the right (left) and $f_i$ steps back. It is assumed that the movement is correct, so after moving $f_i$ steps away from the start point and $f_i$ towards $s$, the robot has reached $s$. This does not hold if there are errors in the movement. In this case, every movement may be erroneous, which causes the robot to move more or less far than expected. We require that the robots error per unit is within a certain error bound, $\delta$. Let $f$ denote the length of a movement required by the strategy then we require that the robot moves at least $(1-\delta)f$ and at most $(1+\delta)f$ for $\delta \in [0,1[$.

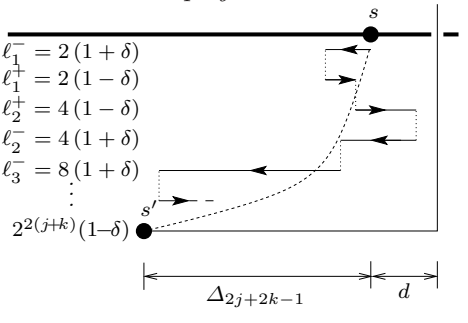
## 3 Finding a Point on a Line

First, we assume that the robot is not aware of making any errors. Thus, the optimal 9-competitive doubling strategy $f_i = 2^i$ [6, 2] seems to be the best choice for the robot. Let $\ell_i^+$ ($\ell_i^-$) be the covered distance to the right (left) in the $i$-th step. Now, the *drift* from $s$, $\Delta_k$, is $\Delta_k = \sum_{i=1}^{k}(\ell_i^- - \ell_i^+)$.

**Theorem 1.** *The robot will find the door with the doubling strategy $f_i = 2^i$, if the error $\delta$ is not greater than $\frac{1}{3}$. The generated path is never longer than $8\frac{1+\delta}{1-3\delta} + 1$ times the shortest path to the door.*

*Proof sketch.* We assume that the goal is found on the right side. For the competitivity it is the worst, if the door is hit in step $2j+2$, but located just a little bit further away than the rightmost point reached in step $2j$.

We get the worst case ratio $\frac{|\pi_{\text{onl}}|}{d} = 1 + \frac{\sum_{i=1}^{2j+1}(2\ell_i^-)}{\ell_{2j}^+ - \sum_{i=1}^{2j-1}(\ell_i^- - \ell_i^+)+\varepsilon}$ $(*)$. This maximizes for $\ell_i^{\mp} = (1\pm\delta)2^i$, and we get $\frac{|\pi_{\text{onl}}|}{d} < 1 + 8\,\frac{1+\delta}{1-3\delta}$. We have to require $\delta \le \frac{1}{3}$, otherwise the distance $(1-3\delta)\,2^{2j} + 4\delta$ from $s$ may not exceed the point $4\delta$ . □




One might wonder if there is a strategy which takes the error $\delta$ into account and yields a smaller factor. Intuitively this seems to be impossible, but we are able to show that there is such a strategy.

**Theorem 2.** *In the presence of an error up to $\delta$ there is a strategy that meets every goal and achieves a competitive factor of* $1 + 8\left(\frac{1+\delta}{1-\delta}\right)^2$.

*Proof sketch.* We design a strategy, $F = (f_i)_{i\in\mathbb{N}}$. From (*) we conclude that it is sufficient to minimize $G_{(n,\delta)}(F) := \frac{\sum_{i=1}^{n+1} f_i}{(1-\delta)f_n - 2\delta\sum_{i=1}^{n-1} f_i}$, which is achieved by the strategy $f_i = \left(2\,\frac{1+\delta}{1-\delta}\right)^i$. This strategy is reasonable since it monotonically increases the distance to $s$, and we reach every goal. □

This factor is optimal. We can show that for every $\delta$ there is a strategy, $F^*$, that achieves the optimal factor $C_\delta$ exactly in every step, and describe $F^*$ by a recurrence. Finally, the condition $f_i > 0$ leads to a lower bound for $C_\delta$. Thus:

**Theorem 3.** *In the presence of an error up to $\delta \in [0, 1[$, there is no competitive strategy that yields a factor smaller than* $1 + 8\left(\frac{1+\delta}{1-\delta}\right)^2$.

## 4 Error-Prone Searching on $m$ Rays

The robot is located at the common endpoint of $m$ infinite rays, knowing neither the location—the ray containing $t$— nor the distance to $t$. Gal [6] showed that w.l.o.g. one can use a *periodic* and *monotone* strategy, i. e., $f_i$ and $f_{i+m}$ visit the same ray, and $f_i < f_{i+m}$ holds. In the error-prone setting, the start point of every iteration cannot drift away, since the start point is the only point where all rays meet.

**Theorem 4.** *Searching for a target located on one of $m$ rays with an error-prone robot using a monotone and periodic strategy is competitive with an optimal factor of* $3 + 2\,\frac{1+\delta}{1-\delta}\left(\frac{m^m}{(m-1)^{m-1}} - 1\right)$ *for* $\delta < \frac{e-1}{e+1}$.

*Proof sketch.* It turns out that we consider the functionals $G_k(F) := \frac{\sum_{i=1}^{k+m-1} f_i}{f_k}$ in this case, which are identical to the functionals considered in the error-free $m$-ray search. Thus, $f_i = (m/m-1)^i$ minimizes $G_k(F)$, see [2, 6]. Ensuring monotony leads to the condition $\delta < \frac{e-1}{e+1}$. □

## 5 Summary

We have analyzed the standard doubling strategy in the presence of errors in movements. The robot still reaches the goal for $\delta \le \frac{1}{3}$ with a competitive ratio of $8\frac{1+\delta}{1-3\delta} + 1$. If $\delta$ is known to the strategy $f_i = \left(2\,\frac{1+\delta}{1-\delta}\right)^i$ is optimal with a competitive factor of $1 + 8\left(\frac{1+\delta}{1-\delta}\right)^2$. In the case of $m$ rays $f_i = (m/m-1)^i$ yields $3 + 2\,\frac{1+\delta}{1-\delta}\left(\frac{m^m}{(m-1)^{m-1}} - 1\right)$ for $\delta \le \frac{e-1}{e+1} \approx 0.46$.

## References


1. S. Alpern and S. Gal. *The Theory of Search Games and Rendezvous*. Kluwer Academic Publications, 2003.
2. R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Inform. Comput.*, 106:234–252, 1993.
3. P. Berman. On-line searching and navigation. In A. Fiat and G. Woeginger, editors, *Competitive Analysis of Algorithms*. Springer-Verlag, 1998.
4. E. D. Demaine, S. P. Fekete, and S. Gal. Online searching with turn cost. Submitted to Theor. Comput. Sci.
5. R. Fleischer, T. Kamphans, R. Klein, E. Langetepe, and G. Trippen. Competitive online approximation of the optimal search ratio. In *Proc. 12th Annu. European Sympos. Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 335–346, Heidelberg, 2004. Springer-Verlag.
6. S. Gal. *Search Games*, volume 149 of *Mathematics in Science and Engeneering*. Academic Press, New York, 1980.
7. M. Hammar, B. J. Nilsson, and S. Schuierer. Parallel searching on $m$ rays. *Comput. Geom. Theory Appl.*, 18:125–139, 2001.
8. C. Hipke, C. Icking, R. Klein, and E. Langetepe. How to find a point on a line within a fixed distance. *Discrete Appl. Math.*, 93:67–73, 1999.
9. C. Icking, T. Kamphans, R. Klein, and E. Langetepe. On the competitive complexity of navigation tasks. In H. Bunke, H. I. Christensen, G. D. Hager, and R. Klein, editors, *Sensor Based Intelligent Robots*, volume 2238 of *Lecture Notes Comput. Sci.*, pages 245–258, Berlin, 2002. Springer.
10. T. Kamphans. *Models and Algorithms for Online Exploration and Search*. PhD thesis, University of Bonn, to appear.
11. T. Kamphans and E. Langetepe. Optimal competitive online ray search with an error-prone robot. Technical Report 003, University of Bonn, 2005. http://web.informatik.uni-bonn.de/I/publications/kl-ocolr-05t.pdf.
12. M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. *Inform. Comput.*, 133(1):63–79, 1996.
13. E. Langetepe. *Design and Analysis of Strategies for Autonomous Systems in Motion Planning*. PhD thesis, Department of Computer Science, FernUniversität Hagen, 2000.
14. A. López-Ortiz and S. Schuierer. The ultimate strategy to search on $m$ rays? *Theor. Comput. Sci.*, 261(2):267–295, 2001.
15. J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
16. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoret. Comput. Sci.*, 84(1):127–150, 1991.
17. N. S. V. Rao, S. Kareti, W. Shi, and S. S. Iyengar. Robot navigation in unknown terrains: introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, 1993.

# An Empirical Study for Inversions-Sensitive Sorting Algorithms

Amr Elmasry[1,2,⋆] and Abdelrahman Hammad[1]

[1] Dept. of Computer Engineering and Systems, Alexandria University, Egypt
[2] Faculty of Science and Information Technology, Al-Zaytoonah University, Jordan

**Abstract.** We study the performance of the most practical internal adaptive sorting algorithms. Experimental results show that adaptive AVL sort performs the least number of comparisons unless the number of inversions is fewer than 1%. In such case, Splaysort performs the fewest number of comparisons. On the other hand, the running time of Quicksort is superior unless the number of inversions is fewer than 1.5%. In such case, Splaysort consumes the smallest running time.

## 1 Introduction

A sorting algorithm is considered adaptive if it performs better for sequences having some existing order. One of the main measures of presortedness is the number of inversions in the input sequence $Inv$; that is the number of pairs in the wrong order. For an adaptive sorting algorithm to be optimal with respect to the number of inversions, its running time should be in $O(n \log \frac{Inv}{n} + n)$ [6].

Among the sorting algorithms optimal with respect to the number of inversions, Splitsort [8] and adaptive Heapsort [9] are the most promising from the practical point of view. Splaysort, sorting by repeated insertion in a splay tree [11], was proved to be optimal with respect to the number of inversions [2]. Moffat et al. [10] performed experiments showing that Splaysort is practically efficient. See [5] as a nice survey of adaptive sorting algorithms. A later study, oriented towards improving the number of comparisons for inversions-optimal sorting algorithms, introduces adaptive AVL sort [4].

On the other hand, Quicksort introduced by Hoare [7] is considered the most practical sorting algorithm. Several empirical studies illustrated that Quicksort is very efficient in practice [13]. The Quicksort algorithm and its variants also demonstrated an efficient performance when applied to nearly sorted lists [3]. A very recent result of Brodal et al. [1] shows that the expected number of swaps performed by randomized Quicksort is $O(n \log \frac{Inv}{n} + n)$.

In this paper, we are interested in demonstrating a practical study of such inversions-optimal sorting algorithms. Our objective is to conclude which of these

⋆ This work was done while the author was visiting Al-Zaytoonah University. Corresponding author. E-mail address: elmasry@alexeng.edu.eg

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 597–601, 2005.
© Springer-Verlag Berlin Heidelberg 2005

algorithms would be a good candidate in practice, and which one we would use under different circumstances. We also compare the performance of these algorithms with Quicksort.

## 2 Experimental Settings

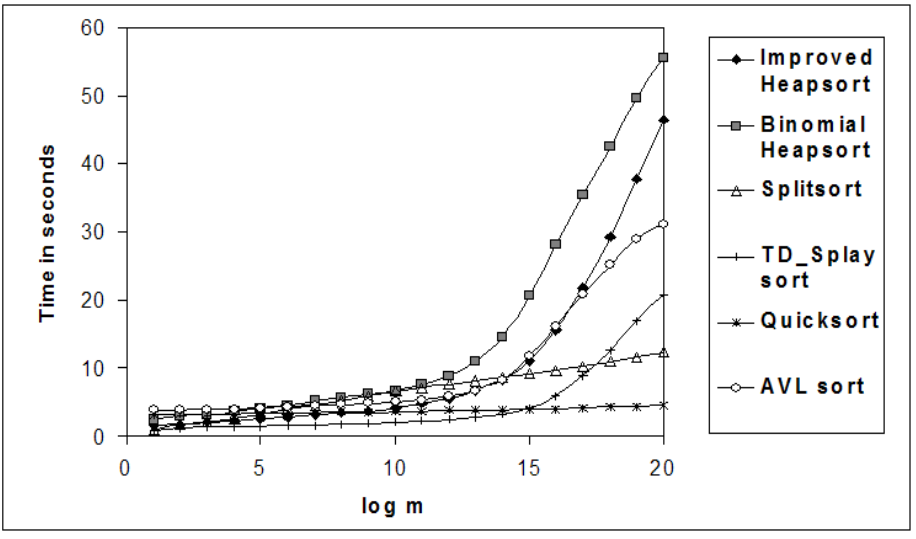
We implemented three versions of adaptive Heapsort; the basic implementation that uses a binary heap, an improved version for heap operations [9], and another variation that uses binomial queues. For Splitsort, we implemented two versions; the first uses an auxiliary array of pointers [8], while the second relies on linked structures. It turns out that the running time of the array-based Splitsort is less than the linked implementation. Two variations of the Splaysort algorithm were implemented, one using bottom-up splaying and the other using top-down(TD) splaying [10]. It turns out that the running time for the top-down version is much less. Finally we implemented adaptive AVL sort as in [4].

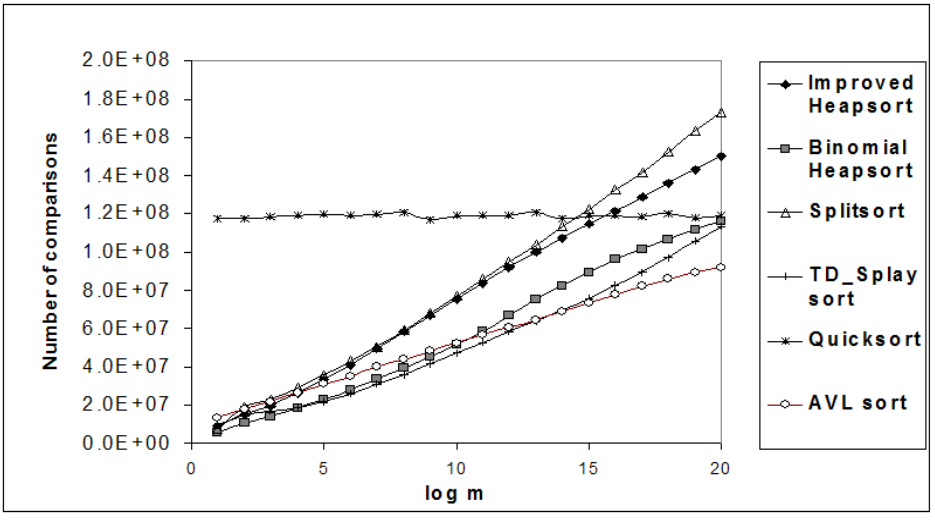
In all the experiments, we start with a sorted sequence $< 1, 2, \ldots, n >$ and perform two phases of permutations. For a given value of a parameter $m$, we want to permute the sorted sequence to have at most $n \cdot m$ with expected $\approx n \cdot m/2$ inversions. For the first phase, the sequence is broken into consecutive $\lceil n/m \rceil$ blocks, almost equally-sized. The elements of each block are randomly permuted, for a total of at most $n \cdot m/2$ inversions. For the second phase, the sorted sequence is broken into $m$ consecutive blocks, almost equally-sized. From each block we select one element at random, and these $m$ elements are then randomly permuted, for at most another $n \cdot m/2$ inversions. Hence, the resulting average number of inversions per element will be at most $m$.

Note that the way we produce the input guarantees that the elements of the input sequence are distinct. It is still an issue to check the performance of the given algorithms when there are duplicate elements. As an example, the adaptive properties of the used version of Heapsort are only correct when applied to distinct elements. Another issue is that we applied the algorithms for integer sorting; all our elements are 4-byte integers. The performance may be different if we apply the algorithms to sort real numbers or other data types.

We fixed $n$ at $2^{22}$. All the experiments were performed on an Athlon750 machine with 384 MB RAM and 256 KB cache, running Windows 2000 platform. With such memory capacity, there was no need to use virtual memory. All the algorithms are implemented in C++ using Borland C++ builder version 5.

Two outcomes are measured; the number of comparisons and the running time. The measured running times are CPU times, ignoring the time for I/O operations. For each point represented in the graphs, the average of 10 sample runs was taken. We first proceeded by investigating different alternatives and implementation issues for each algorithm. Then, we concluded by selecting the best method from each algorithm and comparing them with each other.




## 3 Experimental Results

One of our main observations for the running time of the algorithms that require linked structures is that the dynamic allocations of such structures is time consuming. We noticed that more than 15% of the time of such algorithms (Heapsort, Splaysort and AVL sort) was used for such memory allocations. Our solution was to allocate a chunk of memory at the beginning of the algorithm. It was not hard to decide the memory requirement for Heapsort and

Splaysort. For AVL sort, the theoretical results in [4] indicated that the number of combines (extra nodes needed throughout the algorithm) is linear. Working through these proofs, we came up with the exact constant which is at most $3n$ combines.

Comparing the running time of the other algorithms with Quicksort, these algorithms were better for low inversions. As the number of inversions increases, Quicksort is preferable for both the running time and the storage requirements. As illustrated by the graph, TD-Splaysort shows the best performance when the number of inversions is low. It is better than Quicksort as long as the number of inversions per element is less than $2^{15}$ ($n = 2^{22}$). Splitsort, although not the best with low inversions, is better than Splaysort for high inversions, while it is not much worse than Splaysort with low inversions. In addition, it has less storage requirements. It is also interesting to demonstrate that Quicksort is adaptive with respect to time, this is a consequence of the result of [1] that Quicksort is optimally adaptive with respect to the number of swaps.

Experimental results show that the number of cache misses, and in effect the running time, is related to the number of inversions and the storage complexity of the underlying algorithm. For the algorithms with large storage requirements, the curves have sharp bends when the inversions increase (at $m \approx 13$). This illustrates that these algorithms are adaptive with respect to cache misses.

Regarding the number of comparisons, the adaptive algorithms under study were noticeably much better than Quicksort for low number of inversions. Some of the algorithms perform less comparisons even for random lists. Empirical results show that all the adaptive algorithms under study perform $c \cdot n \log \frac{Inv}{n} + O(n)$ comparisons, with $c$ between 1 and 2. For the AVL sort, $c \approx 1$.

## References


1. G. Brodal, R. Fagerberg and G. Moruz. *On the adaptiveness of quicksort.* 7th (ALENEX) Workshop on Algorithm Engineering and Experiments (2005).
2. R. Cole. *On the dynamic finger conjecture for splay trees. Part II: The proof.* SIAM J. Computing **30** (2000), 44-85.
3. R. Cook and J. Kim. *Best sorting algorithms for nearly sorted lists.* Commun. ACM **23** (1980), 620-624.
4. A. Elmasry. *Adaptive sorting with AVL trees.* 3rd IFIP-WCC International Conference on Theoretical Computer Science (2004), 315-324.
5. V. Estivill-Castro and D. Wood. *A survey of adaptive sorting algorithms.* ACM Computing Surveys **24(4)** (1992), 441-476.
6. L. Guibas, E. McCreight, M. Plass and J. Roberts. *A new representation of linear lists.* 9th ACM (STOC) Symposium on Theory of Computing **9** (1977), 49-60.
7. C. Hoare. *Algorithm 64: Quicksort.* Commun. ACM **4(7)** (1961), 321.
8. C. Levcopoulos and O. Petersson. *Splitsort - An adaptive sorting algorithm.* Information Processing Letters **39** (1991), 205-211.
9. C. Levcopoulos and O. Petersson. *Adaptive Heapsort.* J. Alg. **14** (1993), 395-413.
10. A. Moffat, G. Eddy and O. Petersson. *Splaysort: fast, versatile, practical.* Softw. Pract. and Exper. **126(7)** (1996), 781-797.

11. D. Sleator and R. Tarjan. *Self-adjusting binary search trees.* J. ACM **32(3)** (1985), 652-686.
12. J. Vuillemin. *A unifying look at data structures.* Commu. ACM **23** (1980), 229-239.
13. R. Wainwrigh. *A class of sorting algorithms based on quicksort.* Commun. ACM **28(4)** (1985), 396-402.

# Approximation Algorithm for Chromatic Index and Edge-Coloring of Multigraphs*

Martin Kochol[1,2], Naďa Krivoňáková[2], and Silvia Smejová[2]

[1] MÚ SAV, Štefánikova 49, 814 73 Bratislava 1, Slovakia
kochol@savba.sk
[2] FPV ŽU, Hurbanova 15, 010 26 Žilina, Slovakia
{nada.krivonakova, silvia.smejova}@fpv.utc.sk

**Abstract.** We introduce an invariant $\pi(G)$ on a graph $G$, which is the upper bound for the chromatic index of $G$, and show that $\pi(G)$ and a $\pi(G)$-edge-coloring of $G$ can be found in polynomial time. This generalizes the classical edge-coloring theorems of Shannon and Vizing.

## 1 Introduction

Let $\chi'(G)$ be the *chromatic index* of a graph $G$ (the minimum number $d$ such that $G$ has a proper edge-coloring by $d$ colors). By Holyer [1], it is an NP-hard problem to evaluate $\chi'(G)$. On the other hand classical theorems of Shannon [2] and Vizing [3] say that $\chi'(G) \leq \lfloor \frac{3}{2}\Delta(G) \rfloor$ and $\chi'(G) \leq \Delta(G)+p(G)$, respectively. These estimates give easy computable upper bounds for $\chi'(G)$. Now we generalize these results from algorithmical and theoretical points of view.

We deal with finite graphs with multiple edges and without loops. If $G$ is a graph, then $V(G)$ and $E(G)$ denote the sets of vertices and edges of $G$, respectively. If $v$ and $e$ is a vertex and an edge of a graph $G$, then $d_G(v)$ and $p_G(e)$ denote the degree of $v$ and the multiplicity of $e$ (the number of edges of $G$ with the same ends as $e$), respectively. Let $\Delta(G)$ and $p(G)$ denote the maximum degree of a vertex and the maximum multiplicity of an edge of $G$, respectively.

An edge $e = uv$ of $G$ is called *$r$-critical* on $u$ if

$r \geq d_G(v) + d_G(u) - p_G(e)$, and

for every edge $e' = uv'$, $v' \neq v$ we have $d_G(v') + p_G(e') \leq r$.

Let $\rho_G(u)$ denote the smallest integer $r$ such that either $d_G(v'') + p_G(e'') \leq r$ for every edge $e'' = uv''$ incident to $u$, or there exists an edge $e$ which is $r$-critical on $u$. Denote $\pi_G(u) = \max\{d_G(u), \rho_G(u)\}$.

Let $v_1, \ldots, v_n$ be an ordering of the vertices of a graph $G$. Furthermore, let $G_1 = G$ and $G_{i+1} = G_i - v_i$ for $i = 1, \ldots, n-1$. We say that $v_1, \ldots, v_n$ is an *$r$-ordering* of $G$ if $\pi_{G_i}(v_i) \leq r$ for every $i = 1, \ldots, n$. By a *supermultiplicity* of $G$, denoted by $\pi(G)$, we mean the smallest integer $r$ such that there exists an

* This work was supported by Science and Technology Assistance Agency under the contract No. APVT-51-027604 and partially by VEGA grant 2/4004/04.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 602–605, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$r$-ordering of $G$. Clearly, $\rho_{G-e}(u) \leq \rho_G(u)$ and $\pi_{G-e}(u) \leq \pi_G(u)$ for every $e \in E(G)$ and $u \in V(G)$. Thus $\pi(G)$ is a monotone invariant, i. e., $\pi(G-e) \leq \pi(G)$ for every edge $e$ of $G$.

We prove that $\chi'(G) \leq \pi(G) \leq \lfloor \frac{3}{2}\Delta(G) \rfloor$, $\Delta(G) + p(G)$ for every graph $G$. This generalizes the theorems of Shannon [2] and Vizing [3]. We present a polynomial algorithm which, for every graph $G$, finds $\pi(G)$, a $\pi(G)$-ordering of $G$, and a $\pi(G)$-edge-coloring of $G$. Furthermore we show that the problem to decide whether $\chi'(G) < \pi(G)$ is NP-complete.

## 2 Upper Bounds for Chromatic Index

Suppose we have a proper coloring of edges of a graph $G$ by colors $1, \ldots, r$ and $s, t \in \{1, \ldots, r\}$, $s \neq t$. Then by an $(s,t)$-*path* we mean a component of the graph arising from $G$ after deleting all edges having colors different from $s$ and $t$ (which is either an isolated vertex, or an even circuit, or a path). By a *recoloring* of an $(s,t)$-path we mean a process so that the edges of the $(s,t)$-path colored by $s$ (resp. $t$) receive color $t$ (resp. $s$) and the colors of all other edges remain unchanged. We say that a color $s \in \{1, \ldots, r\}$ *lacks* in a vertex $u$ of $G$ if no edge incident to $u$ is colored by $s$. The following lemmas are proved in [3].

**Lemma 1.** *Suppose we have a graph with a proper edge-coloring. Then recoloring any $(s,t)$-path results in another proper edge-coloring.*

**Lemma 2.** *Let $G$ have a proper edge-coloring by colors $1, \ldots, r$ and $u_1, u_2, u_3$ be pairwise different vertices of $G$. Suppose that there are $s, t \in \{1, \ldots, r\}$, $s \neq t$, such that for every $i = 1, 2, 3$, there exists $s_i \in \{s, t\}$ which lacks in $u_i$. Then there exists $j \in \{1, 2, 3\}$ such that no $(s,t)$-path joins $s_j$ with $s_i$ for any $i \in \{1, 2, 3\} \setminus \{j\}$.*

**Theorem 1.** *For every graph $G$, $\chi'(G) \leq \pi(G)$.*

*Proof.* We use induction on $|E(G)|$. The statement is true if $|E(G)| = 0$. Let $|E(G)| > 0$ and $v_1, \ldots, v_n$ be a $\pi(G)$-ordering of $G$. Set $u = v_1$. If every edge $e'' = uv''$ of $G$ satisfies $d_G(v'') + p_G(e'') \leq \pi(G)$, choose an arbitrary edge $e = uu_0$, incident to $u$. Otherwise choose $e = uu_0$ so that $e$ is $\pi(G)$-critical on $u$. Then $\pi(G-e) \leq \pi(G)$, and by the induction hypothesis, $G-e$ has a proper edge-coloring by $1, \ldots, \pi(G)$. Let $L$ ($L_0$) be the set of colors lacking in $u$ ($u_0$).

If $d_G(u_0) + p_G(e) \leq \pi(G)$, then $d_{G-e}(u_0) < \pi(G)$. If $e$ is $\pi(G)$-critical on $u$, then $d_G(u_0) \leq \pi(G) - d_G(u) + p_G(e) \leq \pi(G)$, whence $d_{G-e}(u_0) < \pi(G)$. Thus $d_{G-e}(u_0) < \pi(G)$ and $L_0 \neq \emptyset$. Similarly $d_{G-e}(u) < d_G(u) \leq \pi(G)$ and $L \neq \emptyset$.

A sequence of colors $s_0, \ldots, s_{k-1}$ is called *semistrong* (of order $k \geq 1$) if there exist edges $e_1 = uu_1, \ldots, e_k = uu_k$ of $G-e$ so that:

(a) edges $e_1, \ldots, e_k$ are colored by $s_0, \ldots, s_{k-1}$, respectively;
(b) colors $s_0, \ldots, s_{k-1}$ are pairwise different;
(c) colors $s_0, \ldots, s_{k-1}$ lack in $u_0, \ldots, u_{k-1}$, respectively.

We show that (a)–(c) imply the following:
(d) $s_0, \ldots, s_{k-1} \notin L$;
(e) edges $e_1, \ldots, e_k$ are pairwise different;
(f) $u_i \neq u_{i+1}$ for every $i = 0, \ldots, k-1$;
(g) vertices $u_0, \ldots, u_k$ are different from $u$.
Really, (d) and (e) follows from (a) and (b), respectively, and (f) follows from (a) and (c). (g) follows from the fact that $u_0, \ldots, u_k$ are adjacent with $u$.
A sequence of colors $s_0, \ldots, s_{k-1}, s_k$ is called *strong* (of order $k \geq 0$) if ($e_1, \ldots, e_k$ and $u_0, \ldots, u_k$ have the same meaning as above):
either $k = 0$ and $s_0 \in L \cap L_0$,
or $k > 0$, $s_k$ lacks in $u$ and $u_k$, and $s_0, \ldots, s_{k-1}$ is semistrong.
From this sequence we can construct a proper edge-coloring of $G$ by colors $1, \ldots, \pi(G)$, because changing the colors of edges $e_1, \ldots, e_k$ to colors $s_1, \ldots, s_k$, respectively, we obtain a proper edge-coloring of $G - e$ so that $s_0$ lacks in $u$ and $u_0$, and we can color $e$ by $s_0$.
Choose $s_0 \in L_0 \neq \emptyset$. If $s_0 \in L$, then the sequence $s_0$ is strong, whence $\chi'(G) \leq \pi(G)$. If $s_0 \notin L$, i. e., there is and edge $e_1 = uu_1$ of $G - e$ colored by $s_0$, then the sequence $s_0$ is semistrong of order 1. Let $s_0, \ldots, s_{k-1}$ be a semistrong sequence of the largest possible order $k \geq 1$. Note that by (e), $k \leq \Delta - 1$. First we show that there exists a color $s_k$ so that ($e_1, \ldots, e_k$ and $u_0, \ldots, u_k$ have the same meaning as above):
(h) $s_k$ lacks in $u_k$,
(i) $s_k \neq s_i$ if $u_k = u_i$, $i \in \{0, \ldots, k-1\}$.

If $d_G(u_k) + p_G(e_k) \leq \pi(G)$, then $d_{G-e}(u_k) + p_{G-e}(e_k) + 1 < \pi(G)$, whence by (e), there exists a color $s_k$ satisfying (h) and (i). If $d_G(u_k) + p_G(e_k) > \pi(G)$, then $e$ is $\pi(G)$-critical on $u$ and $e_k$ must be parallel with $e$. Thus $u_k = u_0$ and $\pi(G) \geq d_G(u_0) + d_G(u) - p_G(e)$, in other words, $u$ is incident with at most $\pi(G) - d_G(u_0)$ edges of $G$ not parallel with $e_k$. Then $u_k$ can be equal to $u_i$ (for $i \in \{0 \ldots, k-1\}$) in at most $\pi(G) - d_G(u_0)$ cases (because if $u_{i_1} = \ldots = u_{i_r} = u_0$, then by (f), $u_{i_1+1}, \ldots, u_{i_r+1} \neq u_0$, thus the edges $e_{i_1+1}, \ldots, e_{i_r+1}$ are not parallel with $e_k$, and by (e), they are also pairwise different, whence $r \leq \pi(G) - d_G(u_0) < \pi(G) - d_{G-e}(u_0)$). Thus there exists a color $s_k$ satisfying (h) and (i).

Now one of the following three cases must occur.
*Case* 1: $s_k \in L$, i. e., $s_0, \ldots, s_k$ is a strong sequence, whence $\chi'(G) \leq \pi(G)$.
*Case* 2: $s_k \notin L$ and there exists $0 \leq j \leq k-1$ such that $s_k = s_j$. By (b), $j$ is unique and since $s_k$ ($s_{k-1}$) lacks (does not lack) in $u_k$, $j \neq k-1$. By (i), $u_k \neq u_j$ and by (g), $u_k \neq u \neq u_j$. Since $L \neq \emptyset$, there exists $t \in L$. Then $t \neq s_j$ and colors $s_j, t, s_j$ lack in $u_k, u, u_j$, respectively. By Lemma 2, one of the vertices $u_k, u, u_j$ is not joined with the other two by an $(s_j, t)$-path. If $u_k$ (resp. $u$ and $u_j$) is not joined by an $(s_j, t)$-path with $u, u_j$ (resp. $u_k, u_j$ and $u_k, u$), then recoloring the $(s_j, t)$-path beginning in $u_k$ (resp. $u$ and $u_j$), we get that $t$ (resp. $s_j$ and $t$) lacks in $u_k$ (resp. $u$ and $u_j$). Thus the sequence $s_0, \ldots, s_{k-1}, t$ (resp. $s_0, \ldots, s_j$ and $s_0, \ldots, s_{j-1}, t$) is strong for the new edge-coloring of $G - e$, i.e., $\chi'(G) \leq \pi(G)$.
*Case* 3: $s_k \notin L$ and $s_k \neq s_i$ for $i = 0, \ldots, k-1$. Then there exists an edge $e_{k+1} = uu_{k+1}$ of $G - e$ having color $s_k$ and by (h), $s_0, \ldots, s_k$ is a semistrong sequence of order $k+1$, a contradiction with the choice of $k$. □

**Corollary 1.** *If a graph $G$ satisfies $\Delta(G) = \Delta$ and for every edge $e$ of $G$, either $p_G(e) \leq p$, or $p_G(e) \geq \Delta - p$ $(1 \leq p \leq \Delta/2)$, then $\chi'(G) \leq \pi(G) \leq \Delta + p$. In particular, $\chi'(G) \leq \lfloor \frac{3}{2}\Delta(G) \rfloor, \Delta(G) + p(G)$.*

*Proof.* Clearly, $\pi_G(v) \leq \Delta + p$ and $\pi_H(v) \leq \pi_G(v)$ for every $v \in V(G)$ and every subgraph $H$ of $G$. Thus $\pi(G) \leq \Delta + p$, whence by Theorem 1, $\chi'(G) \leq \Delta + p$. Setting $p = \lfloor \Delta/2 \rfloor$, we get $\chi'(G) \leq \lfloor \frac{3}{2}\Delta(G) \rfloor, \Delta(G) + p(G)$. □

## 3 Algorithms and Complexity

**Theorem 2.** *There exists a polynomial time algorithm which, for any graph $G$, finds $\pi(G)$, a $\pi(G)$-ordering of $G$, and a $\pi(G)$-edge-coloring of $G$.*

*Proof.* For each $v \in V(G)$, $\pi_G(v)$ can be found in polynomial time. Let $H_1 = G$ and choose $w_1 \in V(H_1)$ so that $\pi_{H_1}(w_1) = \min_{v \in H_1} \pi_{H_1}(v)$. For $i = 2, \ldots, n$, let $H_i = H_{i-1} - w_{i-1}$ and choose $w_i \in V(H_i)$ so that $\pi_{H_i}(w_i) = \min_{v \in H_i} \pi_{H_i}(v)$.

Define $\pi'(G) = \max_{i=1,\ldots,n} \pi_{H_i}(w_i)$. Then $\pi(G) \leq \pi'(G)$, because $w_1, \ldots, w_n$ is a $\pi'(G)$-ordering of $G$. Using induction on $n$ we show that $\pi'(G) \leq \pi(G)$. Let $v_1, \ldots, v_n$ be a $\pi(G)$-ordering of $G$. Then $\pi_G(w_1) \leq \pi_G(v_1) \leq \pi(G)$ by the choice of $w_1$, and $\pi(G - w_1) \leq \pi(G)$ by the monotonicity of $\pi$. By the induction hypothesis, $\pi'(G - w_1) \leq \pi(G - w_1)$, whence $\pi'(G) = \max\{\pi_G(w_1), \pi'(G - w_1)\} \leq \max\{\pi(G), \pi(G - w_1)\} = \pi(G)$. Thus $\pi'(G) = \pi(G)$ and we can find $\pi(G)$ and a $\pi(G)$-ordering $w_1, \ldots, w_n$ of $G$ in polynomial time.

Now choose an edge $e$ incident to $w_1$ so that either every edge $e'' = w_1 v''$ satisfies $d_G(v'') + p_G(w_1 v'') \leq \pi(G)$ or $e$ is $\pi(G)$-critical on $w_1$. From the proof of Theorem 1 it follows that any $\pi(G)$-edge-coloring of $G - e$ can be transformed to a $\pi(G)$-edge-coloring of $G$ after polynomially many steps. □

**Theorem 3.** *It is an* NP*-complete problem to decide whether $\chi'(G) < \pi(G)$. This problems remains* NP*-complete for simple cubic graphs.*

*Proof.* Let $G$ be a simple cubic graph, i.e., $p_G(e) = 1$ for every $e \in E(G)$ and $d_G(v) = 3$ for every $v \in V(G)$. Then $\rho_G(v) = \pi_G(v) = 4$ for every $v \in V(G)$. Thus $\pi(G) = 4$ and $G$ is 3-edge-colorable if and only if $\chi'(G) < \pi(G)$. By Holyer [1], it is an NP-complete problem to decide whether $G$ is 3-edge-colorable. This proves the statement □

## References

1. I. Holyer, The NP-completeness of edge-coloring, SIAM J. Comput. **10** (1981) 718-720.
2. C. E. Shannon, A theorem on coloring the lines of a network, J. Math. Phys. **28** (1949) 148–151.
3. V. G. Vizing, On an estimate of the chromatic class of a $p$-graph, Diskret. Analiz **3** (1964) 25-30 (in Russian).

# Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study*

Thomas Schank and Dorothea Wagner

University of Kalrsruhe, Germany

## 1 Introduction

In the past, the fundamental graph problem of triangle counting and listing has been studied intensively from a theoretical point of view. Recently, triangle counting has also become a widely used tool in network analysis. Due to the very large size of networks like the Internet, WWW or social networks, the efficiency of algorithms for triangle counting and listing is an important issue. The main intention of this work is to evaluate the practicability of triangle counting and listing in very large graphs with various degree distributions. We give a surprisingly simple enhancement of a well known algorithm that performs best, and makes triangle listing and counting in huge networks feasible. This paper is a condensed presentation of [SW05].

## 2 Definitions

Let $G = (V, E)$ be an undirected, simple graph with a set of nodes $V$ and a set of edges $E$. We use the symbol $n$ for the *number of nodes* and the symbol $m$ for the *number of edges*. The *degree* $d(v) := |\{u \in V : \exists\{v, u\} \in E\}|$ of node $v$ is defined to be the number of nodes in $V$ that are adjacent to $v$. The *maximal degree* of a graph $G$ is defined as $d_{\max}(G) = \max\{d(v) : v \in V\}$. An *n-clique* is a complete graph with $n$ nodes. Unless otherwise declared we assume graphs to be connected. A *triangle* $\Delta = (V_\Delta, E_\Delta)$ of a graph $G = (V, E)$ is a three node subgraph with $V_\Delta = \{u, v, w\} \subset V$ and $E_\Delta = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subset E$. We use the symbol $\delta(G)$ to denote the *number of triangles* in graph $G$. Note that an $n$-clique has exactly $\binom{n}{3}$ triangles and asymptotically $\delta_{\text{clique}} \in \Theta(n^3)$. In dependency to $m$ we have accordingly $\delta_{\text{clique}} \in \Theta(m^{3/2})$ and by concentrating as many edges as possible into a clique we observe that there exists a family of graphs $G_m$, such that $\delta(G_m) \in \Theta(m^{3/2})$.

* This work was partially supported by the DFG under grant WA 654/13-2, by the European Commission - Fet Open project COSIN - COevolution and Self-organization In dynamical Networks - IST-2001-33555, and by the EU within the 6th Framework Programme under contract 001907 (integrated project DELIS).

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 606–609, 2005.
© Springer-Verlag Berlin Heidelberg 2005

## 3 Algorithms

We call an algorithm a *counting algorithm* if it outputs the number of triangles $\delta(v)$ for each node $v$ and a *listing algorithm* if it outputs the three participating nodes of each triangle. A listing algorithm requires at least one operation per triangle. For the running time we get worst case lower bounds of $\Omega(n^3)$ in terms of $n$ and $\Omega(m^{3/2})$ in terms of $m$ by the observation in Section 2.

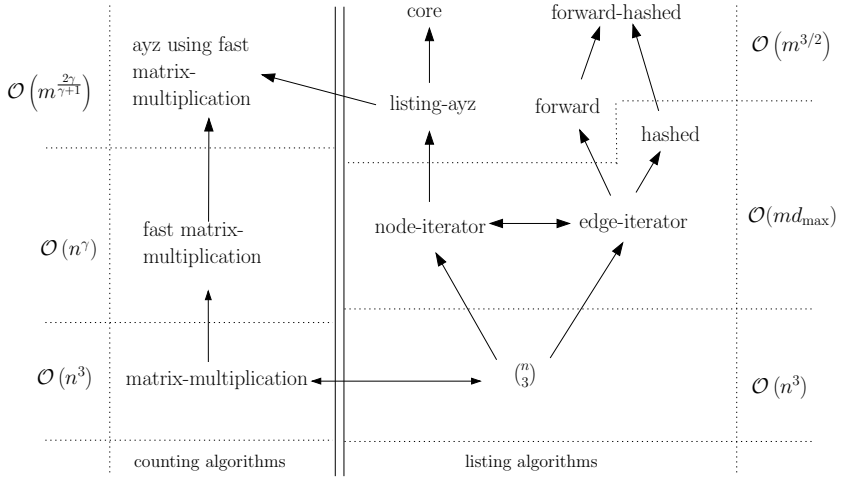



**Fig. 1.** An overview of the presented algorithms

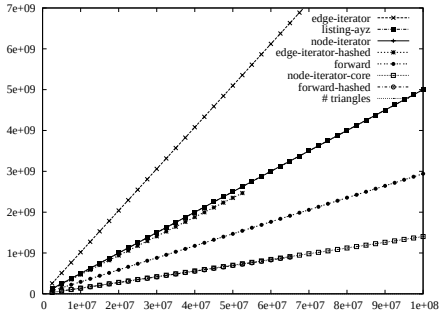
A very simple approach is to use matrix multiplication as a counting algorithm or to check for connecting edges between any three nodes as a listing algorithm. Traversing over all nodes and checking for existing edges between any pair of neighbors is part of the folklore. This algorithm, which we call *node-iterator* has running time $\mathcal{O}\left(nd_{\max}^2\right) \subset \mathcal{O}\left(n^3\right)$. The Algorithm *listing-ayz* is the

**Algorithm 1**: *forward*

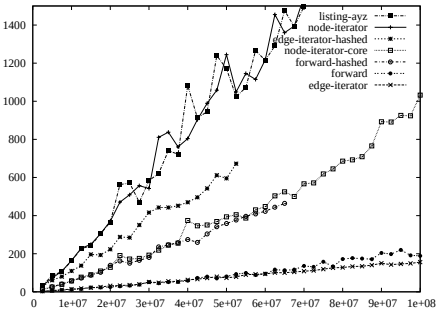
```
Input: ordered list (high degree first) of vertices (1, ..., n); Adjacencies Adj(v)
Data: Node Data: A(v);
for v ∈ V do
    A(v) ← ∅
for s ∈ (1, ..., n) do
    for t ∈ Adj(s) do
        if s < t then
            foreach v ∈ A(s) ∩ A(t) do
                output triangle {v, s, t} ;
            A(t) ⟵ A(t) ∪ {s};
```

listing version of the currently most efficient counting algorithm [AYZ97]. It has running time in $\mathcal{O}(m^{3/2})$. Algorithm *node-iterator-core* uses the concept of cores. It takes a node with currently minimal degree, computes its triangles in the same fashion as in *node-iterator* and then removes the node from the graph. The running time is in $\mathcal{O}(nc_{\max}^2)$, where $c(v)$ is the core number of node $v$. Since *node-iterator-core* is an improvement over *listing-ayz* the running time of *node-iterator-core* is also in $\mathcal{O}(m^{3/2})$.

Similar to *node-iterator* one can also traverse over all edges and compare the adjacency lists of the two incident nodes. This algorithm, which we call *edge-iterator* is equivalent to an algorithm introduced by Batagelj and Mrvar [BM01]. The running time without preprocessing is in $\mathcal{O}(md_{\max})$. It can actually be shown that *node-iterator* and *edge-iterator* are asymptotically equivalent, see [SW05] for details. Algorithm *forward* is an improvement of *edge-iterator*. The pseudo code is listed in Algorithm 1. It can be shown, that *forward* has running

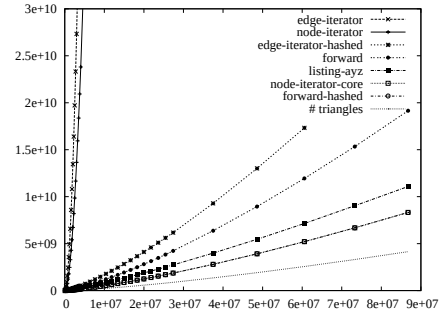



(a) Triangle Operations vs $m$

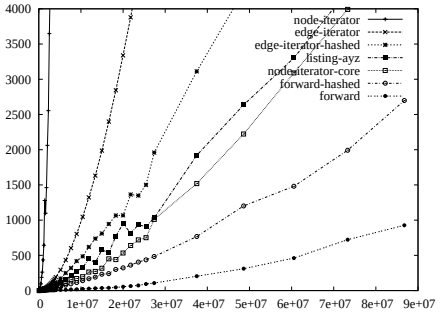



(b) Execution Times (sec.) vs $m$

**Fig. 2.** Generated $G_{n,m}$ Graphs




(a) Triangle Operations vs $m$




(b) Execution Times (sec.) vs $m$

**Fig. 3.** Generated Graphs with High Degree Nodes

time in $\mathcal{O}(m^{3/2})$. Both algorithms can be further improved with certain methods relying on hashing, see [SW05] for details.

## 4 Experiments

The algorithms are tested in two ways. On the one hand we list the execution time of the algorithms. Additionally, we give the *number of triangle operations*, which in essence captures the asymptotic running time of the algorithm without preprocessing. The algorithms are implemented in C++. The experiments were carried out on a 64-bit machine with a AMD Opteron Processors clocked at 2.20G-Hz. Figure 2 shows the results on generated $G_{n,m}$ graphs where $m$ edges are inserted randomly between $n$ nodes. These $G_{n,m}$ graphs tend to have no high degree nodes and to have a very low deviation from the average degree. However, this seems to be not true for many real networks [FFF99]. Therefore, Figure 3 shows results on modified $G_{n,m}$ graphs with $\mathcal{O}(\sqrt{n})$ high degree nodes.

## 5 Conclusion

The two known standard Algorithms *node-iterator* and *edge-iterator* are asymptoticly equivalent. However, the Algorithm *edge-iterator* can be implemented with a much lower constant overhead. It works very well for graphs where the degrees do not differ much from the average degree. If the degree distribution is skewed refined algorithms are required. The Algorithm *forward* shows to be the best compromise. It is asymptotically efficient and can be implemented to have a low constant factor with respect to execution time.

## References

AYZ97. Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

BM01. Vladimir Batagelj and Andrej Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23:237–243, 2001.

FFF99. Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of SIGCOMM'99*, 1999.

SW05. Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs. Technical report, Universität Karlsruhe, Fakultät für Informatik, 2005.

# Selecting the Roots of a Small System of Polynomial Equations by Tolerance Based Matching

H. Bekker*, E.P. Braad*, and B. Goldengorin**

*Department of Mathematics and Computing Science,
**Faculty of Economic Sciences,
University of Groningen, P.O.Box 800,
9700 AV Groningen, The Netherlands
bekker@cs.rug.nl, e.p.braad@wing.rug.nl, b.goldengorin@eco.rug.nl

**Abstract.** The roots of a system of two bivariate polynomial equations are calculated using a two-step method. First all $x$-roots and $y$-roots are determined independently. Then tolerance based weighted matching is used to form $(x, y)$-pairs that together form a minimum-error solution to the system.

**Keywords:** combinatorial optimization, tolerance based bipartite matching, solving polynomial equations.

## 1 Introduction

Consider a system of two polynomial equations

$$f(x, y) = 0 \quad g(x, y) = 0 \tag{1}$$

with symbolic constants and of low degree. By assigning numerical values to the constants we obtain a problem instance. Assuming that it is known that (1) has a finite number of solutions the conventional method to calculate the solutions is as follows. From (1) a univariate polynomial, say $p(x)$, is derived by eliminating $y$. For every problem instance the symbolic constants in $p(x) = 0$, $f(x, y) = 0$ and $g(x, y) = 0$ are replaced by numerical values and $p(x) = 0$ is solved numerically giving the roots $x_1, \ldots, x_n$. Subsequently, for every root $x_i$ the corresponding root $y_i$ has to be determined. To that end, $x_i$ is backsubstituted in $f(x, y) = 0$ and $g(x, y) = 0$, giving the univariate polynomial equations $f(x_i, y) = 0$ and $g(x_i, y) = 0$. Solving $f(x_i, y) = 0$ for $y$ gives the solutions $y_{f_1}, \ldots, y_{f_l}$, and solving $g(x_i, y) = 0$ for $y$ gives the solutions $y_{g_1}, \ldots, y_{g_m}$. The value $y_i$ occurring both in $y_{f_1}, \ldots, y_{f_m}$ and $y_{g_1}, \ldots, y_{g_m}$ is the desired value, i.e., the pair $x_i, y_i$ is a root of (1). During this process, a number of complications may occur:

1. The equation $f(x_i, y) = 0$ may be degenerate, i.e. may be $0 = 0$, or even worse, may be near degenerate within the noise margin. The case of exact

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 610–613, 2005.
© Springer-Verlag Berlin Heidelberg 2005

degeneracy is easily detected but it is not trivial to detect near degeneracy. In both cases every solution of the other equation, that is, of $g(x_i, y) = 0$ is a correct root. Analogously, $g(x_i, y) = 0$ may be degenerate, giving the same problems. As we know that (1) has a finite number of solutions the situation that $f(x_i, y) = 0$ and $g(x_i, y) = 0$ are both degenerate will not occur.

2. It is sometimes hard to select from $y_{f_1}, \ldots, y_{f_l}$ and $y_{g_1}, \ldots, y_{g_m}$ the collective value $y_i$ because, by numerical errors, the actual value of $y_i$ will be different in the two sets.
3. $p(x) = 0$ may have multiple roots, that is, the roots $x_1, \ldots, x_n$ may contain (near) identical values. Let us assume that there is a double root, given by the the identical values $x_i$ and $x_{i'}$. Then there will be two matching roots $y_j$ and $y_{j'}$, not necessarily with the same value. When $y_j$ is matched to $x_i$, in a later stage $y_{j'}$ should be matched to $x_{i'}$ and not to $x_i$.

When solving a problem of computational geometry we ran into these problems, first using our own multivariate polynomial solver and later using methods from packages. As a result a small but significant part of the roots, notably multiple roots, were missed or were completely wrong.

## 2 The CORS Method

To avoid the aforementioned complications we propose and test a two-step method, called the CORS method (Combinatorial Optimization Root Selection). First from (1) two univariate polynomials $p(x)$ and $q(y)$ are derived by eliminating $y$ and $x$ respectively. Whether this is done by calculating resultants or a Groebner basis is irrelevant. Now for every problem instance the symbolic constants in $p(x)$, $q(y)$, $f(x, y)$ and $g(x, y)$ are replaced by numerical values and the roots in $\mathbb{C}$ of $p(x)$ and $q(y)$ are calculated numerically. Both $p(x)$ and $q(y)$ have $n$ roots represented by $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, respectively. These roots are used to calculate $n^2$ weights, where $w_{i,j}$ is defined as

$$w(i, j) = \sqrt{(f(x_i, y_j))^2 + (g(x_i, y_j))^2}. \tag{2}$$

Subsequently a complete weighted bipartite graph $G(V, E)$ is constructed with $V = X \cup Y$ and $|X| = |Y| = n$. The nodes in $X$ consist of the values $x_1, \ldots, x_n$, and the nodes in $Y$ consist of the values $y_1, \ldots, y_n$. The arc between nodes $x_i$ and $y_j$ is assigned the weight $w(i, j)$. On $G$ the minimum-weight perfect matching $\pi_0$ is calculated. The $n$ arcs in $\pi_0$ represent the optimal solutions of (1). Here, optimal means that the sum of the errors is minimal. In the following this method of roots selection is called CORS1.

Instead of minimizing the sum of the errors it is more natural to minimize the maximum error. This is done as follows. All $n^2$ arcs and their weights are stored in a linear list $L$. Subsequently, $L$ is sorted in increasing order of weights. Now the weight in the first entry in $L$ is set to 1, and the weight of item $i$ is equal to the sum of the weights in previous items plus one, i.e. $weight[i] = (\sum_{j=1}^{i-1} weight[j]) + 1$. Thus the weights are $1, 2, 4, 8, 16, \ldots$. A new graph $G'$ is constructed, identical to $G$ but

with the new weights. Of $G'$ the minimum-weight perfect matching $\pi_0$ is calculated. The $n$ arcs in $\pi_0$ represent the optimal solutions of (1). Here, optimal means that the maximum error of $\pi_0$ is minimal. We call this method CORS2.

We here outline the three steps of a proof that this procedure minimizes the maximum weight when no identical weights in $G$ occur, without this assumption the proof is similar but more complex.

1: $\pi_0$ is unique because the total weight of $\pi_0$ can be constructed only in one way from the weights in $G'$.
2: In $\pi_0$ there is only one element $e_m$ with the maximum weight.
3: There is no matching of $G'$ without $e_m$, with a lower weight. q.e.d.

The weights in $G'$ become very large causing overflow on standard integer arithmetic. Therefore the infinite precision integer type should be used. The weights in $G'$ have the nice property that none of the weights can be constructed from other weights. This makes $G'$ very suitable for tolerance based matching.

## 3 Tolerance Based Matching

A *Feasible Assignment (matching, permutation)* (FA) $\pi$ on the bipartite graph $G'$ is a mapping $\pi$ of $X$ *onto* $Y$ with $w(\pi) = \sum_{(i,j)\in\pi} w(i,j) < \infty$ and the set of all FAs is $\Pi$. The Linear Assignment Problem (LAP) is the problem of finding a FA $\pi_0 \in \arg\min\{w(\pi) : \pi \in \Pi\}$, and all algorithms are based on shortest paths and the König-Egervary's theorem with $O(n^3)$ time complexity when applied to dense instances [1]. We sketch the idea of algorithms which are based only on tolerances for the Relaxed LAP (RLAP) without using the König-Egervary's theorem. A *Relaxed FA* (RFA) $\theta$ is defined on the same graph $G'$ as a mapping $\theta$ of $X$ *into* $Y$ with $w(\theta) = \sum_{(i,j)\in\theta} w(i,j) < \infty$. The RLAP is the problem of finding $\min\{w(\theta) : \theta \in \Theta\} = \sum_{i\in X} \min\{w(i,j) : j \in Y\} = w(\theta_0) \leq w(\pi_0)$ on the set of RFA $\Theta \supset \Pi$. A FA $\pi$ on $G'$ is a set of $n$ arcs $(i,j)$ such that the out-degree $od(i) = 1$ for all $i \in X$ and the in-degree $id(j) = 1$ for all $j \in Y$, and a RFA $\theta$ is a set of $n$ arcs $(i,j)$ with $od(i) = 1$ for all $i \in X$ and $\sum_{j\in Y} id(j) = n$. Note that $\theta$ is a FA if the $id(j) = 1$ for all $j \in Y$. For each fixed row $i$ of the matrix $W = ||w(i,j)||$ let $w[i,j_1(i)] \leq w[i,j_2(i)] \leq \ldots \leq w[i,j_n(i)]$ be the ordered set of entries in a non-decreasing order. We define the reduced matrix $W^r = ||w^r(i,j)||$ with $w^r(i,j) = w(i,j) - w[i,j_1(i)]$ for all $i \in X$ and $j \in Y$. The *tolerance problem* for the RLAP is the problem of finding for each arc $(i,j) \in X \times Y$ the maximum decrease $l(i,j)$ and the maximum increase $u(i,j)$ of the arc weight $w(i,j)$ preserving the optimality of $\theta_0$ under the assumption that the weights of all other arcs remain unchanged. Now for an arc $[i,j_1(i)] \in \theta_0$ the *upper tolerance* $u[i,j_1(i)] = w[i,j_2(i)]$, and the *lower tolerance* $l[i,j_1(i)] = \infty$. Similarly, for an arc $(i,j) \notin \theta_0$ the *lower tolerance* $l(i,j) = w^r(i,j)$ and the *upper tolerance* $u(i,j) = \infty$. Let us show that the *bottleneck tolerance* $b(\theta_0) = \max\{u(\theta_0), l(\theta_0)\}$ is a *tightness measure* between known value of $w(\theta_0)$ and the unknown value of $w(\pi_0)$. For a fixed $\theta_0$ we partition the set $Y$ into three subsets of vertices: the *unassigned set* $Y_0 = \{j \in Y : id(j) = 0\}$, *assigned set* $Y_1 = \{j \in Y : id(j) = 1\}$,

and *overassigned set* $Y_2 = \{j \in Y : id(j) > 1\}$. For each fixed $j \in Y_2$ we order the corresponding upper tolerances in non-decreasing order $u[i_1(j), j] \leq u[i_2(j), j] \leq ... \leq u[i_{p_j}(j), j]$ and compute $u(\theta_0) = \sum_{j \in Y_2} u(j)$ with $u(j) = \sum_{t=1}^{p_j - 1} u[i_t(j), j]$. Similarly, for each fixed $j \in V_0$, $l[i(j), j] = \min\{l(i, j) : i \in X\}$, $l[Y_0(j)] = \max\{l[i(t), t] : t \in Y_0(j)\}$ and $l(\theta_0) = \sum_{j=1}^{k} l[Y_0(j)]$ with $Y_0(j) = \{t \in Y_0 : i(t) = i(j)\}$. Here, $Y_0(1), \ldots, Y_0(k)$ is a partition of $Y_0$. Further we treat each $\pi$, and each $\theta$ as the sets of corresponding arcs such that $|\pi| = |\theta| = n$. Note that if either $Y_0 = \emptyset$ or $Y_2 = \emptyset$ then $|Y_1| = n$ and $\theta_0$ is a FA. Hence, for each $\theta_0 \notin \mathit{\Pi}$ we may use the number of unassigned columns $|Y_0| = \sum_{j \in Y_2} |id(j) - 1|$ in the reduced matrix $W^r$ as a *measure of structural infeasibility* of $\theta_0$ to the LAP, for which the bottleneck tolerance $b(\theta_0) \leq w(\pi_0) - w(\theta_0)$. Our algorithm for solving the LAP recursively fixes the arc $(i, j) \in \theta_0$ with the largest tolerance and replaces all other arcs from $Y_2$ by the arcs representing the tolerances ordered in a non-increasing order, regardless of either upper or lower tolerance will be the next tolerance induced by that order. Therefore, the first obtained $\theta \in \mathit{\Pi}$ is $\theta = \pi_0$, and hence the time complexity of LAP for CORS2 is $O(n^2)$.

## 4 Implementation, Tests and Results

**Implementation** We tested CORS on our computational geometry problem. Of this class of problems it is known that every instance has eight solutions. The univariate polynomials $p(u)$ and $q(w)$ are derived with MAPLE. The numerical calculations are implemented in C++ in double precision. Laguerre's method [2] is used to compute the roots of the polynomials $p(u)$ and $q(w)$. The LEDA [3] implementation of the minimum weight bipartite matching algorithm is used.
**Tests** We tested the CORS1 and CORS2 method. Every problem instance is solved in two ways: with the CORS method and with SYNAPS, a C++ package for solving polynomial equations [4]. We solved $10^4$ problem instances with CORS and SYNAPS, and $\approx 400$ with MAPLE. The latter problem instances were solved correctly by CORS and were missed by SYNAPS, i.e. we use MAPLE to decide whether CORS or SYNAPS gave the correct result.
**Results** In general the results of CORS1 and CORS2 are identical. In the tests approximately 2.4% of the solutions is missed by SYNAPS and are found by CORS. No solutions were missed by CORS. The average error of the solutions found by SYNAPS is $1.3\ 10^{-10}$ and of CORS $6.5\ 10^{-11}$. Running $10^5$ problem instances with CORS takes 14 *sec.* and with SYNAPS 475 *sec.*

## References

1. Burkard, R. E. *Selected topics on assignment problems.* Discrete Applied Mathematics 123, 257–302, 2002.
2. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes in C++.* Cambr. Univ. Press, New York.
3. K. Melhorn, Näher, S. *LEDA A Platform for Combinatorial and Geometric Computing.* Cambridge University press,Cambridge. 1999
4. Synaps. Available at: http://www.inria.fr/galaad/logiciels/synaps/inex.html

# Developing Novel Statistical Bandwidths for Communication Networks with Incomplete Information

Janos Levendovszky and Csego Orosz

Budapest University of Technology and Economics,
Department of Telecommunications,
Magyar tudosok korutja 2, H-1117 Budapest, Hungary
{levendov, oroszcs}@hit.bme.hu

**Abstract.** In this paper, the concept of statistical bandwidth of multi-access systems are studied and extended to the case of unknown statistical descriptors. The results can improve the statistical characterization of the tail distribution of aggregated load presented to a multi-access system which is traditionally based on the logarithmic moment generation function (LMGF)[1]. In the paper, an extended moment generating function is introduced for calculating the statistical bandwidth and as a result a novel admission algorithm is presented. To further maximize the admitted load into the multi-access system the free parameter of the extended statistical bandwidth is optimized based on the geometrical optimization of polygonal surfaces. In this way, the system utilization can be near-optimal.

## 1 Introduction

Let us assume that the following quantities are given: (i) a number of traffic classes (e.g. video, ftp, voice ... etc.) denoted by $i = 1, ..., M$; (ii) the random traffic emitted by source $j$ from class $i$ is denoted by $X_j^{(i)}$ (sources form the same class are assumed to be homogenous); (iii) traffic class $i$ is characterized by traffic descriptors $\mathbf{r}_i = (r_{i_1}, ..., r_{i_V})$ (e.g. in the case of On/Off sources $\mathbf{r}_i = (m_i, h_i)$, where $m_i$ refers to the average rate, whereas $h_i$ stands for the peak rate, respectively); (iv) $\mathbf{r}_i, i = 1, ..., M$ are supposed to be random variables due to imperfect measurements and the corresponding p.d.f.-s are denoted by $p_i(\mathbf{x}), i = 1, ..., M$; (v) the traffic state of the network is described by a traffic state vector $\mathbf{n} = (n_1, ..., n_M)$, the $i$th component of which indicates the number of users being present from class $i$; (vi) the network (or access point) capacity is denoted by $C$; (vii) QoS is measured by the cell loss probability (zero buffer approximation) meaning that

$$P\left(\sum_{i=1}^{M}\sum_{j=1}^{n_i} X_j^{(i)} > C\right) < e^{-\gamma}, \tag{1}$$

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 614–617, 2005.
© Springer-Verlag Berlin Heidelberg 2005

where $\gamma$ is the QoS parameter. Our concern is to evaluate equation (1) when traffic descriptors are only given by their p.d.f and possibly reduce this expression into $\sum_{i=1}^{M} n_i\beta_i < \Psi(C,\gamma)$, where $\beta_i$ is referred to as statistical bandwidth of class $i$ (due to the additive rule) and $\Psi$ is an appropriate function.

Traditionally, statistical bandwidth has been calculated on the basis of Chernoff inequality assuming known traffic descriptors

$$P\left(\sum_{i=1}^{M}\sum_{j=1}^{n_i} X_j^{(i)} > C\right) < e^{\sum_{i=1}^{M} n_i\mu_i(s^*)-s^*C}, \tag{2}$$

where $\mu_i(s) := \log\left(E\left(e^{sX^{(i)}}\right)\right)$ and $s^* : \min_s \sum_{i=1}^{M} n_i\mu_i(s) - sC$. In this way inequality (1) can easily be evaluated by checking the equivalent inequality $\sum_{i=1}^{M} n_i\mu_i(s^*) < s^*C - \gamma$. This form defines the statistical bandwidth as $\mu_i(s)$, since it follows the desired additive rule. One must note that this expression defines a dichotomy over the traffic state space $\mathcal{N}$ expanded by the traffic vectors. Unfortunately, $\mu_i(s)$ can only be calculated if the source is fully characterized by its traffic descriptors, i.e. in the case of On/Off sources when $\mathbf{r}_i = (m_i, h_i)$, $P(X_i = 0) = 1 - \frac{m_i}{h_i}$ and $P(X_i = h_i) = \frac{m_i}{h_i}$, the statistical bandwidth becomes $\mu_i(s) = \log\left(1 - \frac{m_i}{h_i} + \frac{m_i}{h_i}e^{sh_i}\right)$. In the case of unknown $\mathbf{r}_i, i = 1, ..., M$ are not known then new methods must be developed for CAC.

## 2 Extension of Statistical Bandwidth

In this section, we embark on extending the statistical bandwidth when $\mathbf{r}_i$, $i = 1, ..., M$ are not given by their exact values but assumed to be random variables and only the family of p.d.f.-s, $p_i(\mathbf{z}), i = 1, ..., M$ is given. This extension is based on the following theorem:

**Theorem 1.** *The function*

$$\beta_i(s) := \log\left(\int_{z_1,...,z_V} p_i(z_1, ..., z_V)e^{\mu_i(z_1,...,z_V,s)}dz_1, .., dz_V\right) \tag{3}$$

*is additive in the sense that calls can be accepted by checking if*

$$\sum_{i=1}^{M} n_i\beta_i(s) < sC - \gamma.$$

The proof drops out form the conditional form of the Chernoff inequality, due to the limited we do not detail the steps.

As a result we call $\beta_i(s), i = 1, ..., M$ as *Generalized Statistical Bandwidth* (GSB). One must note that $\beta_i(s)$ does not need the exact values of the traffic descriptors, but only their p.d.f. is necessary. As was mentioned earlier, the

modified Chernoff-bound is valid for each positive $s$. Therefore, one can select the tightest upper bound given as follows:

$$P\left(\sum_{i=1}^{M}\sum_{j=1}^{n_i} X_j^{(i)} > C\right) < e^{\sum_{i=1}^{M} n_i\beta_i(s^*)-s^*C}, \text{ where } s^* : \min_s . \sum_{i=1}^{M} n_i\beta_i(s^*) - s^*C \tag{4}$$

Consequently, the CAC algorithm in the case of unknown traffic descriptors can be performed as follows: Given: $\mathbf{n}$, $C$ and $\gamma$
(i) Calculate $\beta_i(s) := \log\left(\int_{\mathbf{z}^{(i)}} e^{\mu_i(s)} p_i\left(\mathbf{z}^{(i)}\right) d\mathbf{z}^{(i)}\right)$; (ii) Calculate $s^* : \min_s \sum_{i=1}^{M} n_i\beta_i(s^*) - s^*C$; (iii) Check if $\sum_{i=1}^{M} n_i\beta_(s^*) < s^*C - \gamma$; (iv) If YES then accept traffic vector $\mathbf{n}$, otherwise refuse it.

One can see that this algorithm is rather tiresome in the sense that parameter $s$ has to be re-optimized for each entering traffic vector $\mathbf{n}$. Therefore, it is not suitable for real-time CAC algorithm. To get rid of this computational burden, it is easy to see that with a constant $\tilde{s}$ the formula $\sum_{i=1}^{M} n_i\beta_i(\tilde{s}) < \tilde{s}C - \gamma$ defines a set-separation where the separation surface is a linear hyperplane. Therefore, for a given $\tilde{s}$ the admitted traffic volume can be calculated as counting the number of traffic vectors $\mathbf{n}$ for which the formula holds. This volume can be expressed as $Vol(\tilde{s}) := \frac{(\tilde{s}C-\gamma)^M}{M!\prod_{i=1}^{M}\beta_i(\tilde{s})}$. Hence, one can use a fixed $\tilde{s}$ which maximizes the volume defined above, setting $s_{opt}$ : $s_{opt}$ : $\max_{\tilde{s}} \frac{(\tilde{s}C-\gamma)^M}{M!\prod_{i=1}^{M}\beta_i(\tilde{s})}$. Since this $s_{opt}$ does not depend on the incoming traffic vector $\mathbf{n}$, it can be calculated once for all, demanding only off-line complexity.

## 3 Numerical Results

The aim of this section is to evaluate the traffic volumes accepted by SB-CAC (**ex.SB-CAC**: SB-CAC with expected values ; **av.SB-CAC**: SB-CAC with random selection and averaging) and **GSB-CAC** methods, respectively. The comparison of these volumes will characterize the loss of traffic due to the fact of uncertain link descriptors given only by their p.d.f.-s. The new statistical bandwidths, $\beta_i(s)$, $i = 1, ..., M$ are calculated when the mean rates are considered to be random variables subject to Gaussian and uniform p.d.f.

The second column of Table 1 shows the $s_{opt}$ parameters optimized off-line and calculated by maximizing the traffic volume. All calculations were made with the following initial parameters: (i) the number of traffic classes: $M = 2$; (ii) the network capacity is chosen to be: $C = 10000$ kbps; (iii) the QoS parameter is chosen to be: $\gamma = 10$, (iv) the deviation of the Gaussian p.d.f. is chosen to be: $\sigma = 10$; (v) the traffic rates: $[m_i, h_i]^{TC1} = [32, 64]$kbps ; $[m_i, h_i]^{TC2} = [96, 128]$kbps. Using the $s_{opt}$ parameters for calculating the volume of the accepted users, the results can easily be demonstrated by counting the accepted traffic vectors $\mathbf{n}$. Table 1 shows the calculated accepted volume of the user vectors. As one can see, there is a loss of accepted traffic volume due to the uncertain information

**Table 1.** The calculated $s_{opt}$ parameters, and the accepted volume in the cases of different p.d.f.s and CAC methods

| *volume* | $s_{opt}$ | ex.SB-CAC | av.SB-CAC | GSB-CAC |
|---|---|---|---|---|
| Gaussian pdf | 0.0079 | 10149 | 5626 | 13496 |
| Uniform pdf | 0.0088 | 10240 | 5030 | 13252 |

(unknown traffic descriptors). The larger the variance of the p.d.f. of the traffic descriptor (i.e. the larger the amount of uncertainty), the larger these losses become. The loss in traffic volume is the trade-off for only accepting those traffic configurations which do not put the QoS in jeopardy.

These numerical results demonstrate the advantage of using the GSB concept in admission control.

## 4 Conclusion

In this paper the concept of statistical bandwidth was extended to the case of unknown traffic descriptors. A novel statistical bandwidth was derived by using the LMGF of the traffic descriptors, furthermore the tail estimator was optimized to admit the maximal load volume without violating the QoS requirements. As the numerical results have demonstrated the new method yield minimal loss of load volume despite the incompleteness in characterization of the traffic descriptors.

## Acknowledgement

The research carried out here was supported by the the Laboratory of Analogic Computing, Hungarian Academy of Sciences and by the project GOA/98/06 of Research Fund, Katholieke Universiteit Leuven.

## References

1. F. P. Kelly: "Notes on effective bandwidths", *Stochastic Networks: Theory and Applications*, Editors F.P. Kelly, S. Zachary and I.B. Ziedins), *Royal Statistical Society Lecture Notes, Series, 4. , Oxford University Press,* 1996. 141-168.
2. R.J. Gibbens, F.P. Kelly , P.B. Key.: "A decesion theoretic approach to call admission control in ATM networks", *IEEE Journal on Selected Areas in Communication*, Vol.13, No. 6., Agust 1995.
3. J. Hui: "Switching and traffic theory for integrated broadband networks", *Kluwer Academic Publisher*, 1990.
4. Levendovszky J., Vegso Cs., van der Meulen, E.C.: "Nonparametric decision algorithms for CAC in ATM networks", *Performance Evaluation - Elsevier*, Vol.41, pp. 133-147, 2000.

# Dynamic Quality of Service Support in Virtual Private Networks

Yuxiao Jia[1], Dimitrios Makrakis[2], Nicolas D. Georganas[2], and Dan Ionescu[2]

[1] Nortel Networks, Ottawa, Ont., Canada
[2] School of Information Technology & Engineering, University of Ottawa,
161 Louis Pasteur St., P.O. Box 450 Stn A, Ottawa, Ont., K1N 6N5, Canada
Tel: 1-613-5625800 ext. {6202, 6225, 6209}
jenniej@nortelnetworks.com,
{dimitris, georgana, ionescu}@site.uottawa.ca

**Abstract.** This paper presents a framework for the provision of dynamic Quality of Service (QoS) support in Virtual Private Networks (VPNs) running over an MPLS-enabled public network infrastructure. A Dynamic Bandwidth Allocation scheme that includes traffic estimators and the development of a resource reservation algorithm, capable of modifying the resource allocation in real-time, is used. Three traffic estimation algorithms are implemented and tested. This system can automatically adjust to the bandwidth size of a VPN tunnel. The technique is beneficial to Internet Service Providers (ISPs) and corporate users alike. The superior resource management achieved through the examined approach can produce lower costs for the users and higher profits to the ISP. Implementation and experimental evaluation of the technique, using our MPLS and Diffserv enabled Linux test-bed, confirmed its ability to provide better resource utilization.

## 1 Introduction

VPN services have been offered in various forms over an extended period of time and typically have been implemented at the data link layer using Frame Relay and Asynchronous Transfer Mode (ATM) networking technologies. VPN services based on IP/MPLS are quickly gaining public interest and market acceptance. Most work on VPNs has mainly dealt with the security issue. However, with the advancement of technology and the introduction of sophisticated applications, users don't only demand security, but expect provision of QoS guarantees, in several cases compared to those provided by leased line services. Thus, incorporation of QoS support in the VPN technology, in a resource efficient manner, becomes increasingly important.

In this paper we describe and evaluate a dynamic QoS supporting technique, suitable for VPNs. It was implemented in an MPLS and DiffServ enabled experimental network, incorporating Traffic Engineering and RSVP-TE for the establishment of connections. Three traffic estimators and the CBQ, modified in order to be able to support resource allocation in a dynamic fashion, were implemented.

S.E. Nikoletseas (Ed.): WEA 2005, LNCS 3503, pp. 618 – 621, 2005.
© Springer-Verlag Berlin Heidelberg 2005

## 2 Traffic Estimators

We assume that the measurements comprise samples gathered at regularly spaced instants during a measurement window $T_{meas}$. The measurements are used to estimate the bandwidth for the traffic flow over some reservation window *Y*. The parameters are: *X*: inter-sample interval; *Y*: reservation window; W: number of samples taken within the measurement window $T_{meas}$; *N*: number of samples taken within the reservation window *Y*; $R_i$: average sampling rate.

The traffic estimation algorithms we implemented and evaluated are the Maximum Estimator (ME) [1], Gaussian Estimator (GE) [2] and α-stable Estimator (ASE) [5]. The following relations between the above defined parameters hold: $Y = N*X$ ($N$ = 1,2, …...…; N is positive integer).

The expressions of the renegotiated bandwidth $R_{ren}$ for the three estimators are given in equation 1. In the case of ME (equation 1(a)), the renegotiated rate $R_{ren}$ is the maximum of the rate samples collected during the measurement window $T_{meas}$, i.e.:

$$R_{ren} = \max\{R_i\} \textbf{ (a)} \qquad R_{ren} = m + a\sqrt{v} \textbf{ (b)} \qquad \hat{W}_j = m + K\sigma_W \textbf{ (c)} \qquad (1)$$

The GE is based on the assumption that aggregated traffic can be characterized by the Gaussian distribution. The renegotiated rate $R_{ren}$ for GE is given in equation 1(b), where $m$ and $v$ are respectively the mean and variance of the rates sampled during the measurement window, and $a$ is a scale factor that controls the extent to which the negotiated rate accommodates the variability of the samples.

The ASE estimator assumes that the statistical behavior of aggregate traffic streams is statistically described through α-stable long-range dependent stochastic processes [3]. This is a valid assumption, since the authors of [3] proved that such distributions describe more accurately aggregate traffic passing through modern networks as compared to earlier models. In [4], traffic estimators for α-stable long-range dependent traffic have been proposed. In [5], the concept of probabilistic envelope processes was extended to the α-stable case, providing us with a very practical and simple to implement traffic estimator. The bandwidth demand imposed by the traffic is represented by an envelope process. The following envelope process $\hat{W}_j$ is used (see equation 1(c)), where $m$ is the mean of the number of arrivals per unit time, and $c_W$ is the scale parameter. It is similar to the variance of the Gaussian distribution. How to compute $c_W$ can be found in [3,4,5]. $K$ is determined by the overshoot probability $\varepsilon$ [5] and index of stability $\alpha$ [3].

## 3 Experimental Set-Up and Performance Evaluation

Figure 1 shows the architecture of our test-bed. The source customer network includes an IP router used to mark IP packets (setting up the DSCP value) by using *iptables*. The MPLS backbone network is made up of two edge LERs and one core LSR, which establish one or more Diffserv enabled LSPs. All routers run under the Linux *Redhat*™ system. At the core router, CBQ has been implemented for the allocation of bandwidth. The CBQ has been modified in order to be able to change the allocation in

real-time. The speed of the segment, to which the core LSR's output is connected, is 10 Mbps, becoming the source of congestion. The other segments have 100 Mbps speed.

In our experiments, we used our α-stable Long Range Dependent (LRD) traffic generator, set to produce traffic streams with index of stability: i) α=1.95, ii) α=1.60, and a Short Range Dependent (SRD) UDP Poisson traffic generator. The QoS parameters we measure are the packet loss rate, average delay, delay jitter and the ratio of average reserved bandwidth over the average traffic volume. The parameters we examine in terms of their impact on the performance are: i) buffer size at the core router *B*; ii) sample interval *X*; iii) measurement window $T_{meas}$; iv) resource reservation window *Y*. For each parameter, we plot: a) the amount of packet losses at the core router; b) the average forwarding delay at the core router; c) the average delay jitter (IPDV) at the core router; d) the ratio of average reservation over the average traffic volume.

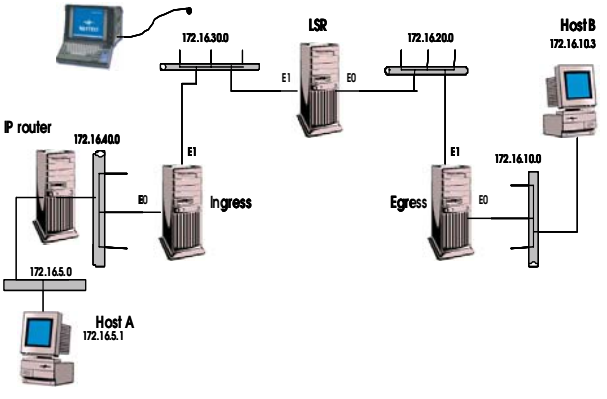



**Fig. 1.** Test-bed's architecture

Due to the limited space, we only show the performance as function of $T_{meas}$ for α-stable traffic with α=1.95 when the GE and ASE are used. The results are presented in figure 2. We consider the cases where the traffic process exceeds its envelope by 0.1, 0.01 and 0.001. Please note that differently from the Gaussian case, the α-stable estimator is able to provide better performance when the QoS requirement becomes more stringent. Dynamic bandwidth allocation jointly with the ASE provides more reliable performance, especially when the traffic exhibits high variability.

## 4 Conclusions

This paper presents a framework for QoS provisioning in VPNs by combining dynamic Bandwidth Allocation and traffic monitoring. Three traffic estimation algorithms were assessed by developing the proposed scheme on an experimental testbed with DiffServ MPLS capabilities.

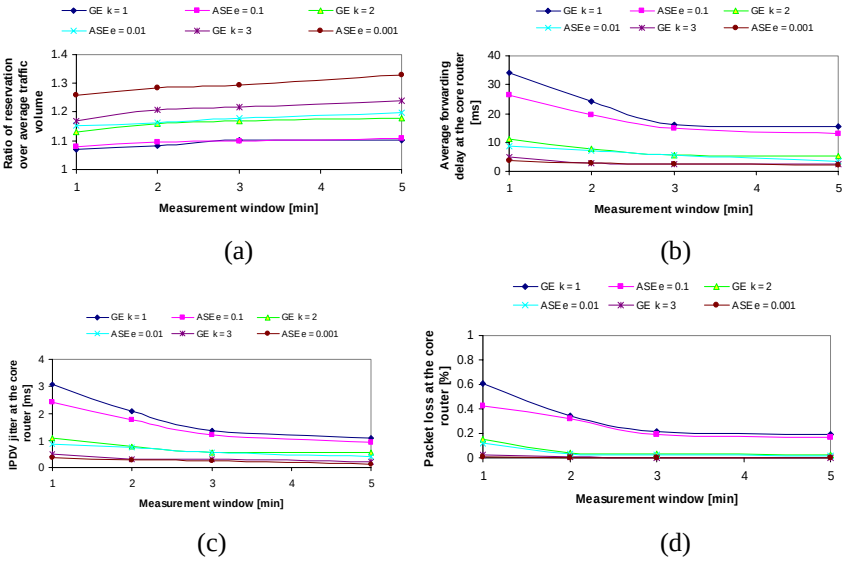



**Fig. 2. (a)** Ratio of reservation over average traffic volume; **(b)** average forwarding delay [ms] measured at the core router ; **(c)** IPDV jitter [ms] measured at the core router; **(d)** packet loss [%] measured at the core router, vs. $T_{meas}$, for GE and ASE when stable traffic with $\alpha$=1.95 is applied

## References


1. Cisco White Paper, *"Cisco MPLS Auto-Bandwidth Allocator for MPLS Traffic Engineering: A Unique New Feature of Cisco IOS Software"*, http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/mpatb_wp.htm
2. N .G. Duffield, P. Goyal, A. Greenberg, P. Mishra, **"***A Flexible Model for Resource Management in Virtual Private Networks"*, ACM SIGCOMM' 99, Oct. 1999, Cambridge, MA, USA.
3. J. R. Gallardo, D. Makrakis, L. Orozco-Barbosa, *"Use of Alpha-Stable Self-Similar Stochastic Process for Modeling Traffic in Broadband Networks"*, Performance Evaluation, 40 (1-3), pp. 71-98, 2000.
4. J. R. Gallardo, D. Makrakis, M. Angulo, "*Dynamic Resource Management Considering the Real Behavior of Aggregate Traffic*", IEEE Trans. On Multimedia, Vol. 3, No. 2, pp. 177-185, June 2001.
5. M. L. Guerrero, L. Orozco-Barbosa, D. Makrakis, *"Probabilistic Envelope Processes for alpha-Stable Self-Similar Traffic Models and their Application to Resource Provisioning"*, to appear in the Performance Evaluation Journal.

# Author Index


Albuquerque, Paul 341
Andreou, M.I. 302
Asgeirsson, Eyjolfur 545
Ayala-Rincon, Mauricio 464

Bachoore, Emgad H. 216
Bader, David A. 16
Bar-Noy, Amotz 139
Bast, Holger 67
Becker, Bernd 452
Behle, Markus 452
Bekker, H. 610
Bleischwitz, Yvonne 228
Bodlaender, Hans L. 101, 216
Botelho, Fabiano C. 488
Boughaci, Dalila 501
Boukerche, Azzedine 403, 464
Braad, E.P. 610

Cantone, Domenico 265, 428
Christensen, Jacob 139
claffy, kc 113
Cristofaro, Salvatore 428

de Andrade, Marcos R.Q. 558
de Andrade, Paulo M.F. 558
de Melo, Alba Cristina
Magalhaes Alves 403, 464
de Souza, Cid C. 328
De Wachter, Bram 177
Dimitropoulos, Xenofontas 113
Drias, Habiba 501

Eisenbrand, Friedrich 452
Elmasry, Amr 597

Fahle, Torsten 89
Faro, Simone 428
Fernandes, Eraldo R. 4
Ferro, Alfredo 265
Festa, Paola 367
Flammini, Michele 22

Gambin, Anna 534
Genon, Alexandre 177
Georgakopoulos, George F. 570
Georganas, Nicolas D. 618
Geraci, Filippo 580
Giugno, Rosalba 265
Goldberg, M. 513
Goldengorin, B. 610
Gomułkiewicz, Marcin 415
Grossi, Roberto 580

Hammad, Abdelrahman 597
Hassin, Refael 44
Heinrich-Litan, Laura 55
Hollinger, D. 513
Huffaker, Bradley 113
Hüffner, Falk 240
Hyyrö, Heikki 380

Ionescu, Dan 618

Jia, Yuxiao 618

Kamphans, Tom 593
Kaporis, A.C. 77
Kchikech, Mustapha 165
Kim, Dong Kyue 315
Kim, Ji Eun 315
Kimbrel, Tracy 391
Kirousis, L.M. 77
Kliewer, Georg 228
Koch, Jeferson 403
Kochol, Martin 602
Kohayakawa, Yoshiharu 488
Köhler, Ekkehard 126
Koster, Arie M.C.A. 101
Krioukov, Dmitri 113
Krivoňáková, Naďa 602
Kutyłowski, Mirosław 415

Ladner, Richard E. 139
Langetepe, Elmar 593
Leone, Pierre 341
Levendovszky, Janos 614
Liebchen, Christian 354
Lo Presti, Giuseppe 265

Lu, Qiang 152
Lübbecke, Marco E. 55

Magdon-Ismail, M. 513
Makrakis, Dimitrios 618
Martins, Simone L. 558
Massart, Thierry 177
Mazza, Christian 341
Mehlhorn, Kurt 32
Michail, Dimitrios 32
Möhring, Rolf H. 126, 189
Moura, Arnaldo V. 328

Na, Joong Chae 315
Navarra, Alfredo 22

Orogz, Csego 614
Orponen, Pekka 524
Ossamy, Rodrigue 290

Panagopoulou, Panagiota N. 203
Papadimitriou, Christos H. 1
Papadopoulou, V.G. 302
Pardalos, Panos M. 367
Park, Kunsoo 315
Pereira, Romulo A. 328
Perennes, Stephane 22
Pérez, Xavier 253
Pinzon, Yoan 380
Pitsoulis, Leonidas S. 367
Plastino, Alexandre 558
Politopoulou, E.I. 77
Pulvirenti, Alfredo 265

Resende, Mauricio G.C. 367
Ribeiro, Celso C. 4
Riley, George 113
Rolim, Jose 341

Santana, Thomas M. 464
Sawitzki, Daniel 277
Schaeffer, Satu Elisa 524
Schank, Thomas 606
Schilling, Heiko 126, 189
Schütz, Birk 189
Schwartz, Justus 476
Segev, Danny 44
Shinohara, Ayumi 380
Smejová, Silvia 602
Spirakis, Paul G. 77, 203, 302
Steger, Angelika 476
Stein, Cliff 545
Steinder, Malgorzata 391
Sviridenko, Maxim 391

Tamir, Tami 139
Tantawi, Asser 391
Theodorides, B. 302
Tiemann, Karsten 89
Togni, Olivier 165

Vierhaus, Heinrich Theodor 415

Wagner, Dorothea 189, 606
Weber, Ingmar 67
Weißl, Andreas 476
Weinard, Maik 440
Willhalm, Thomas 189
Wimmer, Ralf 452
Wlaź, Paweł 415
Wójtowicz, Damian 534
Wolle, Thomas 101

Xeros, A. 302

Zhang, Hu 152
Ziviani, Nivio 488