Potential Function Methods for Approximately Solving Linear Programming Problems

INTERNATIONAL SERIES IN
OPERATIONS RESEARCH & MANAGEMENT SCIENCE
Frederick S. Hillier, Series Editor Stanford University

Miettinen, K. M. / *NONLINEAR MULTIOBJECTIVE OPTIMIZATION*
Chao, H. & Huntington, H. G. / *DESIGNING COMPETITIVE ELECTRICITY MARKETS*
Weglarz, J. / *PROJECT SCHEDULING: Recent Models, Algorithms & Applications*
Sahin, I. & Polatoglu, H. / *QUALITY, WARRANTY AND PREVENTIVE MAINTENANCE*
Tavares, L. V. / *ADVANCED MODELS FOR PROJECT MANAGEMENT*
Tayur, S., Ganeshan, R. & Magazine, M. / *QUANTITATIVE MODELING FOR SUPPLY CHAIN MANAGEMENT*
Weyant, J./ *ENERGY AND ENVIRONMENTAL POLICY MODELING*
Shanthikumar, J.G. & Sumita, U./*APPLIED PROBABILITY AND STOCHASTIC PROCESSES*
Liu, B. & Esogbue, A.O. / *DECISION CRITERIA AND OPTIMAL INVENTORY PROCESSES*
Gal, T., Stewart, T.J., Hanne, T./ *MULTICRITERIA DECISION MAKING: Advances in MCDM Models, Algorithms, Theory, and Applications*
Fox, B. L./ *STRATEGIES FOR QUASI-MONTE CARLO*
Hall, R.W. / *HANDBOOK OF TRANSPORTATION SCIENCE*
Grassman, W.K./ *COMPUTATIONAL PROBABILITY*
Pomerol, J-C. & Barba-Romero, S. / *MULTICRITERION DECISION IN MANAGEMENT*
Axsäter, S. / *INVENTORY CONTROL*
Wolkowicz, H., Saigal, R., Vandenberghe, L./ *HANDBOOK OF SEMI-DEFINITE PROGRAMMING: Theory, Algorithms, and Applications*
Hobbs, B. F. & Meier, P. / *ENERGY DECISIONS AND THE ENVIRONMENT: A Guide to the Use of Multicriteria Methods*
Dar-El, E./ *HUMAN LEARNING: From Learning Curves to Learning Organizations*
Armstrong, J. S./ *PRINCIPLES OF FORECASTING: A Handbook for Researchers and Practitioners*
Balsamo, S., Personé, V., Onvural, R./ *ANALYSIS OF QUEUE1NG NETWORKS WITH BLOCKING*
Bouyssou, D. et al/ *EVALUATION AND DECISION MODELS: A Critical Perspective*
Hanne, T./ *INTELLIGENT STRATEGIES FOR META MULTIPLE CRITERIA DECISION MAKING*
Saaty, T. & Vargas, L./ *MODELS, METHODS, CONCEPTS & APPLICATIONS OF THE ANALYTIC HIERARCHY PROCESS*
Chatterjee, K. & Samuelson, W./ *GAME THEORY AND BUSINESS APPLICATIONS*
Hobbs, B. et al/ *THE NEXT GENERATION OF ELECTRIC POWER UNIT COMMITMENT MODELS*
Vanderbei, R.J./ *LINEAR PROGRAMMING: Foundations and Extensions, 2nd Ed.*
Kimms, A./ *MATHEMATICAL PROGRAMMING AND FINANCIAL OBJECTIVES FOR SCHEDULING PROJECTS*
Baptiste, P., Le Pape, C. & Nuijten, W./ *CONSTRAINT-BASED SCHEDULING*
Feinberg, E. & Shwartz, A./ *HANDBOOK OF MARKOV DECISION PROCESSES: Methods and Applications*
Ramík, J. & Vlach, M. / *GENERALIZED CONCAVITY IN FUZZY OPTIMIZATION AND DECISION ANALYSIS*
Song, J. & Yao, D. / *SUPPLY CHAIN STRUCTURES: Coordination, Information and Optimization*
Kozan, E. & Ohuchi, A./ *OPERATIONS RESEARCH/ MANAGEMENT SCIENCE AT WORK*
Bouyssou et al/ *AIDING DECISIONS WITH MULTIPLE CRITERIA: Essays in Honor of Bernard Roy*
Cox, Louis Anthony, Jr./ *RISK ANALYSIS: Foundations, Models and Methods*
Dror, M., L'Ecuyer, P. & Szidarovszky, F./ *MODELING UNCERTAINTY: An Examination of Stochastic Theory, Methods, and Applications*
Dokuchaev, N./ *DYNAMIC PORTFOLIO STRATEGIES: Quantitative Methods and Empirical Rules for Incomplete Information*
Sarker, R., Mohammadian, M. & Yao, X./ *EVOLUTIONARY OPTIMIZATION*
Demeulemeester, R. & Herroelen, W./ *PROJECT SCHEDULING: A Research Handbook*
Gazis, D.C. / *TRAFFIC THEORY*
Zhu/ *QUANTITATIVE MODELS FOR PERFORMANCE EVALUATION AND BENCHMARKING*
Ehrgott & Gandibleux/ *MULTIPLE CRITERIA OPTIMIZATION: State of the Art Annotated Bibliographical Surveys*
Bienstock, D. / *Potential Function Methods for Approx. Solving Linear Programming Problems*

POTENTIAL FUNCTION METHODS FOR APPROXIMATELY SOLVING LINEAR PROGRAMMING PROBLEMS: THEORY AND PRACTICE

DANIEL BIENSTOCK
Department of IEOR
Columbia University
New York

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-47626-6
Print ISBN: 1-4020-7173-6

Visit Kluwer Online at: http://kluweronline.com
and Kluwer's eBookstore at: http://ebooks.kluweronline.com

Contents

List of Figures

List of Tables

Preface

After several decades of sustained research and testing, linear programming has evolved into a remarkably reliable, accurate and useful tool for handling industrial optimization problems. Yet, large problems arising from several concrete applications routinely defeat the very best linear programming codes, running on the fastest computing hardware. Moreover, this is a trend that may well continue and intensify, as problem sizes escalate and the need for fast algorithms becomes more stringent.

Traditionally, the focus in optimization research, and in particular, in research on algorithms for linear programming, has been to solve problems "to optimality." In concrete implementations, this has always meant the solution of problems to some finite accuracy (for example, eight digits). An alternative approach would be to explicitly, and rigorously, trade off accuracy for speed. One motivating factor is that in many practical applications, quickly obtaining a partially accurate solution is much preferable to obtaining a very accurate solution very slowly. A secondary (and independent) consideration is that the input data in many practical applications has limited accuracy to begin with.

During the last ten years, a new body of research has emerged, which seeks to develop provably good approximation algorithms for classes of linear programming problems. This work both has roots in fundamental areas of mathematical programming and is also framed in the context of the modern theory of algorithms. The result of this work has been a family of algorithms with solid theoretical foundations and with growing experimental success.

In this manuscript we will study these algorithms, starting with some of the very earliest examples, and through the latest theoretical and computational developments.

DANIEL BIENSTOCK

1. Introduction

Since its inception, Mathematical Programming has combined methodology (theory), computation (experimentation) and pragmatism. What we mean by the last term is that Mathematical Programming arguably focuses on solving *models* as opposed to *problems*. Semantics aside, we frequently deviate from the mathematical ideal of being to solve an arbitrary mathematical programming formulation of a problem, when an alternative formulation will yield an easier solution which still addresses the "real" underlying problem.

Putting it differently, we prove theorems so as to obtain robust and fast algorithms; we deploy these using sophisticated implementations, and we use careful modeling to effectively leverage the resulting systems. Altogether, we obtain tools that we can trust because of their theoretical pedigree and which empirically yield good results, while at the same time tackling realistic problems. The focus is always that of obtaining good results: even though we intellectually enjoy solving mathematical puzzles, we are ready to substitute an algorithm/implementation/model for another if proven more effective.

The premier tool of Mathematical Programming has always been Linear Programming. Linear programs arise directly in many applications and also as subproblems of integer programs. As a result, we place a premium on being able to solve large linear programs speedily. During the last twenty years we have seen a number of important developments in Linear Programming. Among others, we cite the Ellipsoid Method (which underlined the issue of computational complexity, and which led to important consequences in Combinatorial Optimization), the probabilistic analysis of random linear programming instances (which tried to provide an explanation as to why linear programs, in practice, are not exponentially difficult), Karmarkar's algorithm (which catalyzed the field of interior point methods), the effective implementation of steepest-edge pivoting schemes (with important practical implications) and others. In parallel to this, the field of Numerical Analysis (in particular, numerical Linear Algebra) has greatly matured, yielding algorithms that are faster and numerically more stable – a good example for this would be the much improved Cholesky factorization routines now available. See [SC86], [W97]. Moreover, there has been growing understanding of what makes a "good" formulation. Finally, computational architectures and software environments have vastly improved, yielding dramatically more capable platforms and allowing algorithm implementors greater freedom to concentrate on high-level algorithm design. The result of all these positive developments, of course, has been a very visible improvement in the performance of linear programming codes. As experts have put it,

the best LP codes are thousands or hundreds of times faster than those available ten years ago and far more robust [B00], [D00].

In spite of this improvement, large linear programming instances arising from various practical applications frequently render linear programming packages inadequate. Consider Table 0.1, showing running times using the CPLEX LP solver package, and a modern computer, to solve *concurrent flow* problems, a class of routing problems arising in networking, that will be discussed at length later.

Instance	m	n	nonz	time (sec.)	its.
RMF20	40620	163181	489540	326	78649
RMF22	59323	241242	723723	809	116719
RMF15	90980	382721	1148160	3332	190324
RMF21	132968	565445	1696332	7058	278408
RMF13	160143	689056	2067177	9434	305711
RMF23	234446	1019054	3057169	40368	542959
RMF19	251201	1095606	3286819	66112	691682
RMF25	356669	1569446	4708372	137865	894641
RMF26	481236	2131038	6393176	455541	1560685

Table 0.1. CPLEX 6.6 dual on concurrent flow problems

Here m is the number of rows, n is the number of columns, and *nonz* is the number of nonzeros. The last two columns indicate the time to optimality using the dual simplex method (with steepest edge pivoting) and the number of iterations, respectively. Note that $n \approx 4m$ and $nonz = 3n$. Thus, the running time appears to grow cubically with the number of columns. Figure 0.1 overlays a best-fit cubic with a plot of time as a function of columns.

time (sec.)	1999	7852	41023	148811	160639	412155
bound	0.328	0.599	10.442	10.898	11.056	19.847

Table 0.2. Dual bounds on $RMF26$

Consider also Table 0.2. This table displays the performance of the dual simplex method on problem RMF26, as a function of time. At around 161000 seconds, the algorithm first proved a bound of 18.48, which is close to the optimum. The table shows that just prior to that point, the bound was still quite poor.

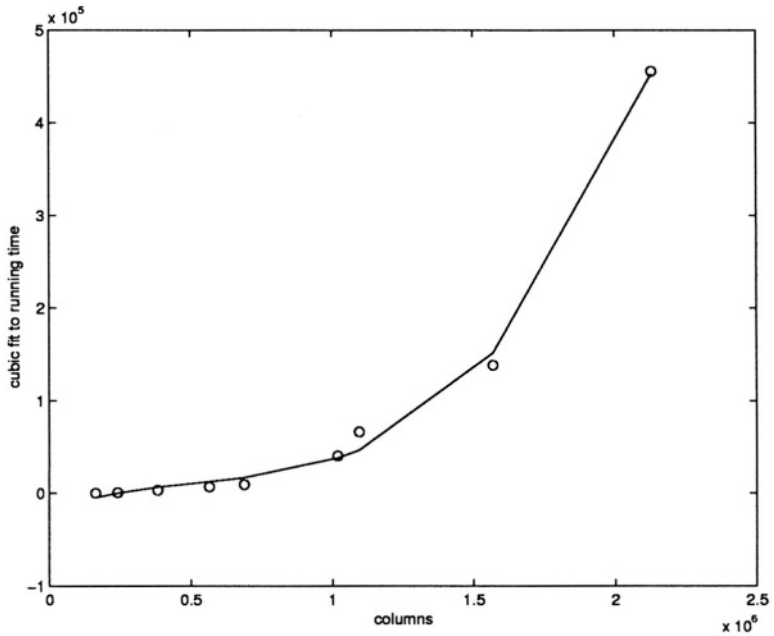

Figure 0.1. **Time as a function of columns for dual simplex on concurrent flow problems; best-fit cubic**

Finally, consider Figure 0.2. Here we present data on the network design problem instance *netd*9 which will be discussed in Chapter 4 (Table 4.1). To produce this Figure, this problem instance was solved using the Barrier code in CPLEX. To set an estimate of the accuracy of the algorithm as a function of time, at each iteration of the Barrier code we computed the relative deviation of the primal value (produced by the Barrier code) from the eventual optimum LP value, the relative deviation of the dual value from the LP value, and we used the maximum of these two deviations as our error estimate. We then discarded the initial set of iterations where this error bound was greater than 1.0. Figure 0.2 plots running time as a function of the logarithm (base 10) of the inverse of this error bound, which is an estimate of the number of correct digits produced by the code. As we can see from this Figure, the code had three digits of accuracy after roughly 87000 seconds. Further, until the code had (slightly more than) two digits of accuracy, running time grew faster than quadratically as a function of the inverse error. Finally, in the period where the code had between two and four digits of accuracy, running time was still growing as a positive power of the inverse error (but slower than quadratically).

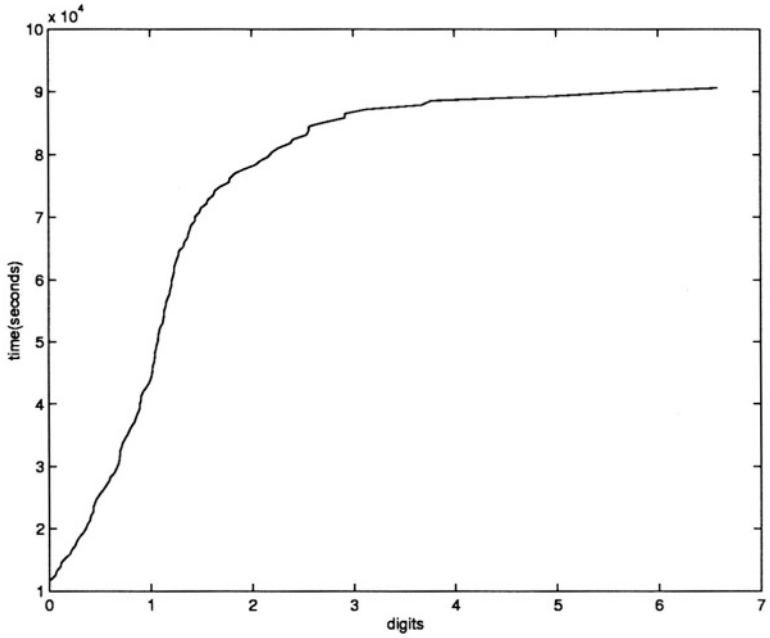

Figure 0.2. Time as a function of correct digits for Barrier code on instance *netd*9

The situation exemplified by this data has led researchers to consider *approximation algorithms* for linear programming, as an alternative to the traditional focus on "exactly" solving problems. The idea of approximation has a long and distinguished record in all areas of science and engineering. Surely, it should be a worthwhile scientific goal to develop approximation algorithms for linear programming that are endowed with solid mathematical foundations and which can be efficiently implemented. As for the usefulness of such approximation algorithms, it is abundantly clear from practical experience that obtaining a partially accurate answer, quickly, can be much preferable to having to wait an excessively long time for an "exact" solution (a misnomer given that standard LP codes incur round-off error). What is more, in many practical industrial applications it is debatable whether the input data is correct to more than two digits. Thus, in principle, we should be able to use approximation algorithms for linear programming and yet achieve the synergy between theory, computation and practicality mentioned above.

The issue of how rough an approximation "can" be in order to remain useful is difficult to quantify, and for good reason: this is entirely dependent on the particular needs of the application. Suppose that a real

application gave rise to a linear program with 10^9 variables and 10^6 constraints. In such a situation, being able to obtain a (guaranteed) factor of 2 estimate of the LP value in (say) one day could well be preferable to having to wait one year for an "exact" solution.

At the same time, one trend in the development of mathematical programming has been that problem sizes demanded by applications are constantly increasing. Whereas ten years ago it may have been somewhat rare to have to solve a one-million variable linear program, today many real applications routinely give rise to problems of this order of magnitude. And in some applications we face much larger problems. It is certain that in the future we will face problems that are vastly larger. Given this likelihood, should we abandon the richness and flexibility that we can achieve with today's models, so as to be able to claim that we "solve" problems to optimality? We would argue that as problems become massive, and the asymptotic properties of algorithms become visible, the need for provably good approximation schemes with good experimental behavior will become pressing.

In this monograph we survey theoretical and experimental results on approximation algorithms for linear programs, with special emphasis on provably good potential function methods. In Chapter 1 we survey some classical results that constitute the starting point for this field. Chapter 2 describes some of the core results on exponential potential function methods. Chapter 3 contains some fairly recent results on streamlined algorithms based on exponential penalties. Chapter 4 presents our results with an implementation of the exponential potential function method for general block-angular linear programs.

Acknowledgment. This work was partially funded by NSF awards NCR-9706029, CDA-9726385 and 29132-5538 (CORC).

Chapter 1

EARLY ALGORITHMS

Optimization problems in networking and telecommunications have historically given rise to very difficult linear programming instances. This is still true today – with the added hazard of massive problem size. In particular, problems arising in this field contain routing components, which can be abstracted as multicommodity flow problems. Although there has been a large amount of research into multicommodity flow problems, just why they are so difficult remains poorly understood.

Broadly speaking, in a multicommodity flow problem we are given a graph in which multiple "demands" are to be routed. For the purposes of this chapter, we may think of a demand as an origin-destination pair of nodes and an amount to be routed from the origin to the destination. In addition, for each edge (i, j) there is an upper bound u_{ij} on the sum of the flows of all commodities on (i, j). Here we are assuming that the graph is directed (and that the demands are routed using directed flows) but the problem and the solution techniques easily generalize. In summary, a multicommodity flow is a solution to a linear system of the form

$$\sum_k f_{ij}^k \leq u_{ij} \quad \forall\,(i,j) \tag{1.1}$$

$$N^k f^k = b^k, \quad k = 1, \cdots, K \tag{1.2}$$

$$0 \leq f^k \qquad k = 1, \cdots, K \tag{1.3}$$

Here K is the number of commodities, and for each k, (1.2) is the set of flow conservation equations for commodity k, with f_{ij}^k the flow of commodity k on (i, j). Sometimes the problem may include upper bounds on each f_{ij}^k. *Minimum cost* multicommodity flow problems are (usually linear) optimization problems with constraints (1.1)-(1.3). Multicommod-

ity flow problems are difficult because of the joint capacity constraints (1.1), which can actually render the problem infeasible. In fact, it has long been known that the pure feasibility problem can already be fairly difficult. When a system (1.1)-(1.3) is infeasible, it is sometimes important to know "how" infeasible it is. This question can be cast as an optimization problem by replacing (1.1) with

$$\sum_k f_{ij}^k \leq \lambda u_{ij} \quad \forall\, (i,j) \tag{1.4}$$

and asking that λ be minimized, subject to (1.4), (1.2) and (1.3). Define the *congestion* of a multicommodity flow f on an edge (i,j) *as* $\frac{\sum_k f_{ij}^k}{u_{ij}}$. Then in the above problem we are asking that commodities should be routed so as to minimize the maximum congestion.

Alternatively, we could keep (1.1) and (1.3), and adjust (1.2) to obtain

$$\begin{aligned} & \max \quad \theta \\ \textbf{(CONG)} \qquad & \text{s.t.} \\ \sum_k f_{ij}^k \leq\ & u_{ij} \quad \forall\, (i,j) && (1.5) \\ N^k f^k =\ & \theta b^k, \quad k = 1, \cdots, K && (1.6) \\ 0 \leq\ & f^k \qquad\qquad k = 1, \cdots, K && (1.7) \end{aligned}$$

In other words, we want to maximize the "throughput" (θ) of the network: we want to route as large a common percentage of each demand as possible while not exceeding any of the joint capacities. This is known as the *maximum concurrent flow* problem. Both problems have important applications in networking, and perhaps it is their min-max nature that makes the underlying optimization problems difficult. Note that a feasible solution (f, θ) to **CONG,** with $\theta > 0$, yields the multicommodity flow $\theta^{-1} f$ which is feasible to (1.2)-(1.3) and which has maximum congestion at most θ^{-1}. Using this fact, the minimum congestion and maximum concurrent flow problems are easily seen to be equivalent.

Minimum congestion/maximum throughput problems arise in many guises in networking applications, always with the common thread of "packing" commodities into subgraphs of a network (paths, trees, etc) so as to optimize a min-max or max-min measure of the packing. Another common property is that, when tackled as general linear programs, these problems can be rather ferociously difficult. This fact has provided motivation for studying these problems from the very earliest days and continuing through today. The particular formulation described above (which does arise directly) can be considered an abstracted form of the

general problem, which captures its significant characteristics. The approximation algorithms that we will describe, in turn, are easily adapted to particular forms of the problem.

1. The Flow Deviation Method

In 1971, Fratta, Gerla and Kleinrock [FGK71] developed an iterative approximation algorithm for various routing problems in telecommunications, the "Flow Deviation Method." Their work is evocative of research published much later, and in fact some of their techniques can be found almost verbatim in articles published twenty years later. At the same time, the Flow Deviation Method can be viewed as an amalgam of research ideas developed in the sixties and before by the mathematical programming community. We will examine the results in [FGK71] in some detail, as they provide a good stepping stone to modern work, and because a careful analysis reveals a surprise: the algorithm for the maximum throughput problem given in [FGK71] is provably good, thereby predating similar results by some twenty years.

In [FGK71] the authors consider two closely related nonlinear optimization problems. The central one has constraints (1.1), (1.2) and (1.3), and its objective, to be minimized, is of the form

$$\Psi(f) = \sum_{ij} \frac{\sum_k f_{ij}^k}{u_{ij} - \sum_k f_{ij}^k} \tag{1.8}$$

There is a concrete justification for this nonlinear objective arising from the application (related to queueing delays), but, as quickly becomes clear, the justification is serendipitous, because the crucial role for this objective in [FGK71] is as a *barrier* function in an algorithm for finding a multicommodity flow feasible to (1.1)-(1.3). A simple modification to this algorithm solves the maximum concurrent flow problem. In this role, the particular form of $\Psi(f)$ is unimportant (as stressed in [FGK71]), other than

(i) Ψ is separable, or more precisely, $\Psi(f) = \sum_{ij} \psi(\sum_k f_{ij}^k / u_{ij})$ for an appropriate ψ such that

(ii) In the range $0 \leq x < 1$, $\psi(x)$ is increasing, continuous and convex, and $\psi(x) \to +\infty$ as $x \to 1^-$.

The particular Ψ used in [FGK71] yields $\psi(x) = \frac{x}{1-x}$. For completeness, we define $\psi(x) = +\infty$ outside the range $0 <= x < 1$.

In the remainder of this section we outline the algorithm used in [FGK71] to solve the maximum concurrent flow problem, in slightly

modified form. We refer the reader to [BR00], [R01] for a rigorous analysis of the algorithm in [FGK71] that proves better bounds than those we will obtain below.

It is instructive to first examine the central idea in this algorithm, which in our view is the critical contribution. Suppose we are given a flow vector f with throughput θ, and suppose that f is strictly feasible, i.e. all constraints (1.5) are slack. To fix ideas, suppose all capacities u_{ij} are at most 50% utilized, and that this congestion level is attained on some edge. Then it is a simple matter to increase throughput: simply scale up all flows by a factor of 2. The new flow will still be feasible, with throughput 2θ, and will fully utilize the capacity of at least one edge.

In general; since f is strictly feasible, there is a number $\beta > 1$ (i.e. the inverse of the maximum congestion) such that βf is feasible, with throughput $\beta\theta$. However, this comes at the cost of fully utilizing at least one edge of the network, that is βf will have some edge with congestion 1. Given this new flow, we can ask the following question: can we find a flow vector g, also with throughput $\beta\theta$, but with (much) smaller congestions than βf? If we could magically find such a vector, we would have an algorithm, because we could reset $f \leftarrow g$ and go back to the starting point above, scale up f, and so on. The way that the "magic" is implemented in [FGK71] is to obtain g from βf by reducing the value of a nonlinear function that penalizes congestion; namely Ψ given above. We need a quick correction here: since Ψ equals $+\infty$ at 1, we must choose the quantity β so that βf is also strictly feasible (in the 50% example we would set, say $\beta = 1.99$).

Now we return to the description of the general algorithm. It relies on an iterative routine (the Flow Deviation method, described below) to implement the "magic" described above, i.e. to approximately solve problems of the form

$$\begin{array}{ll} & \Psi^* = \min \quad \Psi(z) \\ \mathbf{R}(\theta) & \text{s.t.} \\ & N^k z^k = \theta b^k, \quad k = 1, \cdots, K \qquad (1.9) \\ & 0 \leq z^k \qquad\qquad k = 1, \cdots, K, \qquad (1.10) \end{array}$$

where $0 < \theta$ is a parameter. This iterative routine, which will be described later, requires as starting point a feasible solution to (1.9 - 1.10).

In outline, the algorithm for the concurrent flow problem is as follows, where $0 < \epsilon$ is the desired optimality tolerance, and $0 < \gamma$ is a related parameter discussed later.

Algorithm FGK

Step 1. Let $(f^1,\ \theta^1)$ be a feasible solution to **CONG**, with maximum congestion $1 - \frac{\gamma}{2}$. Let $t = 1$.

Step 2. Use the Flow Deviation method to compute, starting from f^t, a feasible solution y^t to $R(\theta^t)$ with $\Psi(y^t) \leq \max\{(1+\epsilon)\Psi^*, \Psi^* + \epsilon\}$.

Step 3. Let λ^t denote the maximum congestion of y^t. If $\lambda^t \geq 1 - \gamma$, stop, θ^t is the maximum throughput, up to tolerance.

Step 4. Otherwise, $f^{t+1} \doteq \frac{1-\frac{1}{2}(1-\lambda^t)}{\lambda^t} y^t = \frac{1+\lambda^t}{2\lambda^t} y^t$ is a multicommodity flow with throughput $\theta^{t+1} \doteq \frac{1+\lambda^t}{2\lambda^t}\theta^t$, i.e. (f^{t+1}, θ^{t+1}) is feasible for **CONG**. Set $t \leftarrow t+1$ and go to Step 2.

There are several technical details about this algorithm that are worth noting. First, for $t \geq 2$ consider the flow f^t produced in Step 4 at the end of iteration $t-1$, which has has throughput θ^t and maximum congestion $\frac{1+\lambda^{t-1}}{2}$, achieved on at least one edge (i,j). The objective in Step 3 of iteration t is to compute another flow with equal throughput (θ^t) but with hopefully much smaller congestions. Having achieved this purpose, in Step 4 we use the free capacity available in *every* edge so as to scale up the flow, and increase throughput, by a factor of $\frac{1+\lambda^t}{2\lambda^t} = \frac{1+1/\lambda^t}{2} \geq 1 + \gamma/2$.

Another point is how to implement Step 1. This can be carried out by individually routing every commodity, and then scaling down the combined flow appropriately.

Finally, the implementation of Step 2 given above is somewhat different from that in [FGK71]. We will return to this point later. The second term in the maximum is needed to avoid difficult cases of the optimization problem in Step 2, where the value of the optimum Ψ is very small.

1.1 Convergence Analysis

What can be proved about the **FGK** algorithm? First, one wonders if the termination criterion is correct. Let m be the number of edges in the graph. The following result, although not proved in [FGK71], follows easily.

THEOREM 1.1 *Suppose* ϵ *is small enough that* $\psi(1-\epsilon) > 1$, *and that* $0 < \gamma < \epsilon$ *satisfies* $\psi(1-\gamma) > m(1+\epsilon)\psi(1-\epsilon)$. *If algorithm FGK stops*

(in Step 3) at iteration t, then $\theta^t \geq (1-\epsilon)\theta^*$, *where* θ^* *is the maximum throughput.*

Proof. Assume by contradiction that $\theta^t < (1-\epsilon)\theta^*$, and let f^* be a multicommodity flow feasible for **CONG,** with throughput θ^*. Then the flow $w = \frac{\theta^t}{\theta^*} f^*$ achieves throughput θ^t. Moreover, for every edge (i,j)

$$\frac{\sum_k w_{ij}^k}{u_{ij}} \leq \frac{\theta^t}{\theta^*} < (1-\epsilon), \tag{1.11}$$

and consequently

$$m\psi(1-\epsilon) > \Psi(w). \tag{1.12}$$

Since the algorithm stopped at iteration t, $\lambda^t > 1-\gamma$, and thus

$$\Psi(y^t) \geq \psi(1-\gamma) > m(1+\epsilon)\psi(1-\epsilon) \tag{1.13}$$

$$> (1+\epsilon)\Psi(w). \tag{1.14}$$

Consider the value of Ψ^* in Step 2 (at termination). Since $\psi(1-\epsilon) > 1$, from (1.13) we get

$$\begin{aligned} m+\epsilon &< \Psi(y^t) \\ &\leq \max\{(1+\epsilon)\Psi^*, \Psi^*+\epsilon\}. \end{aligned} \tag{1.15}$$

This implies $\Psi(y^t) \leq (1+\epsilon)\Psi^*$, which contradicts (1.14). ■

We note first that the properties of ψ guarantee that for any ϵ small enough, a value γ as desired exists. Using modern terminology, the theorem states that the throughput produced by the algorithm at termination is $\epsilon - optimal$ with regards to the objective Ψ. We may think of Theorem 1.1 as specifying, for each fixed desired accuracy $\epsilon > 0$, an internal precision γ to be used in running algorithm **FGK.** Note that γ could be very small compared to ϵ, potentially outstripping the numerical precision we can realistically expect from a modern computer (the tighter analysis presented in [BR00], [R01] does not suffer from this shortcoming). Another consequence is that algorithm **FGK** might stop with a value θ^t very close to, but strictly smaller than 1. By scaling the flows by a factor $1/\theta^t$ we obtain a multicommodity flow that is almost feasible for (1.1)-(1.3) – more precisely the maximum relative infeasibility is at most ϵ – but no guarantee that (1.1)-(1.3) is infeasible.

The next point of interest regarding algorithm **FGK** is *how many* iterations it will require to reach termination. This point was not addressed in [FGK71]. In order to obtain a provably fast algorithm we find that we must make two adjustments to the generic algorithm.

The first issue concerns the relationship between ϵ and γ, which is defined by the particular function ψ we choose: if γ is too small the algorithm may converge slowly. We will make the additional assumption:

(iii) There is a value $c > 0$ such that for all $0 < \epsilon < 1/2$, $\psi(1 - c\frac{\epsilon}{m}) > m\psi(1-\epsilon)$, i.e. $\gamma \geq c\epsilon m$.

For $\psi(x) = \frac{x}{1-x}$, $c = \frac{1}{3m}$ suffices.

The other issue concerns the initial value θ^1: should this be too small the algorithm will require many iterations. One way to address this issue is simple and intuitive enough that it might be inadvertently chosen when implementing Step 1 of **FGK**. First compute, for each commodity k $(1 \leq k \leq K)$, a maximum flow vector (with capacities u_{ij}). Let θ_{min} be the minimum, over all commodities, of the throughput achieved by this routing. Then we can assign throughput $\theta_{min} * (1 - \gamma/2) * K^{-1}$ to all commodities to obtain a flow as desired for Step 1.

The next Theorem parallels analysis found in work dating some twenty years later than that in [*FGK*71].

THEOREM 1.2 *Suppose ψ satisfies (iii) and that the flows in Step 1 are computed using a maximum flow algorithm, and scaling. Then, given $\epsilon > 0$,* **FGK** *converges in at most $O(m \log \epsilon^{-1} + \log K)$ iterations of Step 2.*

Proof. If we obtain the initial flow vector f^1 as indicated above we will have

$$\theta^* \leq (K+1)\theta^1. \tag{1.16}$$

Consider an iteration t where

$$(1 - 2\epsilon)\theta^* \leq \theta^t \tag{1.17}$$

If the algorithm does not stop at this iteration, then we will have

$$\theta^{t+1} \geq (1 + \gamma/2)\theta^t \geq \left(1 + \Omega\left(\frac{\epsilon}{m}\right)\right)\theta^t,$$

and consequently there are at most $O(m)$ iterations such as (1.17). Using the same technique, if $r > 0$ is an integer such that $\epsilon < 2^{-(r+1)}$, the number of iterations t where

$$(1 - 2^{-r})\theta^* \leq \theta^t < (1 - 2^{-(r+1)})\theta^*$$

can be shown to be $O(m)$. Finally, since $\theta^* \leq O(K\theta^0)$, another use of the same idea shows that there are at most $\log K$ iterations t with $\theta^* > 2\theta^t$. ■

1.2 Analysis of the Flow Deviation method

Now we turn to the algorithm used in [FGK71] to solve the problem $R(\theta)$ in Step 2 of algorithm **FGK**. For convenience, we restate this problem here:

$$\mathbf{R}(\theta) \qquad \begin{aligned} \Psi^*(\theta) &= \min \quad \Psi(z) \\ &\text{s.t.} \end{aligned}$$

$$N^k z^k = \theta b^k, \quad k = 1, \cdots, K \tag{1.18}$$

$$0 \leq z^k \qquad k = 1, \cdots, K, \tag{1.19}$$

The algorithm presented in [FGK71] is iterative and can be viewed as a steepest descent method, or, more properly, a Frank-Wolfe method (see [FW56]).

Flow Deviation Method

FD0. Let z^0 satisfy (1.18)-(1.19). Set $h = 0$.

FD1. Let v^h be an optimal solution to the linear program

$$\min \quad [\nabla \Psi(z^h)]^T v$$

s.t. v satisfies (1.18)-(1.19).

FD2. Choose δ_h which minimizes $\Psi((1-\delta)z^h + \delta v^h)$ over $0 \leq \delta \leq 1$.

FD3. Set $z^{h+1} \doteq (1-\delta_h)z^h + \delta_h v^h$. If $\Psi(z^{h+1}$ is not appreciably smaller than Ψ^h, **stop.** Otherwise set $h \leftarrow h+1$ and go to **1**.

Given that Ψ is convex, the above algorithm will converge to a global optimum of Ψ, although it is a classical result that the convergence will be slow if the eigenvalue structure of the Hessian matrix of Ψ is not convenient. Parenthetically, we note that in [FGK71] the terminology "flow deviation method" is sometimes used to refer just to one application of **FD1 - FD3**. In [FGK71] the authors present the following analysis of the convergence rate of the above algorithm.

At iteration h, for $0 \leq \delta \leq 1$ write

$$g(\delta) \doteq \Psi((1-\delta)z^h + \delta v^h) = \Psi(z^h + \delta(v^h - z^h)). \quad (1.20)$$

Then g is convex, $g(0) = \Psi(z^h)$,

$$g'(0) = [\nabla\Psi(z^h)]^T \cdot (v^h - z^h), \text{ and} \quad (1.21)$$

$$\Psi(z^{h+1}) = g(\delta_h) = g(0) + g'(0)\delta_h + (1/2)g''(\alpha)\delta_h^2, \quad (1.22)$$

where $0 < \alpha < \delta_h$ (2nd order Taylor expansion). Let $H > 0$ be an upper bound on $g''(\alpha)$. [To see that such a finite bound exists, note that that since for any $h > 0$, $\Psi(z^{h+1}) \leq \Psi(z^0)$, we have $z_{ij}^{h+1} \leq 1 - c\frac{1-z_{ij}^0}{m}$, for any (i,j), where c is the constant in assumption (iii).] By our choice of v^h in **FD1**, replacing v^h with any other flow vector will not decrease the right-hand side of equation (1.21). In particular, we may use a flow z^* optimal for $\mathbf{R}(\theta)$. Putting together these facts and the above equations we obtain

$$\Psi(z^{h+1}) - \Psi(z^h) \leq [\nabla\Psi(z^h)]^T \cdot (z^* - z^h)\delta_h + (1/2)H\delta_h^2. \quad (1.23)$$

By definition $\Psi^*(\theta) = \Psi(z^*)$, and since Ψ is convex,

$$\Psi^*(\theta) \geq \Psi(z^h) + [\nabla\Psi(z^h)]^T \cdot (z^* - z^h); \quad (1.24)$$

if we write $\Psi^h \doteq \Psi(z^h)$ and substitute in (1.23), we obtain

$$\Psi^{h+1} - \Psi^h \leq (\Psi^*(\theta) - \Psi^h)\delta_h + (1/2)H\delta_h^2 \doteq Q(\delta_h). \quad (1.25)$$

We choose δ_h in **FD3** so as to minimize the left-hand side of (1.25). We would like to argue that the minimum value that the quadratic $Q(\delta_h)$ takes, over **all** possible $0 \leq \delta_h \leq 1$, is an upper bound on $\Psi^{h+1} - \Psi^h$. This will follow if we can guarantee that the value of $0 \leq \delta_h \leq 1$ that minimizes $Q(\delta_h)$ is such that (1.25) holds. The choice of H guarantees that (1.25) is valid whenever

$$(1-\delta_h)z_{ij}^h + \delta_h v_{ij}^h \leq 1 - c\frac{1-z_{ij}^0}{m} \quad (1.26)$$

for each edge (i,j). On the other hand, if $\delta_h = \hat{\delta}$ is such that (1.26) holds with equality for some edge (i,j), then $\Psi(z^h) \leq \Psi((1-\hat{\delta})z^h + \hat{\delta}v^h)$, i.e.,

$Q(\hat{\delta}) \geq 0$, from which it follows, since $Q(0) = 0$, that $Q(\delta_h)$ indeed achieves its minimum (over the reals) in the interval $[0, \hat{\delta})$, as desired.

Note that unless z^h is already optimal, $\Psi^*(\theta) - \Psi^h < 0$, and a calculation shows that

$$\Psi^{h+1} - \Psi^h \leq -\frac{H}{2} \min\left\{ \frac{(\Psi^*(\theta) - \Psi^h)^2}{H^2}, 1 \right\} \qquad (1.27)$$

In [FGK71] this equation is used to argue that the Flow Deviation Method asymptotically converges to an optimal solution. Here, we will instead tighten the analysis to obtain a complexity result for the algorithm. First, we need to make an additional assumption on the barrier function ψ. This assumption essentially states that as $x \to 1^-$, ψ does not grow "too fast".

(iv) There is a value $p > 0$ such that for all $x < 1$,

$$\psi''(x) = O((1-x)^{-p}).$$

For $\psi(x) = \frac{x}{1-x}$ we have $p = 3$. Another ingredient that enters the mix here is the parameter that in later work became known as the "width". For the purposes of the maximum concurrent flow problem, the width σ equals the ratio of the largest demand to the smallest capacity.

Now if we assume that $\Psi^0 = O(m^2/\epsilon)$, then the same will hold for all Ψ^h since algorithm **FDO-FD3** strictly decreases Ψ. Thus, an analysis shows that $H \leq O(\sigma^2 m^{2p}/\epsilon^p)$. Combining these observations with inequality (1.27), we obtain

$$(\Psi^{h+1} - \Psi^*(\theta)) - (\Psi^h - \Psi^*(\theta)) \leq -\frac{\sigma^2 \epsilon^p}{2m^{2p}} (\Psi^h - \Psi^*(\theta))^2. \qquad (1.28)$$

The recursion given by this equation can be abstracted as

$$x_{h+1} - x_h \leq -L x_h^2.$$

Given any $\mu > 0$, this recursion satisfies

$$x_h \leq \mu \text{ for } h \geq O((\mu L)^{-1} + 1). \qquad (1.29)$$

Thus, $\Psi^h - \Psi^*(\theta) < \epsilon$ for $h = O(\frac{m^{2p}}{\epsilon^{p+1}})$. Putting all these facts together, and making use of equation (1.29) we obtain:

THEOREM 1.3 *Suppose ψ satisfies (i) - (iv), and suppose θ is at least a fraction $\Omega(K^{-1})$ as large as the maximum throughput. Then algorithm* **FD0-FD3** *converges to a feasible flow z with*

$$\Psi(z) \leq \max\{(1+\epsilon)\Psi^*(\theta), \Psi^*(\theta) + \epsilon\} \qquad (1.30)$$

in at most $O(\sigma^2 m^{2p} \epsilon^{-(p+1)})$ *iterations.* ■

Note that, as stated, Theorem 1.3 actually guarantees the stronger condition $\Psi(z) \leq \Psi^*(\theta) + \epsilon$. Thus equation (1.28) can be used to detect termination. As a corollary of this theorem, we now have:

THEOREM 1.4 *Suppose* ψ *satisfies (i) - (iv), and that we start algorithm* **FGK** *by computing a maximum flow for each commodity, and scaling to achieve feasibility. Then algorithm* **FGK** *finds a feasible flow, with throughput* $\theta^t \geq (1-\epsilon)\theta^*$, *after solving at most* $O(\sigma^2 m^{2p} \epsilon^{-(p+1)} \log(\epsilon^{-1}))$ *linear programs with constraints (1.9)-(1.10).* ■

Using a slightly tighter analysis, we can improve the bound in Theorem 1.3 and through it, that in Theorem 1.4. The point is that, as noted before, the condition $\Psi(z) \leq \Psi^*(\theta) + \epsilon$ guaranteed by Theorem 1.3 is stronger than needed. This condition is needed to handle the initial calls to the Flow Deviation method during the course of algorithm **FGK**, where the quantity Ψ^* may be too small to approximate very accurately (and it is unimportant that we do so). The following is easily obtained:

LEMMA 1.5 *Consider an iteration t of* **FGK** *where* $(1-2^{-r})\theta^* \leq \theta^t < (1-2^{-(r+1)})\theta^*$ *for some integer* $r > 0$. *Then* $\Psi^*(\theta^t) \geq \Omega(2^r)$. ■

Inserting this result directly in Theorem 1.3 yields the improved iteration bound $O(\sigma^2 m^{2p} \epsilon^{-p} \log(\epsilon^{-1}))$ for algorithm **FGK**, instead of that in Theorem 1.4. We can obtain another improvement by noticing that algorithm **FGK** from the outset is working with the final precision ϵ; but a more effective implementation is to gradually increase the precision (by powers of two) until we obtain the desired ϵ. Then we will have:

THEOREM 1.6 *Algorithm* **FGK** *can be implemented so as to terminate with a throughput* $\theta^t \geq (1-\epsilon)\theta^*$, *after solving at most* $O(\sigma^2 m^{2p} \epsilon^{-p})$ *linear programs with constraints (1.9)-(1.10).* ■

In [BR00], [R01] we analyze an algorithm that follows that in [FGK71] fairly closely, and show that our algorithm solves the maximum concurrent flow problem to relative error ϵ by solving $O(\epsilon^{-2} m^3 k + m^3 k \log m)$ minimum-cost flow computations, where m and k are, respectively, the number of edges and commodities.

1.3 Historical Perspective

Algorithm **FGK** approximately solves a sequence of nonlinear programs $R(\theta)$, and through the algorithm given in Section 1.2, it produces a sequence of Frank-Wolfe steps, i.e., linear programs with constraints

(1.18)-(1.19). Each of these problems breaks up into one shortest path computation for each commodity and thus, putting aside Step 0 of algorithm **FGK,** the entire procedure for the concurrent flow problem can be viewed as sequence of shortest path problems.

To put it differently, steps **FD1 - FD2** of the Flow Deviation Method embody the following simple idea: in order to "improve" a given flow with a given throughput, we compute edge weights that penalize congestion (with weights increasing rapidly with congestion). Then each commodity we compute a shortest path between its ends using the computed weights, and then we shift ("deviate") some of flow of the commodity to the computed path. The procedure then repeats using the new flow vector. Conceptually, we might think of the deviation as transferring weight from a "long" paths to a shortest path.

According to [FGK71], Dafermos and Sparrow [DS69] already described an embryonic version of this idea, although in a more limited setting. More generally, throughout the above we have been describing the Flow Deviation Method as a "Frank-Wolfe" algorithm: this is the technique whereby a nonlinear objective is linearized by replacing it with its gradient, which is related to the classical steepest descent algorithm for unconstrained optimization (see [FW56], [Lu]). It is also related (consider step **FD2**) to Dantzig-Wolfe decomposition [DW]. In fact, the extreme points of the polyhedron defined by (1.18)-(1.19) are the incidence vectors of the shortest paths for all the commodities, scaled by the demand amounts. Hence another implementation of the Flow Deviation method is that it computes, for each commodity, that convex combination of paths that yields the lowest possible objective.

Taking a step further back, algorithm **FGK** removes the troublesome capacity constraints (1.5) and instead "penalizes" us when we get too close to violating them by using the nonlinear function Ψ as a "barrier". See [FM68] for a general analysis of the penalty function technique, which is closely related to the barrier function method. Courant [Cou43] is viewed as an early proponent of this type of idea.

In summary, the Flow Deviation Method made use of several algorithmic ideas that were already fairly mature at the time of its development. What seems to be novel is their application to a min-max problem, as embodied, in particular, by Steps 2-4 of algorithm **FGK**, as well as the first part of the convergence analysis of the Flow Deviation Method proper, as described in Section 1.2.

Quite likely, step **FD1** would be the most time-consuming when running the Flow Deviation Method. A simple improvement would be to restrict each application of steps **FD1 - FD3** to one commodity (and, for example, to cycle through the commodities). In this approach each step

FD1 is substantially faster, while at the same time the more frequent gradient updates may even yield faster convergence. Ideas of this sort were used in more recent algorithms to be studied in later sections, and parallel the improvement of the Gauss-Seidel scheme for solving linear systems over Gauss-Jacobi.

It is also likely that, due to its steepest-descent pedigree, the Flow Deviation Method may "tail-off". In the context of its utilization in algorithm **FGK**, one might then wish to replace the call in **Step 2** with a moderate number of iterations of **FD1 - FD3**. In particular, one might use just one iteration. This is actually the approach used in [FGK71]. If we further restrict each iteration of **FD1 - FD3** to one commodity at a time, then we obtain an algorithm that is (a bit) reminiscent of the Garg-Könemann [GK98] algorithm that we will discuss later.

2. The Shahrokhi and Matula Algorithm

Even though the Flow Deviation Method was always quite well-known within the telecommunications community (and somewhat less known within the optimization community) the above complexity analysis is the first, as far as we know.

In the late 1980s, Shahrokhi and Matula [SM91] developed a different technique for approximately solving the maximum concurrent flow problem in the special case that all arc capacities u_{ij} are equal to a common value u. This is known as the *uniform* maximum concurrent flow problem. Their technique relies on a penalty function (as opposed to a barrier function) and can also be viewed as a Frank-Wolfe algorithm (thus eventually breaking up into a sequence of shortest path computations), but the similarities with the Flow Deviation Method end there.

In addition, the method in [SM91] introduced a component that is conspicuously missing in algorithm **FGK**: the use of duality. In the remainder of this section, we will outline the basic motivation for the algorithm in [SM91], and how it naturally gives rise to duality.

The Shahrokhi-Matula algorithm is best viewed in the context of the min-max congestion problem, which for convenience we restate here.

$$\begin{array}{ll} & \min \quad \lambda \\ \textbf{(CONG)} & \text{s.t.} \end{array}$$

$$\sum_k f_{ij}^k \leq \lambda u_{ij} \quad \forall\,(i,j) \tag{1.31}$$

$$N^k f^k = b^k, \quad k = 1, \cdots, K \tag{1.32}$$

$$0 \leq f^k \qquad k = 1, \cdots, K \tag{1.33}$$

In an optimal solution to this problem we would expect flows to be "balanced," i.e. ideally no arc should carry "much more" flow than any other arc. This suggests using a convex, increasing function ϕ, and replacing the above linear program with

$$\begin{array}{ll} & \min \ \sum_{ij} \phi(\sum_k f_{ij}^k) \\ \textbf{(PF)} & \text{s.t.} \\ N^k f^k & = b^k, \quad k = 1, \cdots, K \qquad (1.34) \\ 0 \leq f^k & \qquad k = 1, \cdots, K \qquad (1.35) \end{array}$$

To motivate this approach, consider the case of a directed graph with vertex set $\{a, b, c, d\}$ and arc set $\{(a, b), (b, c), (c, d), (a, d)\}$, where every arc has unit capacity and one unit of flow is to be sent from a to d. The optimal solution sends 1/2 units of flow along the path a, b, c, d and another 1/2 along a, d. Suppose we choose $\phi(x) = x^p$ for some $p > 0$. In this example problem **PF** takes the form

$$\begin{array}{ll} & \min \ 3(y)^p + (1-y)^p \\ \textbf{(PF)} & \text{s.t.} \\ & 0 \leq y \leq 1 \end{array}$$

with solution $y = (1 + 3^{\frac{1}{p-1}})^{-1}$. Thus for $p = 2$ the congestion achieved by this solution is 3/4. For $p = 50$, the congestion is 0.5056, i.e. still more than 1% away from the optimum. The problem here is that our choice for ϕ does not increase fast enough, and as a result the optimal solution to problem **PF** will favor short paths over achieving minimum congestion. We are drawn then to a function ϕ of the form $e^{\alpha x}$ for appropriately chosen $\alpha > 0$.

To better motivate the use of an exponential potential/penalty function, it is useful to consider a generalization of the minimum congestion problem **CONG**. Here we are given a convex set $P \subseteq R^n$, and a non-negative $m \times n$ matrix A. We are interested in the min-max "packing" problem **PACK(A,P)** given by:

$$\begin{array}{ll} & \lambda^*_{A,P} = \min \quad \lambda \\ \textbf{PACK(A,P):} & \text{s.t.} \\ & Ax \ \leq \ \lambda e \\ & x \ \in \ P, \end{array}$$

where e denotes the vector of 1s. In terms of problem **CONG**, the capacity inequalities (1.31) can be represented by a system of the form $Ax \leq \lambda e$, and P would denote the convex hull of flow vectors that satisfy

(1.32), (1.33). Thus $\lambda^*_{A,P}$ is the value of problem **CONG**, i.e., the minimum congestion value. Equivalently, we may describe the minimum congestion problem using a *path* formulation. In this formulation the set P is homeomorphic to a product of simplices; $P = P^1 \times P^2 \times \cdots \times P^K$, where for each commodity k, P^k is homeomorphic to the simplex $\left\{x \in R_+^{n(k)} : \sum x_j = 1\right\}$, i.e., the convex hull of paths corresponding to commodity k. Note that the system $S = \{Ax \leq e,\ x \in P\}$ is feasible precisely when $\lambda^*_{A,P} \leq 1$, and we may view the task of finding an approximate solution to **PACK(A,P)** as a (strengthened) feasibility check.

For $x \in R^n$ let $\Phi(x) = \sum_i e^{\alpha(\sum_j a_{ij}x_j)}$, and define

$$\lambda_A(x) \quad = \quad \max_{1 \leq i \leq m} \sum_j a_{ij}x_j \tag{1.36}$$

The following theorem is adapted from [GK94].

THEOREM 1.7 *Let $y \in R^n$ an optimal solution to $min_{x \in P}\Phi(x)$. Then*

$$\max_{1 \leq i \leq m} \sum_j a_{ij}y_j \quad \leq \quad \frac{\log m}{\alpha} + \lambda^*_{A,P}.$$

Proof. Suppose x^* is an optimal solution to problem **PACK(A,P)**. Then the total penalty incurred by y is at least $e^{\alpha\lambda_A(y)}$, and the total penalty incurred by x^* is at most $me^{\alpha\lambda^*_{A,P}}$, and so $e^{\alpha\lambda_A(y)} \leq me^{\alpha\lambda^*}$, which yields the desired result. ■

The significance of this theorem, as applied to the minimum congestion problem, is that we can adjust the parameter α so that the solution to problem **PF** is a flow vector whose congestion is optimal within any desired tolerance. In particular, if we knew $\lambda^*_{A,P} \geq 1$, we could in principle set $\alpha = \frac{\log m}{\epsilon}$ to obtain a solution with congestion at most $(1+\epsilon)\lambda^*_{A,P}$. It is also striking that the Theorem does not make use of any combinatorial features of the problem. Finally, the proof of Theorem 1.7 has a hint of brute-force flavor: the principle at work is that if we choose α large enough, then a flow vector with a single highly congested edge will be very suboptimal for problem **PF**.

In light of Theorem 1.7, let us consider the vector y that minimizes Φ over P. Since Φ and P are convex, we have that y is a solution to the linear optimization problem

$$\min\left\{[\nabla\Phi(y)]^T x : x \in P\right\}$$

which in the case of the minimum congestion problem is a minimum-cost flow problem. But a simple calculation shows that

$$[\nabla\Phi(y)]^T \quad = \quad \pi^t A$$

where $\pi_i = \alpha e^{\alpha(\sum_j a_{ij}x_j)}$.

This fact directs our attention toward the structure of optimal dual solutions to problem **PACK(A,P)**: if $v \in R_+^m$ were optimal dual variables for the constraints $Ax - \lambda e \leq 0$, then an optimal solution to **PACK(A,P)** would also solve the linear program with objective $v^T A$. One of the dual constraints for **PACK(A,P)** (the one arising from the λ column) is that the sum of dual variables equals 1, and therefore a vector π as just described may not be dual feasible, but $\pi(\sum_i \pi_i)^{-1}$ will be. One of the contributions of [SM91] (in the context of **PACK(A,P)**) was to show that, surprisingly, if α is large enough, then whenever $\hat{x}$ is a near-optimal solution to **PACK(A,P)**, the vector $\pi = \pi(x)$ with entries

$$\pi_i = \frac{e^{\alpha \sum_j a_{ij}\hat{x}_j}}{\sum_k e^{\alpha \sum_j a_{kj}\hat{x}_j}} \quad 1 \leq i \leq m \tag{1.37}$$

is near-optimal for the dual of **PACK(A,P)**.

Hence, as a preamble to further study of this point, it is useful to consider the nature of the dual of problem **PACK(A,P)** in more detail. Using standard duality techniques the following result is easily obtained:

LEMMA 1.8

$$\lambda^*_{A,P} = \max_{\pi \geq 0} \frac{\min_{x \in P} \pi^T Ax}{\sum_i \pi_i}. \tag{1.38}$$

Moreover, if $(x^*, \lambda^*_{A,P})$ *is an optimal primal solution to* **PACK(A,P)** *and* $\pi^* \geq 0$ *is the negative of an optimal dual solution to* **PACK(A,P)**, *then*

$$\sum_j a_{ij}x^*_j = \lambda^*_{A,P} \quad \forall i \text{ with } \pi^*_i > 0. \tag{1.39}$$

Further, suppose $P = P^1 \times P^2 \times \cdots \times P^K$, *where each* P^k *is homeomorphic to a simplex. Suppose* $x^*_j > 0$, *where coordinate* j *appears in simplex* P^k. *Let* $C(k)$ *denote the set of all coordinates appearing in simplex* P^k. *Then:*

$$\sum_i \pi^*_i a_{ij} = \min_{h \in C(k)} \sum_i \pi^*_i a_{ih}. \tag{1.40}$$

■

[SM91] presents a proof for the special version of this result arising from the (uniform) concurrent flow problem. Even though Lemma 1.8 is more general, we will look at what duality says in this specific context, as it provides useful insight.

Consider a flow f with maximum congestion λ and suppose we assign to each edge (i, j) a nonnegative length π_{ij}. Clearly,

$$\sum_{ij} \pi_{ij} (\sum_{k} f_{ij}^{k}) \quad \leq \quad \lambda \sum_{ij} \pi_{ij} u_{ij} \tag{1.41}$$

Next, for each pair of vertices s, t let $L^{\pi}(s, t)$ the shortest path length from s to t. Then we also have

$$\sum_{k} L^{\pi}(s^k, t^k) d^k \quad \leq \quad \sum_{ij} \pi_{ij} (\sum_{k} f_{ij}^{k}) \tag{1.42}$$

where s^k is the source node for commodity k, t^k is its destination, and d^k is its demand. Consequently,

$$\frac{\sum_k L^{\pi}(s^k, t^k) d^k}{\sum_{ij} \pi_{ij} u_{ij}} \quad \leq \quad \lambda. \tag{1.43}$$

Lemma 1.8 states that

$$\max_{\pi} \frac{\sum_k L_{\pi}(s^k, t^k) d^k}{\sum_{ij} \pi_{ij} u_{ij}} = \frac{\sum_k L_{\pi^*}(s^k, t^k) d^k}{\sum_{ij} \pi_{ij}^* u_{ij}} \quad = \quad \lambda^* \tag{1.44}$$

for appropriate π^*, where λ^* is the maximum congestion attained using an optimal flow f^*. Further, by (1.39), $\pi_{ij}^* > 0$ only on edges (i, j) with maximum congestion in f^* (i.e., congestion λ^*), and by (1.40), the only paths carrying positive flow in f^* are shortest paths under the metric π^*.

Inequalities (1.43), usually credited to [OK71]. are closely related to the "metric inequalities", which have received a great deal of attention from the combinatorial optimization community, particularly in the context of network design problems (see e.g. [STODA94]). One frequent algorithmic use of these inequalities is that where the left-hand side of (1.43) is strictly greater than 1, proving that the capacities u_{ij} are infeasible for simultaneously routing all the demands.

In this context, we may choose very special length vectors π. A special case is that of "cut" metrics. In a cut metric we set $\pi_{ij} = 1$ for edges (i, j) crossing a cut C in a given direction, and $\pi_{ij} = 0$ otherwise. For such a metric (1.43) simply states that the capacity of C, scaled by λ, must be at least as large as the sum of demands crossing C. Using the technique in [SC86] (page 115), it is possible to show that in the case of a single commodity, equation (1.44) implies the max-flow min-cut theorem, and that the maximum in (1.44) is attained by a cut metric. However, in the general multicommodity case, the reader may wonder how tight a

bound on λ^* is proved by using cut metrics alone. We will return to this topic at the end of this chapter; however, for k commodities the ratio between λ^* and the max min-cut can be as large as $O(\log k)$ [LR98] and this is best possible [LLR94].

Returning to exponential penalties, the following result, generalized to **PACK(A,P)**, is adapted from [SM91].

LEMMA 1.9 *Consider problem* **PACK(A,P)**. *Let* $x \in P$, *and let* $\pi = \pi(x)$ *be defined as in (1.37). Let* $\phi_* = \min_{y \in P} \pi^T A y$. *Then*

$$\lambda_{A,P}(x) - \phi_* \leq \frac{m}{\alpha} + \pi^T A x - \phi_* \tag{1.45}$$

Proof.

Denote by a_i denote the i^{th} row of A, written as a column vector. We have

$$\begin{aligned} \lambda_A(x) - \sum_i \pi_i a_i^T x &= \sum_i (\lambda(x) - a_i^T x)\pi_i \\ &= \sum_{i:\lambda_A(x) \neq a_i^T x} (\lambda(x) - a_i^T x)\pi_i \end{aligned} \tag{1.46}$$

Consider a term appearing in (1.46). For such an index i we have

$$\pi_i \leq \frac{e^{\alpha a_i^T x}}{e^{\alpha \lambda_A(x)}} = \frac{1}{e^{\alpha(\lambda_A(x) - a_i^T x)}} < \frac{1}{\alpha(\lambda_A(x) - a_i^T x)}$$

Therefore

$$\lambda_A(x) < \frac{m}{\alpha} + \sum_i \pi_i a_i^T x. \tag{1.47}$$

■

Since $\phi_* \leq \lambda^*_{A,P}$, this lemma suggests a strategy for an algorithm based on the exponential potential function. Say that we want a solution with (relative) error guarantee ϵ. Then choose $\alpha = \frac{\epsilon}{2m}$, and search for $x \in P$ such that with penalties $\pi = \pi(x)$, we have $\sum_i \pi_i a_i^T x - \min_{y \in P} \pi^T A y \leq \frac{\epsilon}{2}$. The flaw in this argument is that it will only yield the *absolute* error guarantee

$$\lambda_A(x) - \lambda^*_{A,P} \leq \epsilon. \tag{1.48}$$

In other words, problem instances where $\lambda^*_{A,P}$ is very small could prove troublesome. Now throughout this section, we have assumed that $P \in R^n$. Suppose we strengthen this assumption to $P \in R^n_+$, and that $0 \notin P$. In the multicommodity flow setting, the first part of the assumption

always holds, and the second one holds whenever there is at least one nonzero commodity to be routed (and if not **CONG** is trivial). Having made this assumption, we can then assume that for any $x \in P$, $\sum_i \sum_j a_{ij} x_j \geq 1$ – this only requires scaling P by a positive constant, which changes $\lambda_A(x)$, and $\lambda^*_{A,P}$, by the same scale factor, and thus relative errors are unchanged. In [SM91], the authors assume that the sum of demands is 1, which achieves a similar result. Consequently, for any $x \in P$, we will have $\lambda_A(x) \geq \frac{1}{m}$. We will make the above assumption in what follows.

Armed with this assumption, we can now take a second look at equation (1.47). Choose $\alpha = \frac{2m^2}{\epsilon}$. Suppose we could find $x \in P$ with $\sum_i \pi_i a_i^T x - \min_{y \in P} \pi^T A y \leq \frac{\epsilon}{2m}$ where $\pi = \pi(x)$. Then we would have that $\frac{\lambda_A(x) - \lambda^*_{A,P}}{\lambda^*_{A,P}} \leq \epsilon$, as desired. This is (roughly) the main thrust of the algorithm in [SM91].

It is instructive to contrast Lemma 1.9 to Lemma 1.7 – in particular, it appears that the choice $\alpha = \frac{2m^2}{\epsilon}$ may not be the only possible, in that an algorithm that uses $\alpha = \frac{m \log m}{\epsilon}$ and that minimizes Φ with absolute error $\frac{\epsilon}{2m}$ would also yield an $x \in P$ with ϵ-optimal congestion. Later chapters will examine the issue of choosing the "best" α – one critical consideration is that of numerical stability, and the second choice is clearly preferable in this regard, as it gives rise to exponential functions with smaller exponents.

As an aside, we note that the difficulty of approximating problems with "small" optimal value is well-known to researchers. However, in the context of the problems we study here, one can follow a different line of analysis based on equation (1.48).

2.1 The algorithm proper

The algorithm in [SM91] finds an ϵ-optimal solution to the minimum congestion problem **CONG** *with uniform capacities*, i.e., all values u_{ij} are equal. Without loss of generality, $u_{ij} = 1$ for all (i, j).

As before, commodity k will have demand d^k, source node s^k and destination node t^k. One important fact regarding this algorithm is that at any iteration it explicitly stores a list of *active* paths for each commodity – these are the paths currently used to route the commodity. Next we present the algorithm in [SM91] in simplified (and slightly incorrect) form.

Algorithm SM

Step 1. Let f^1 be a multicommodity flow where each commodity is routed using a single path. Let $t = 1$.

Step 2. For each edge (i, j), let

$$\pi_{ij}^t = \frac{e^{\frac{m^2}{\epsilon}\sum_k f_{ij}^{k,t}}}{\sum_{ab} e^{\frac{m^2}{\epsilon}\sum_k f_{ab}^{k,t}}}.$$

Step 3. For each commodity k, let L_t^k denote the length of a shortest path (active or not) from s^k to t^k using metric π^t, and let λ^t denote the maximum congestion attained by flow f^t. If

$$\lambda^t \leq (1+\epsilon)\sum_k L_t^k d^k,$$

then **stop** (f^t is ϵ-optimal).

Step 4. For each commodity k, let P_t^k be a longest active path for commodity k, with length W_t^k, and let Q_t^k be a shortest (active or not) for commodity k, both using metric π^t. Choose a commodity h such that $W_t^h - L_t^h$ maximized.

Step 5. Let d^* denote the sum of lengths of edges in $P_t^h - Q_t^h$, and d_* the sum of lengths of edges in $Q_t^h - P_t^h$. [Note that $W_t^h - L_t^h = d^* - d_*$]. Set $\sigma = \frac{\epsilon}{4m^2}\log(d^*/d_*)$, and $f(P_t^h)$ = current flow allocation to path P_t^h. **Reroute** $\min\left\{\sigma, f(P_t^h)\right\}$ units of flow from P_t^h to Q_t^h. Set $t \leftarrow t+1$, and **go to Step 2.**

Remark. Note that the set of active paths may change in Step 5. For example, path P_t^h may become inactive after the rerouting step.

In what follows, we assume that the sum of all demands is 1 – this simply scales the value of problem **CONG** by a constant factor.

THEOREM 1.10 *Algorithm* **SM** *terminates with an* ϵ*-optimal flow after* $O(m^4/\epsilon^3)$ *iterations and always maintains* $O(m^3/\epsilon^2)$ *active paths; with overall complexity* $O(nm^7/\epsilon^5)$.

Proof. (Sketch). The key idea is to show that the *potential function*

$$\Phi^t = \sum_{ij} e^{\frac{m^2}{\epsilon}\sum_k f_{ij}^{k,t}} \tag{1.49}$$

is decreasing fast enough as a function of t. We have that $\Phi^t \geq m$ for any t, and since the sum of all demands is 1, the sum of flows on any edge never exceeds 1, and in particular $\Phi^1 \leq me^{\frac{m^2}{\epsilon}}$. Consequently, if we can argue that $\Phi^t - \Phi^{t+1}$ is large enough, we will have the desired result. In particular, if we could argue that $(\Phi^t - \Phi^{t+1})/\Phi^t = \Omega(\frac{\epsilon^2}{m^2})$ the number of iterations would be as claimed.

We can make this argument a bit more precise as follows. Consider a particular iteration t. For simplicity we will assume that the amount of flow rerouted in Step 5 is σ. The critical fact to note is that the percentage contribution to the potential function arising from the edges of a particular path p is precisely the length of p, using the current metric.

Now in iteration t, we reroute σ units of flow from a longest path P_t^h, to a shortest path Q_t^h, both being paths corresponding to some commodity h. Consequently, after the rerouting, the contribution to the potential function due to each edge in $P_t^h - Q_t^h$ is decreased by a factor of $e^{-\frac{m^2}{\epsilon}\sigma}$. Similarly, the contribution to the potential function due to each edge in $Q_t^h - P_t^h$ is increased by a factor of $e^{\frac{m^2}{\epsilon}\sigma}$. In all other edges the flow is unchanged, and so will be their contribution to potential. As defined in the algorithm, $P_t^h - Q_t^h$ has length d^* and $Q_t^h - P_t^h$ has length d_*. Thus,

$$\frac{\Phi^t - \Phi^{t+1}}{\Phi^t} = (1 - e^{-\frac{m^2}{\epsilon}\sigma})d^* + (1 - e^{\frac{m^2}{\epsilon}\sigma})d_*.$$

The particular choice of σ made in Step 5, then yields

$$\frac{\Phi^t - \Phi^{t+1}}{\Phi^t} = \left(\frac{d^* - d_*}{d^* + d_*}\right)^2 > \frac{(d^* - d_*)^2}{4} \tag{1.50}$$

since the sum of all edge lengths is at most 1.

Assume by contradiction that $d^* - d_* < \frac{\epsilon}{3m}$. Now $d^* - d_*$ is also equal to $W_t^h - L_t^h$, i.e., the difference in lengths between a longest active path and a shortest overall path for commodity h. But commodity h was chosen in the algorithm precisely so as to maximize this difference. We conclude that, for any commodity k,

$$\sum_{ij} \pi_{ij}^t f_{ij}^k - L_t^k d^k < \frac{\epsilon}{3m} d^k,$$

where as previously d^k is the demand for commodity k. Summing these equations and recalling that the sum of demands equals 1, we will have

that

$$\sum_k \pi_{ij}^t f_{ij}^k - \sum_k L_t^k d^k \quad < \quad \frac{\epsilon}{3m}. \tag{1.51}$$

Now recall Lemma 1.9, and the analysis following the Lemma. The left-hand side of inequality (1.51) is precisely the expression $\pi^T Ax - \phi_*$ in equation (1.45). Thus we will have that $\lambda^t - \sum_k L_t^k d^k = O(\epsilon/m)$ and also, since $\lambda^t \geq \lambda^* \geq m^{-1}$, that $\sum_k L_t^k d^k \geq 2/m$ for ϵ small enough. We conclude that when $d^* - d_* < \frac{\epsilon}{3m}$, algorithm **SM** will correctly terminate in Step 3. ■

As concluding remarks on the Shahrokhi-Matula algorithm, we note that (a) The algorithm is best viewed as a *coordinate descent* method, in the space of the path variables (as opposed to a Frank-Wolfe algorithm), and (b) As was the case with the Flow Deviation method, it may be possible to improve the ϵ^{-3} worst-case iteration count by using a "powers of 2" strategy that gradually improves the approximation quality until the desired ϵ is attained.

2.2 Cut metrics and minimum congestion

We return now to the topic we touched on earlier in this chapter, regarding how good a lower bound on λ^* is obtained by restricting the metric in (1.43) to a cut metric π, that is to say given a set S of vertices $\pi_{ij} = 1$ if and only if $i \in S$ and $j \notin S$. If we denote by $D(S)$ and $C(\delta(S))$, respectively, the total demand leaving S and the sum of capacities of arcs leaving S, then the cut-metric inequality states $\tau(S) \doteq D(S)/C(\delta(S)) \leq \lambda^*$, and we are interested in the quantity

$$\tau^* \quad = \quad \max_{S \subset V} \frac{D(S)}{C(\delta(S))} \tag{1.52}$$

where V is the set of vertices. As mentioned before, the max-flow min-cut theorem implies that in the case of exactly one commodity, $\tau^* = \lambda^*$, as (1.52) states that if we scale all arc capacities by τ^* we can then feasibly route the commodity; but in general $\tau^* < \lambda^*$.

An extreme example for this inequality is provided by *expander* graphs. In this discussion we will deal with undirected graphs (and flows). We refer the reader to [LR98] and references therein for background on these graphs, which, due to their favorable connectivity properties, have been frequently used in the theoretical literature to obtain provably good network designs. For the purposes of this manuscript, given $\alpha > 0$ and $d > 0$, an (α, d)-expander graph is an undirected graph where every subset S of vertices that is not too large has at least $\alpha|S|$ neighbors (this

is the so-called expansion property), and such that every vertex has degree at most d. Expander graphs were originally proved to exist using probabilistic arguments; a technical tour-de-force eventually produced explicit constructions. In any event, for given (selected) d, α, one can construct arbitrarily large (α, d)-expander graphs.

Consider such a graph, with, say, n vertices, and consider the multicommodity flow problem where there is one unit of demand between each pair of vertices. The structure of the graph is seen to imply that the average distance between any two vertices is $\theta(\log n)$. Thus, in any routing, the sum of edge congestions is $\theta(n^2 \log n)$. On the other hand, the total number of edges in the graph is at most $2dn = O(n)$. We conclude $\lambda^* = \Omega(n \log n)$.

At the same time, if we consider a cut separating a set S of vertices from its complement $\bar{S}$, where $|S| \leq |\bar{S}|$, then $D(S) = |S||\bar{S}|$. Further, $C(\delta(S)) \geq \alpha|S|$. Therefore $C(\delta(S))/D(S) = O(|\bar{S}|)$, and $\tau^* = O(n)$, proving an order of magnitude $\log n$ gap between τ^* and λ^* [LR98].

The proof that this bound is best possible also involved some highly sophisticated and elegant mathematics, see [LLR94] and the references cited there. In this case the underlying technical concept is that of embedding a finite metric space into a standard metric space, e.g. R^m for some $m > 0$. In [LLR94] it is proved (heavily relying on previous work by other researchers) that given a set of V n points, a metric defined on V, and a set P consisting of k pairs of elements of V, we can embed V in R^k such that the distance between the two elements of each pair in P is distorted by at most $O(\log k)$. Using this fact, [LLR94] proves that, for an arbitrary k-commodity flow problem, the gap between τ^* and λ^* is at most $O(\log k)$.

2.2.1 Cut metrics and capacitated network design

Capacitated network design problems arise in many practical applications, especially in telecommunications. See, for example [AMW98], [Ba96], [BCGT96], [MMV91]. In its simplest form (the so-called "network loading problem") we are given a network and a set of multicommodity demands to be routed. We have to assign, at minimum cost, integer capacities on the edges of the network, for which a feasible routing exists. In the case of a directed network, this problem may be formulated as:

$$\begin{aligned} \min \quad & \textstyle\sum_{ij} c_{ij} x_{ij} \\ \text{s.t.} \quad & \\ \sum_k f_{ij}^k - x_{ij} \quad & \leq \quad 0 \quad \forall\, (i,j) \end{aligned} \tag{1.53}$$

$$N^k f^k = b^k, \quad k = 1, \cdots, K \tag{1.54}$$

$$0 \leq f^k \qquad k = 1, \cdots, K \tag{1.55}$$

$$x_{ij} \geq 0 \text{ and integral } \forall (i,j) \tag{1.56}$$

These mixed-integer programs tend to be quite difficult, due primarily to two reasons: the poor quality of their continuous relaxation, and the difficulty in solving the continuous relaxation itself. The reader may notice that for fixed x we essentially recover the feasibility problem **CONG**.

To address the first concern, various polyhedral approaches have been proposed. The most common type of inequality that can be used to strengthen the above formulation is the cutset inequality

$$x(\delta(S)) \geq \lceil D(S) \rceil, \tag{1.57}$$

where S is some some set of vertices, and $D(S)$, $x(\delta(S))$ have the same meaning as above. These can be generalized to k-partition inequalities for $k > 2$.

Putting aside the issue of separating such inequalities (it can be done effectively), and of the quality of the strengthened formulation (it is good) one must next face the more significant obstacle of how to solve the continuous relaxation. For problem instances of realistic size (say, 500 nodes) this difficulty can be insurmountable, at least by standard methods.

To avoid this problem, some researchers (e.g. [STODA94]) have proposed that one should work on the *projection* of the above formulation to the space of the x variables. This gives rise to an alternative algorithm where at any time one has a set of constraints (involving the x variables only) that are known to be valid. This integer program is solved (or nearly solved) and if the resulting solution $\hat{x}$ is feasible for the multicommodity flow problem with the given demands, we are done. Otherwise (i.e., $\hat{x}$ is infeasible) we find a metric inequality that is violated by $\hat{x}$, we add this inequality to the working formulation and repeat the process.

In its simplest form, this alternative algorithm just uses cutset inequalities in order to enforce feasibility. In essence, the problem then being solved is:

$$\begin{array}{ll} \min & \sum_{ij} c_{ij} x_{ij} \\ \text{s.t.} & \end{array}$$

$$x(\delta(S)) \geq \lceil D(S) \rceil \ \forall\, S \tag{1.58}$$

$$x_{ij} \geq 0 \text{ and integral } \forall (i,j) \tag{1.59}$$

The advantages of using a formulation based on the x variables only is clear: the linear programs are incomparably easier to solve than those using the full multicommodity flow formulation given above. At the same time, we have to worry whether we have jumped from the frying pan and into the fire: in trying to avoid a difficult linear program, we may have made the integer programming component of the problem of the problem even more difficult.

In fact, the expander graph example given in the previous section shows that formulation (1.58 - 1.59), including the integral restriction, can be a factor of $\Omega(\log n)$ smaller than the value of the linear programming relaxation of (1.53 - 1.56). Of course, (1.58 - 1.59) could not in general even guarantee feasibility for the multicommodity flow problem, and one would need to use more general metric inequalities, but the same expander graph example can be used to show that using k-partition inequalities with k bounded still yields a bound that is $\Omega(\log n)$ away from the true linear programming value.

As we will argue later in this manuscript, solving the formulation with constraints (1.53 - 1.56) should *not* be very difficult, at least to reasonable accuracy, and perhaps this will yield a more effective solution procedure for the network loading problem (see [B96]).

Chapter 2

THE EXPONENTIAL POTENTIAL FUNCTION – KEY IDEAS

In this chapter we review some of the fundamental results on the exponential potential function methods that were developed through the middle '90s. These results substantially built on, and were motivated by, the work of Shahrokhi and Matula described in Chapter 1. We will concentrate on the work by Grigoriadis and Khachiyan [GK94], and Plotkin, Shmoys and Tardos [PST91]. More recent results will be covered in Chapter 3.

The developments in [GK94] and [PST91] were among many extending and improving on the Shahrokhi-Matula approach. Also see [KPST90] and [LMPSTT91]. An unrelated approach is described in [S91]. In essence, the new algorithms were Frank-Wolfe algorithms, using exponential potential functions to penalize infeasibilities, to find approximately feasible solutions to systems of inequalities. The contribution from this work can be classified in three categories, which we outline below and which will be examined in more detailed form later.

0.2.2 Handling more general problems

In previous sections we saw algorithms that while designed to handle the concurrent flow problem, actually contained fairly little explicit combinatorial elements. In fact, we saw that several of the theorems could easily be generalized to problem **PACK(A,P)**.

Thus, we are drawn to consider problems of the following general form: find a vector $x \in R^n$ satisfying:

$$Ax \leq b \tag{2.1}$$

$$x \in P \tag{2.2}$$

where A is a $m \times n$ matrix, b is an m-vector, and P is a polyhedron over which it is "easy" to solve linear programs. Our relaxed objective is to find $x \in P$ with small relative violation of the *linking* constraints $Ax \leq b$. More precisely: given $\epsilon > 0$ we seek $x \in P$ with $Ax \leq (1+\epsilon)b$. Such an x will be called *ϵ-feasible*. For now we assume $b_i > 0$ for all i; in the special case where $A \geq 0$, $b > 0$, we obtain problem **PACK(A,P)**.

In addition, we are of course interested in approximately *optimization* problems of the form min $\{c^T x : x$ satisfies (2.1) – (2.2)$\}$. As we will see, the techniques used to handle pure feasibility systems extend to optimization problems through the simple trick of adding a "budget" constraint.

0.2.3 Handling large width

It is well-known among practitioners of Lagrangian relaxation that, whenever possible, a Linear Programming formulation should employ "small" or "tight" upper bounds on variables – otherwise slow or no convergence may be experienced.

It is possible to make this more precise in the context of algorithms, such as the ones studied here, that make use of potential/penalty functions to achieve feasibility. This can be motivated by a simple example. Suppose we want to obtain a feasible solution to the system:

$$\begin{array}{rcrcrcl} x_1 & + & 10x_2 & & & \geq & 7 \\ & & x_2 & + x_3 + & x_4 & \geq & 1 \\ 2x_1 & + & x_2 & & & \leq & 2 \\ & & & & x & \in & P \end{array} \tag{2.3}$$

where

$$P = \begin{cases} 0 \leq x_1 \leq 10^6, & 0 \leq x_2 \leq 1, \\ 0 \leq x_3 \leq 1, & 0 \leq x_4 \leq 1, \end{cases} \tag{2.4}$$

Suppose that we are at the point $(1.0, 0.0001, 0.6, 0.3)^T$. At this point, the (relative) infeasibility for the first constraint of system (2.3) is approximately 85.7%, for the second constraint it is 9.99%, and for the third it is 0.005%. Consequently, the gradient of the potential function will be strongly dominated by the contribution from the first constraint, i.e., we would expect the gradient to be proportional to a vector of the form $(1.0 - \epsilon_1, 10 - \epsilon_2, -\epsilon_3, -\epsilon_4)^T$, where the ϵ_i are very small positive values. As a result, the step direction will be the vector $(10^6, 1, 1, 1)^T$, and therefore, the step size will be of the order of 10^{-6} (otherwise the violation of the third constraint will become much larger than $O(1)$). The net result is that none of the violations improves by much (actually,

that of the third constraint worsens), and we expect that a large number of iterations will be required to make each of the violations smaller than, say, 1%.

On the other hand, in the above example the set P could have been strengthened since the third constraint implies $x_1 \leq 1$. Using the stronger representation for P, it is clear that if we use any reasonable potential function the algorithm will quickly converge to a point with small infeasibilities.

Returning to the original system (2.3), we observe that the slow convergence is caused by the existence of points $x \in P$ which substantially violate one of the linking constraints. In terms of the general problem (2.1)-(2.2) this motivates the following definition:

Definition: The *width* of the polyhedron $\{x \in R^n \;\; Ax \leq b\}$ with $b > 0$, with respect to P equals

$$\rho = \max_{x \in P} \left\{ \max_i \frac{a_i^T x - b_i}{b_i} \right\} \tag{2.5}$$

In the case of the system (1.1)-(1.3), the width equals the maximum congestion incurred by any flow, minus one. Sometimes we may wish to use $1 + \rho$ as the width; this is not a critical difference in the sense that width is important when it is large.

The width parameter was formally introduced by Plotkin, Shmoys and Tardos in [PST91]. They also provided a basic algorithm which, under appropriate assumptions on A, b and P, given $\epsilon > 0$ either proves (2.1)-(2.2) is infeasible or finds an ϵ-feasible $\hat{x} \in P$. Their algorithm uses the exponential potential function and solves linear programs over P, following the broad format described above. The worst-case number of iterations of their basic algorithm depends polynomially on m and n, and is also proportional to $\rho\epsilon^{-3}$. We will examine this algorithm in more detail later. In subsequent work extending this line of research, the dependence on ρ, ϵ was improved to $\rho\epsilon^{-2}$.

One wonders whether this dependence can be improved upon, especially given that ρ could be quite large. Klein and Young [KY98] have constructed examples of **PACK(A,P)** where *any* Frank-Wolfe algorithm *must* take $\Omega(\rho\epsilon^{-2})$ iterations. Their examples are limited to the range $\rho\epsilon^{-2} = O(n^{1/2})$. More significantly, their result only applies to algorithms that access P solely through an oracle that solves linear programs over P (in particular, their result does not apply to algorithms that dynamically *change* P). This work will be analyzed in more detail in Chapter 3.

Grigoriadis and Khachiyan [GK94] also provided an algorithm that finds an ϵ-approximate solution to (a generalization of) **PACK(A,P)**, also by following the generic exponential potential function framework. However, critically, the algorithm in [GK94], as well as the more elaborate algorithm in [PST91], both modify P, thus bypassing the Klein-Young assumptions – one way to view their modification is that individual variable bounds are changed. As a result, the complexity of these algorithms does not depend on ρ, and up to polynomial factors in the size of the problems, is $O(\epsilon^{-2})$.

It is important to recognize the impact of large width on stepsize, when using a Frank-Wolfe algorithm: large width implies small stepsize. Conversely, as we will see, techniques that reduce width immediately guarantee a corresponding increase in stepsize, resulting on proved faster convergence. From a practical standpoint, large width can give rise to stepsizes that are numerically indistinguishable from zero, resulting in aborted convergence (so-called "jamming").

0.2.4 Leveraging block-angular structure

Consider a case of problem (2.1)-(2.2) which is block-angular, i.e. $P = P^1 \times P^2 \times \cdots \times P^K$, where each P^i is contained in a disjoint space $R^{n(i)}$ as discussed before. Each Frank-Wolfe step reduces to solving K independent linear programs, one for each P^j, but one wonders if there is a more fundamental strategy for taking advantage of the block structure. We will discuss deterministic and randomized strategies that solve only one block (on the average) per iteration, resulting in provably faster algorithms. In addition, these techniques can be viewed as natural strategies for handling large width problems.

1. A basic algorithm for min-max LPs

In this section we present an algorithm that finds ϵ-approximate solutions to **PACK(A,P).** Our description largely follows that in [GK94], although the algorithm in [PST91] is quite similar. In addition, our description temporarily puts aside the issue of how to take advantage of possible "block-angular" structure of P. Also recall that $A \geq 0$.

We assume that A has m rows, n columns and a maximum of Q nonzeros per row.

The algorithm in [GK94] can be seen as operating in two phases. In the first phase we find a lower bound $\lambda^L_{A,P}$ on $\lambda^*_{A,P}$, and also a point $x \in P$ such that $\lambda^U_{A,P} \doteq \lambda_A(x)$ satisfies $\lambda^U_{A,P} \leq O(\min\{m, Q\})\lambda^L_{A,P}$. Thus, we are guaranteed a polynomial multiplicative error in our initial estimate of $\lambda^*_{A,P}$. In the second stage we progressively decrease the gap

between the lower and upper bounds on $\lambda^*_{A,P}$ until the desired accuracy ϵ is achieved. We will show that the overall procedure executes $O(\frac{Q \log m}{\epsilon^2} + Q \log m \ \min\{\log Q, \log m\})$ Frank-Wolfe steps in order to estimate $\lambda^*_{A,P}$ to relative error of at most ϵ.

For many problem classes, e.g. routing problems, this approach yields much better worst-case complexity bounds than competing algorithms when ϵ is not too small (i.e. ϵ fixed as a function of m and n). For example, consider a maximum concurrent flow problem on a graph with N nodes and M arcs, and with N commodities (that the number of commodities is at most N can always be assumed through aggregation). This yields a problem **PACK(A,P)** with $m = M$ and $Q = N$. In this case, the subproblem we must solve in a Frank-Wolfe iteration breaks up into N separate minimum-cost flow problems. Using Orlin's algorithm [OR88] we can solve these in time $O((M \log N)(M + N \log N))$. For the sake of simplicity assume that the graph is dense enough that $M > N \log N$. Then our overall approach will have worst-case complexity

$$O\left(NM^2 \log N \left(\frac{N \log N}{\epsilon^2} + N \log^2 N\right)\right) = O((N^2 M^2 \log^2 N)(\epsilon^{-2} + \log N)). \tag{2.6}$$

In contrast, a direct application of Vaidya's algorithm for linear programming [V90] to **PACK(A,P)** (which has $NM + 1$ variables and roughly $N^2 + M$ constraints) produces a bound of $O(L(NM)^3)$, where L is the maximum number of bits in a solution to **PACK(A,P)**. This is almost a factor LNM worse than (2.6) for fixed ϵ. However, Vaidya's algorithm is a polynomial-time algorithm, and the bound (2.6) is not polynomial – it depends exponentially in ϵ. See [PST91] for other comparisons.

1.1 The first stage

In order to motivate the strategy employed by the algorithm in the first stage, suppose we had a known upper bound $\hat{\lambda}$ on $\lambda^*_{A,P}$. Since $A \geq 0$, it follows that

$$P(\hat{\lambda} v^A) \doteq \left\{x \in P : x_j \leq \hat{\lambda} v^A_j \ \forall j\right\} \tag{2.7}$$

is nonempty, where for $1 \leq j \leq n$

$$v^A_j = \min_i a^{-1}_{ij}. \tag{2.8}$$

Conversely, if we had a value $\gamma > 0$ such that $P(\gamma v^A) \neq \emptyset$, then we would know that $\lambda^*_{A,P} \leq Q\gamma$. In summary, if $\gamma > 0$ is such that $P(2\gamma v^A) \neq \emptyset$, but $P(\gamma v^A) = \emptyset$, then $\gamma < \lambda^*_{A,P} \leq 2Q\gamma$.

These observations suggest a simple algorithm. Start with a point $\bar{x} \in P$, and initially set $\gamma = \lambda_A(\bar{x})$. Then $P(\gamma v^A)$ is nonempty; suppose we were to repeatedly reduce γ by a factor of 2 until the first time that $P(\gamma v^A)$ becomes empty. The number of iterations needed to reach this point is

$$O(\log(\frac{\lambda_A(\bar{x})}{\gamma})) \quad = \quad O(\log(\frac{\lambda_A(\bar{x})}{\lambda^*_{A,P}})).$$

In particular, suppose our starting point $\bar{x}$ is the optimal solution to the linear program $\min_{x \in P} e^T Ax$. Then, clearly, $\lambda_A(\bar{x}) \leq m\lambda^*_{A,P}$, and we conclude that the first-phase algorithm requires $O(\log m)$ iterations, and the resulting vector $\bar{x}$ yields a $O(\min\{m, Q\})$-approximation on $\lambda^*_{A,P}$.

1.2 The second stage

In order to compute an ϵ-approximation to $\lambda^*_{A,P}$, the second stage runs a binary search procedure. This procedure improves at each iteration the gap between the upper bound and the lower bound on $\lambda^*_{A,P}$ by a factor of **2/3,** until the gap is reduced to at most the desired accuracy ϵ.

Step 0. Let $\lambda^L_{A,P}$, $\lambda^U_{A,P}$ be the bounds on $\lambda^*_{A,P}$ produced by the first stage, and let $\hat{x}_0 \in P$ attain $\lambda_A(\hat{x}_0) = \lambda^U_{A,P}$. Set $k = 0$.

Step 1. Using the algorithm described in the next section, find $y \in P$ with $\lambda_A(y) \leq \lambda^*_{A,P} + (\lambda^U_{A,P} - \lambda^L_{A,P})/3$.

Step 2. If $\lambda_A(y) > \lambda^L_{A,P} + 2(\lambda^U_{A,P} - \lambda^L_{A,P})/3$, then reset $\lambda^L_{A,P} \leftarrow \lambda^L_{A,P} + (\lambda^U_{A,P} - \lambda^L_{A,P})/3$, an improved lower bound on $\lambda^*_{A,P}$.

Step 3. Otherwise, $\lambda_A(y) \leq \lambda^L_{A,P} + 2(\lambda^U_{A,P} - \lambda^L_{A,P})/3$ is an upper bound on $\lambda^*_{A,P}$, and we reset $\lambda^U_{A,P} \leftarrow \lambda_A(y)$.

Step 4. If $\lambda^U_{A,P} \leq (1+\epsilon)\lambda^L_{A,P}$, stop. Else, reset $k \leftarrow k+1$, $\hat{x}_k \leftarrow y$, and go to Step 1.

Modulo the algorithm used in Step 1, the validity of this procedure is clear.

In the next section we describe the algorithm used in [GK94] to carry out Step 1. Here we will first analyze the overall workload we incur, to show that it has the desired dependency on ϵ, m and n.

As we will show in the next section, the algorithm for Step 1 requires at most

$$O\left(Q \log m \left(\frac{\lambda^U_{A,P}}{\lambda^U_{A,P} - \lambda^L_{A,P}}\right)^2\right) \tag{2.9}$$

Frank-Wolfe steps over sets of the form $P(\gamma v^A)$, for appropriate $\gamma > 0$.

To bound the workload resulting from all the executions of Step 1, consider the ratio $\frac{\lambda^U_{A,P}}{\lambda^U_{A,P}-\lambda^L_{A,P}}$. Each execution of Step 1 reduces the denominator by a factor of 2/3, and the numerator does not increase. Consider first the set of iterations, at the start of the procedure, during which $\lambda^L_{A,P} = 0$ and as a consequence the ratio equals 1. Since each such iteration (except the last) reduces $\lambda^U_{A,P}$ by at least a factor of 2/3, and since the initial vector $\hat{x}_0$ in the execution of Step 0 is guaranteed to satisfy $\lambda_{A,P}(\hat{x}_0) \leq O(\min(m, Q)\lambda^*_{A,P})$, the total number of iterations with $\lambda^L_{A,P} = 0$ is at most $O(\min\{\log Q, \log m\})$.

To account for the remaining iterations, note that the first time that $\lambda^L_{A,P} > 0$ we have $\frac{\lambda^U_{A,P}}{\lambda^U_{A,P}-\lambda^L_{A,P}} = O(1)$. As a result, the number of Frank-Wolfe steps resulting from iterations with $\lambda^L_{A,P} > 0$ is

$$O\left(Q \log m \sum_{i=0}^{i=T} \left(\frac{3}{2}\right)^i\right) \tag{2.10}$$

where T is the smallest positive integer such that $(\frac{2}{3})^T \leq \epsilon$. The sum in (2.10) is dominated by the last term, and thus the expression in (2.10) is bounded by $O(\frac{Q \log m}{\epsilon^2})$.

In summary, the total number of Frank-Wolfe steps resulting from iterations with $\lambda^L_{A,P} = 0$ or $\lambda^L_{A,P} > 0$ is $O(\frac{Q \log m}{\epsilon^2} + Q \log m \ \min\{\log Q, \log m\})$, as desired.

Next we describe how to carry out Step 1.

1.3 Computing λ^* to absolute tolerance

Consider one of the iterations of Step 1. Given the current iterate $\hat{x}$, and the current bounds $\lambda^L_{A,P}$, $\lambda^U_{A,P} = \lambda_A(\hat{x})$, we seek $y \in P$ such that

$$Ay \leq \left(\lambda^*_{A,P} + \frac{\lambda^U_{A,P} - \lambda^L_{A,P}}{3}\right) e.$$

Multiplying this equation by $\frac{1}{\lambda_A(\hat{x})}$, and writing

$$M = \frac{1}{\lambda_A(\hat{x})} A$$

$$\sigma = \frac{\lambda^U_{A,P} - \lambda^L_{A,P}}{3\lambda_A(\hat{x})}$$

our objective becomes that of finding a vector y such that

$$\lambda_M(y) \leq \lambda^*_{M,P} + \sigma \tag{2.11}$$

Further, $1/3 - \lambda^*_{M,P}/3 \leq \sigma \leq 1/3$, and we start with a vector $\hat{x}$ with $1 \geq \lambda_M(\hat{x}) \ (\geq \lambda^*_{M,P})$.

In the remainder of this section, we describe an algorithm that, given a convex set P, an arbitrary nonnegative matrix M, a vector $\hat{x} \in P$ such that $\lambda_M(\hat{x}) \leq 1$ (and thus, $\lambda^* \doteq \lambda^*_{M,P} \leq 1$), and a value σ with $1/3 - \lambda^*_{M,P}/3 \leq \sigma \leq 1/3$, computes a point $y \in P$ such that $\lambda_M(y) \leq \lambda^* + \sigma$, in at most $O(\frac{Q \log m}{\sigma^2})$ Frank-Wolfe iterations. As applied to the particular problem we are interested in, the bound we obtain is therefore $O\left(Q \log m \left(\frac{\lambda^U_{A,P}}{\lambda^U_{A,P} - \lambda^L_{A,P}}\right)^2\right)$ iterations, as desired.

Proceeding as in Section 1.1, since $\lambda^* \leq 1$ we will replace the set P with $P(v^M)$ (see (2.7), (2.8)). In what follows, we will use the notation $\lambda(x)$ to refer to $\lambda_M(x)$.

To carry out the task at hand, [GK94] uses the potential function

$$\Phi(x) = \ln(\sum_i e^{\alpha m_i^T x}), \tag{2.12}$$

for appropriate $\alpha > 0$, where m_i^T denotes the i^{th} row of M. The reader may wonder what theoretical significance the logarithm has, as opposed to using a straight exponential function. In fact, it has none. At any given point x, the gradient of (2.12) and that of $\sum_i e^{\alpha m_i^T x}$, are parallel, and therefore a Frank-Wolfe algorithm will behave identically when using either function, for any given choice of step sizes. However, the logarithm simplifies the counting arguments and the presentation of the algorithm. One nice feature that the potential function (2.12) satisfies is that its gradient equals

$$\nabla\Phi(x) = \alpha p^T M,$$

where

$$p_i = \frac{e^{\alpha m_i^T x}}{\sum_k e^{\alpha m_k^T x}}, \tag{2.13}$$

which is a dual-feasible vector for **PACK(A,P).**

In addition, $\Phi(x)$ is convex. A quick proof of this fact is obtained by showing that the function $f : R^m \to R$ defined by $f(v) = \ln(e_1^v + e_2^v + \cdots + e_m^v)$ is convex. This reduces to proving that, given numbers $a_i \geq 0$,

$i = 1, \cdots, m$, the matrix H defined by

$$\begin{aligned} h_{ii} &= a_i \sum_{k \neq i} a_k, \quad 1 \leq i \leq m, \\ h_{ij} &= -a_i a_j, \quad 1 \leq i \neq j \leq m, \end{aligned} \tag{2.14}$$

is positive-semidefinite. This is proved by observing that, if we apply one iteration of the Cholesky factorization algorithm [GV] to H, we obtain a matrix of the same general form as H, and that for $m = 2$, H is positive-semidefinite. A different proof is obtained by appealing to the Gershgorin circle theorem [GV].

In setting α, we are already gave a preview in Lemma 1.7. We are also inspired by Lemma 1.8, Lemma 1.9, and the discussion in Section 2 just prior to Section 2.1. The following result (with a simple proof) is proved in [GK94].

LEMMA 2.1 *Let* $x \in P$. *Then*

$$\alpha\lambda(x) \leq \Phi(x) \leq \alpha\lambda(x) + \ln m. \tag{2.15}$$

Consequently

$$\alpha\lambda^* \leq \Phi^* \leq \alpha\lambda^* + \ln m. \tag{2.16}$$

■

Corollary. Suppose $\alpha = (\delta + \ln m)/\sigma$ for some $\delta > 0$. Then if $x \in P$ satisfies $\Phi(x) \leq \Phi^* + \delta$, we have $\lambda(x) \leq \lambda^* + \sigma$.

The immediate consequence of this lemma and its corollary is that we should seek to minimize $\Phi(x)$ to some absolute tolerance. Of course, for a given σ, we have that δ and α are interrelated; we must pick these quantities so as to optimize the running time of our algorithm.

The following result, which is a strengthening of Lemma 1.9 (and with similar proof) is given in [GK94]:

LEMMA 2.2 *Let* $x \in P$, *and* p_i *be defined as in (2.13). Then*

$$\lambda_{A,P}(x) \leq \frac{\ln m}{\alpha} + p^T M x \tag{2.17}$$

■

This Lemma, together with Lemma 2.1 shows that, at optimality, the linearization $p^T M x$ of Φ is within an additive error $O(1 \text{n } m)$ of Φ^*. This (again) argues in favor of a Frank-Wolfe algorithm for approximately minimizing Φ, and in favor of choosing $\delta = \theta(\ln m)$.

A more precise argument supporting this choice will emerge from an analysis of the algorithm used in [GK94]. We first present this algorithm in outline, assuming a specific choice for δ and $\alpha = (\delta + \ln m)/\sigma$.

Algorithm GK (outline)

Step 0. Set $x^0 = \hat{x}$. Let $t = 0$.

Step 1. Let y^t be the optimal solution to $\min_{y \in P(v^M)} [\nabla \Phi(x^t)]^T y$.

Step 2. For appropriate $0 \leq \tau^t \leq 1$, set $x^{t+1} = (1 - \tau^t)x^t + \tau^t y^t$.

Step 3. If $\Phi(x^t) - \Phi(x^{t+1})$ is small enough, stop.

Step 4. Otherwise, reset $t \leftarrow t + 1$ and go to 1.

Note that Step 2 implies $x^t \in P(v^M)$ for all t. In order to give this algorithm substance, we must provide a choice of step size τ^t in Step 2, and the termination criterion in Step 3 must be made precise. Suppose that our choice of stepsize were such that we know a constant $\Delta > 0$, so that at each iteration t,

- **(GK.a)** If $\Phi(x^t) > \Phi^* + \delta$, then $\Phi(x^{t+1}) - \Phi^* \leq (1 - \frac{\Delta}{\delta})(\Phi(x^t) - \Phi^*)$.
- **(GK.b)** Otherwise, $\Phi(x^t) - \Phi(x^{t+1}) \geq \Delta$.

Then, first of all, we will make Step 3 (and thus, the overall algorithm) correct by setting

Step 3'. $\Phi(x^t) - \Phi(x^{t+1}) < \Delta$, stop.

as the termination condition, formalizing Step 3. Assuming that a choice for Δ can be made so that **(GK.a)** and **(GK.b)** hold, there remains to estimate the number of iterations needed until the termination condition in Step 3' is satisfied. To bound this number, we consider two kinds of iterations:

(I) Those satisfying $\Phi(x^t) > \Phi^* + \delta$, and

(II) Those satisfying $\Phi(x^t) \leq \Phi^* + \delta$, but for which the termination condition $\Phi(x^t) - \Phi^{x^{t+1}} < \Delta$ is not yet satisfied.

Clearly, the number of iterations of type (II) is at most

$$\frac{\delta}{\Delta}. \tag{2.18}$$

Iterations of type (I) are governed by rule **(GK.a)**, and therefore they number at most

$$O(\tfrac{\delta}{\Delta} \log(\tfrac{\Phi(x^0) - \Phi^*}{\delta})).$$

By appealing to (2.15)-(2.16), we have that $\Phi(x^0) - \Phi^* \leq \alpha(\lambda(x^0) - \lambda^*) + \log m$, and therefore the number of iterations of type (I) is at most

$$O(\tfrac{\delta}{\Delta} \log(\alpha \tfrac{\lambda(x^0)-\lambda^*}{\delta} + \tfrac{\log m}{\delta})).$$

Recall that $\alpha = (\delta + \log m)/\sigma$. Also, by assumption $\sigma \geq \frac{1-\lambda^*}{3}$, and $x^0 = \hat{x}$, satisfying $\lambda(\hat{x}) = 1$. So $\alpha(\lambda(x^0) - \lambda^*) = O(\delta + \log m)$. Thus, the number of type (I) iterations is at most

$$O\left(\frac{\delta}{\Delta} \log\left(\frac{\log m}{\delta} + O(1)\right)\right). \tag{2.19}$$

We would like to finesse the choice of δ so that the maximum of (2.19) and (2.18) is minimized. For given Δ, clearly we want to keep δ as small as possible. But if $\delta = o(\log m)$ then (2.19) dominates.

Hence, it appears sensible to choose $\delta = \theta(\log m)$. With this choice, both (2.19) and (2.18) are $O(\frac{\log m}{\Delta})$. Now we can make the above algorithm more precise:

- We set $\delta = 5 \log m$, and $\alpha = \frac{6 \log m}{\sigma}$,
- In Step 2, the stepsize is

$$\tau = \frac{\sigma}{4Q\alpha} \tag{2.20}$$

$$= \frac{\sigma^2}{24Q \log m} \tag{2.21}$$

  (independent of the iteration),

- (Step 3') The algorithm terminates as soon as an iteration fails to decrease the potential by at least $\Delta = O(\frac{\sigma^2}{Q})$
- As a consequence, the overall number of iterations will be $O(\frac{Q \log m}{\sigma^2})$, as desired.

There remains to show:

THEOREM 2.3 *With the choice of parameters just indicated, algorithm* **GK** *satisfies conditions* **(GK.a)** *and* **(GK.b).**

Proof. Consider an iteration t. Let $\Phi^t = \Phi(x^t)$ denote the current potential. Let y^t be the step direction in Step 1. Then

$$\Phi^{t+1} - \Phi^t = \ln \frac{\sum_i e^{\alpha m_i^T((1-\tau)x^t + \tau y^t)}}{\sum_i e^{\alpha m_i^T x^t}} \tag{2.22}$$

$$= \ln \sum_i \pi_i e^{\alpha\tau m_i^T (y^t - x^t)}, \quad \text{where} \tag{2.23}$$

$$\pi_i = \frac{e^{\alpha m_i^T x^t}}{\sum_j e^{\alpha m_j^T x^t}}, \quad 1 \le i \le m \tag{2.24}$$

(thus $\pi = \pi(x^t)$). For $u \ge 0$, $\ln u \le u - 1$. Also note that $\sum_i \pi_i = 1$. Thus,

$$\Phi^{t+1} - \Phi^t \le \sum_i \pi_i (e^{\alpha\tau m_i^T (y^t - x^t)} - 1) \tag{2.25}$$

Since x^t, $y^t \in P(v^M)$,

$$m_i^T (y^t - x^t) \le Q. \tag{2.26}$$

[Parenthetically, this equation exemplifies how "width" is controlled by the **GK** algorithm. More on this at the end of this chapter. The upper bound Q is overly conservative in many special cases.] Therefore, by (2.20), $\alpha\tau m_i^T (y^t - x^t) \le \sigma \le 1/3$. For $z \le 1/3$, $e^z \le z + \frac{3}{4} z^2$, and we obtain:

$$\Phi^{t+1} - \Phi^t \le \sum_i (\alpha\tau\pi_i m_i^T (y^t - x^t) + \frac{3}{4}\pi_i \alpha^2 \tau^2 [m_i^T (y^t - x^t)]^2). \tag{2.27}$$

Since $[m_i^T (y^t - x^t)]^2 \le (m_i^T y^t)^2 + (m_i^T x^t)^2 \le Q(m_i^T y^t + m_i^T x^t)$, we can simplify this equation to

$$\Phi^{t+1} - \Phi^t \le \alpha\tau(1 - \frac{3}{4} Q\alpha\tau)\pi^T M(y^t - x^t) + \frac{3}{2} Q\alpha^2\tau^2 \pi^T M y^t \tag{2.28}$$

By construction in Step 1, y^t minimizes the inner product with $[\nabla\Phi(x^t)]^T = \alpha\pi^T M$, over $P(v^M)$. In particular, $\pi^T M y^t \le \pi^T M x^0 \le \sum_i \pi_i = 1$. This fact, together with $\tau = \sigma/(4Q\alpha)$, implies that the last term in (2.28) is at most $\frac{3\sigma^2}{32Q}$. Hence

$$\Phi^{t+1} - \Phi^t \le \alpha\tau(1 - \frac{3}{4} Q\alpha\tau)\pi^T M(y^t - x^t) + \frac{3\sigma^2}{32Q}. \tag{2.29}$$

Consider now the first term in the right-hand side of (2.29). Since Φ is convex we have

$$\Phi^t + \alpha\pi^T M(y^t - x^t) \le \Phi^*, \tag{2.30}$$

where Φ^* denotes the minimum potential over $P(v^M)$. Thus,

$$\begin{aligned}\Phi^{t+1} - \Phi^t &\leq -\tau(1 - \frac{3}{4}Q\alpha\tau)(\Phi^t - \Phi^*) + \frac{3\sigma^2}{32Q} \\ &= -\frac{\sigma^2}{24Q\ln m}(1 - \frac{3\sigma}{16})(\Phi^t - \Phi^*) + \frac{3\sigma^2}{2Q} \\ &\leq -\frac{15\sigma^2}{384Q\ln m}(\Phi^t - \Phi^*) + \frac{3\sigma^2}{32Q} \qquad (2.31)\end{aligned}$$

since $\sigma \leq 1/3$. Suppose $\Phi^t - \Phi^* > \delta = 5\ln m$. Then, by (2.31), the potential decrease in iteration t is at least $(\frac{75}{384} - \frac{3}{32})\frac{\sigma^2}{Q} > \frac{\sigma^2}{10Q}$. Consequently, Step 3' of algorithm **GK** is correct: if the algorithm terminates we must have $\Phi^t - \Phi^* < 5\ln m = \delta$. More to the point, the algorithm satisfies property **(GK.b)**.

On the other hand, if $\Phi^t - \Phi^* > 5\ln m$, the right-hand side of (2.31) is less than

$$(-\frac{15\sigma^2}{384Q\ln m} + \frac{3\sigma^2}{160Q\ln m})(\Phi^t - \Phi^*) \quad < \quad -\frac{\sigma^2}{50Q\ln m}(\Phi^t - \Phi^*),$$

which together with (2.31) implies

$$\begin{aligned}\Phi^{t+1} - \Phi^* &\leq (1 - \frac{\sigma^2}{50Q\ln m})(\Phi^t - \Phi^*) \qquad (2.32) \\ &= (1 - \frac{\Delta}{\delta})(\Phi^t - \Phi^*) \qquad (2.33)\end{aligned}$$

This is precisely condition **(GQ.a)**, as desired. ■

2. Round-robin and randomized schemes for block-angular problems

Suppose we have a block-angular problem, i.e. $P = P^1 \times P^2 \times \cdots \times P^K$, where each $P^i \subseteq R^{n(i)}$ using the same notation as before. The algorithm discussed in the previous section will of course work, but can we do better than observe that each iteration consists of K separate "small" linear programs?

As it turns out, the block-angular case not only gives rise to "easier" subproblems, but the algorithmic techniques that we will present put a sharper relief on, and take better advantage of, the relationship between width, stepsize, and convergence rates, leading to better iteration counts.

As a starting point for the discussion, note that since one iteration of algorithm **GK**, $x^t \to x^{t+1} = (1-\tau)x^t + \tau y^t$, can be viewed as a sequence of K block steps (Frank-Wolfe steps restricted to a block each), the decrease in potential $\Phi(x^t) - \Phi(x^{t+1})$ could be accounted for incrementally.

However, in the above algorithm the steps for blocks $2, 3, \ldots, K$ all use the somewhat "stale" $\nabla\Phi$ computed at the starting point, and we should instead "refresh" the gradient after each block step.

More precisely, consider the critical inequality (2.29) discussed in the last section. The linear term on the right-side is (up to the additive term Φ^t) the linearization of the potential function at x^t, since as was described above, $\alpha\pi^T M(y^t - x^t)$ is the gradient of Φ at x^t. In the block angular case we may write:

$$\pi^T M(y^t - x^t) \quad = \quad \sum_{i=1}^{K} \pi^T M^{(i)}(y^{(i)t} - x^{(i)t}), \tag{2.34}$$

where for $1 \leq i \leq K$, $y^{(i)t}$ is the subvector of y^t corresponding to P^i, and similarly for $x^{(i)t}$ and $M^{(i)}$. Note that $\alpha\pi^T M^{(i)}$ is the gradient of Φ restricted to $R^{n(i)}$, where $\pi = \pi(x^t)$. Thus, term i can be viewed as an estimation on the reduction of potential due to the step in P^i. This suggests the so-called round-robin procedure, which replaces a single Frank-Wolfe iteration as described in the previous section.

Step 0. Compute $\pi^T M$.

Step 1. For $i = 1, 2, \cdots, K$:

1 Let $y^{(i)t}$ be the optimal solution to $\min_{y \in P^i(v)} [\pi^T M^{(i)}]^T y$.

2 Update $x^{(i)t} \leftarrow (1 - \tau)x^{(i)t} + \tau y^{(i)t}$.

3 Update Mx^t through the change in $M^{(i)}x^{(i)}$.

4 Update gradient: recompute π, and $\pi^T M^{(i+1)}$ if $i < K$.

Suppose we were to analyze the improvement on the potential function due to one iteration of Steps 0-1 of the round-robin procedure (all K blocks), and of algorithm **GK**, by incrementally accounting for the improvement resulting from each block step.

Then the second order terms in both cases would be the same as order of magnitude as that in (2.29), while (apparently) the first order term in the round-robin procedure would be better. Thus, intuitively, K consecutive block-steps in the round-robin procedure should yield at least the same improvement in the potential function as one iteration of **GK**. This argument is only superficially correct: it is only valid if $K = 2$. Radzik [R95] provided a a valid analysis of the round-robin procedure, with step 1 above modified so that a step in block i is taken provided the decrease in potential is large enough. Radzik's method is also closely related to a randomized algorithm that we described below.

2.1 Basic deterministic approach

The essential ideas on how to take advantage of block-angular structure can be found in [GK94], [PST91]. One important component is the substitution of the set P with a different set over which we solve linear programs, to better handle width. We already saw a basic form of this approach in the previous sections; there we replaced P with sets of the form $P(v^A)$ (i.e., P augmented with upper bounds v_j on the variables, see (2.7)). The implication was that if $x \in P(v^A)$, every entry of Ax is at most the number of nonzeros in the corresponding row of A.

When we have a block-angular problem, [GK94] and [PST91] elaborate on this idea as follows. For $1 \leq i \leq K$, let A^i be the submatrix of A corresponding to block i. Then we replace P^i with

$$P^i(A) \;=\; P \cap \left\{x \,:\, A^i x \leq 1\right\} \tag{2.35}$$

which is valid so long as we know that the set $P \cap \{x : Ax \leq 1\}$ is nonempty. This certainly holds in the cases considered in section 1.3. As a special case, if each row of each A^i has at most one nonzero (e.g. in a multicommodity flow problem) then we essentially recover the set $P(v)$. As a consequence, if $x \in P$ is such that $x^i \in P^i(A)$, then $Ax \leq K$, i.e. guaranteeing width at most K.

Of course, we could have used this construct in conjunction with the algorithm in Section 1 when handling a block-angular problem. The algorithm we discussed there updates *all* variables in each iteration; i.e. each iteration we counted in the analysis of that algorithm implies K block-iterations; i.e. the actual count of block iterations would be $O(\frac{QK \log m}{\epsilon^2})$. Using the $P^i(A)$ improves the dependence on Q in the above proofs to factors of K; and in particular, the stepsize can be increased to

$$\tau \;=\; O\left(\frac{\sigma}{K\alpha}\right) = O\left(\frac{\sigma^2}{K \log m}\right) \tag{2.36}$$

A variation of the above proof shows that this strategy improves the block-iteration count to $O(\frac{K^2 \log m}{\epsilon^2})$. To fix ideas, in what follows we will refer to this approach as the ALL-BLOCKS strategy.

Given this improvement, the reader may now wonder why we could not simply view the general problem **PACK(A,P)** as a one-block problem, i.e. set $K = 1$. A moment's reflection shows that when $P^1(A) = \{x : Ax \leq 1, x \in P\}$, i.e. we recover the problem we are trying to solve, and this "reduction" can hardly be viewed as a simplification.

More to the point, in a general case the replacement of a P^i by $P^i(A)$ potentially adds a substantial number of constraints to P^i and quite

possibly a Frank-Wolfe step will become much more difficult, in addition to having to handle a much larger linear program – although the overall nonzero count among all the $P^i(A)$ is the same as the total nonzero count in the original formulation of **PACK(A,P)**.

Consider the special case where each row of A^i has (at most) one nonzero. In the multicommodity case, we replace shortest path computations with theoretically more difficult minimum-cost flow problems. However, an effective implementation of these ideas will carry out the Frank-Wolfe iterations from a warm start, and in our experimentation the improved convergence rate more than offsets the slightly worsened running time per iteration. In fact, this observation holds true even for much more general problems than multicommodity flow problems, even when the matrices A^i contain many nonzeros per row, provided that K is large.

In [GK94] the authors propose a procedure that, unlike ALL-BLOCKS, leverages block-angular structure. To simplify notation, in the description we assume that the procedure has already been run some number of times, and that $x = (x^1, x^2, \cdots, x^K)^T$ is the current iterate, where $x^i \in P^i(A)$, $1 \leq i \leq K$. Also, here we are now using

$$\tau = O\left(\frac{\sigma}{\alpha}\right) = O\left(\frac{\sigma^2}{\log m}\right) \tag{2.37}$$

in contrast with (2.36).

The procedure is as follows:

Procedure BEST.

1 For each $i = 1, 2, \cdots, K$, individually, compute $y^i \in A^i(P)$, the solution to a Frank-Wolfe step **restricted** to block i, and Δ^i, the decrease in potential function that would result from updating $x^i \leftarrow (1-\tau)x^i + y^i$ and leaving all x^j, $j \neq i$, unchanged.

2 Pick h such that $\Delta^h = \max_i \Delta^i$.

3 Update x by resetting $x^h \leftarrow (1-\tau)x^h + y^h$.

Notice that to a certain extent, this procedure appears to waste some computational effort (a factor of $K-1$ if all blocks are identical). Hence one may wonder what benefit, it any, this procedure entails over the ALL-BLOCKS strategy. Part of the answer lies in that since we only use one block at a time, the effective width in BEST.1 is 1, rather than K. This enables us to use the stepsize τ in (2.37), larger by a factor of K than that in (2.36). A simple manipulation of the proof in Section 1.3 shows that this increase in stepsize yields a $\sum_i \Delta^i$ which is a factor of

K larger than the decrease in potential achieved by the ALL-BLOCKS strategy in one iteration. Of course, we cannot realize a decrease of potential equal to $\sum_i \Delta^i$ - this would entail using all blocks, yielding effective width K, not 1. But the block h that we pick is guaranteed to have Δ^h at least equal to $\frac{\sum_i \Delta^i}{K}$, and we recover the performance of the ALL-BLOCKS strategy, but with the advantage of a stepsize that is a factor of K longer, which is extremely significant from a numerical standpoint.

Also note that the potential function improvement resulting from using the best block can be used to check termination: as soon as this improvement is less than $O(\frac{\sigma^2}{K})$ we can stop – i.e. the termination condition in Step 3' of algorithm **GK** modified to account for the improved treatment of width: stop when the potential fails to decrease by at least $O(\frac{\sigma^2}{K})$. Using this rule, a modification of the proof in Section 1.3 shows that we will obtain an ϵ-approximate solution to *PACK(A, P)* in at most $O(\frac{K \log m}{\epsilon^2})$ iterations of BEST, for a total of $O(\frac{K^2 \log m}{\epsilon^2})$ Frank-Wolfe steps.

2.2 Randomized approaches

The analysis we just presented essentially argues that one Frank-Wolfe iteration using a single *randomly* chosen block should yield a potential decrease that is good enough to guarantee convergence. More precisely, the expected value of the potential decrease is good enough to guarantee convergence. [As a technical aside, it is possible that a random block iteration may fail to improve potential; and so we should only use the step provided we actually get improvement]. In principle, this would yield an algorithm that is a factor of K faster than one based on procedure BEST or on the ALL-BLOCKS strategy. However, such an algorithm might appear to converge in expectation only; there is a simple trick that produces a provably good algorithm with the same asymptotic running time. In addition, we need some means to verify the termination condition.

This trick (or similar versions of it) consists of occasionally running one iteration of the ALL-BLOCKS strategy. This idea was applied to multicommodity flow problems in [KPST90]. In [PST91], we follow up (with probability $1/K$) each iteration using a randomly chosen block with one iteration of the ALL-BLOCKS strategy. (the procedure in [PST91] is slightly different from the way we described it here; but it reduces to our version when using the $P^i(M)$ sets, as we are). In [GK94], the approach is to repeatedly choose one of the following $K+1$ alternatives with equal likelihood: run one iteration using one block only, or

run one iteration of BEST. In either case the expected number of blocks used per iteration is $O(1)$.

In both cases, the resulting stochastic process and its analysis are reminiscent of the classical gambler's ruin problem (see, for example, [F]), with potential acting as "wealth". In either case, one can essentially prove a $O(\frac{K \log m}{\epsilon^2})$ bound on the expected number of Frank-Wolfe steps needed until termination.

2.3 What is best

Which is the best approach to use with block-angular problems? Here we give a quick view on this subject based on our experimentation with large problems, mostly minimum-cost multicommodity flow problems and network design problems on graphs with several hundred to several thousand nodes, thousands of arcs, and thousands of commodities (yielding linear programs with millions of variables).

First, it appears that the ALL-BLOCKS strategy is the weakest, primarily because of its small step size, which leads to very small potential improvement per iteration. This is the case even though in our implementation we conduct a line search to determine the best step size.

In our testing, both the round-robin and randomized strategies performed well, with round-robin giving slightly better results. On the other hand, most problem instances have non-uniform blocks, and occasionally the BEST strategy was useful. In our implementation (described later) we use a hybrid strategy, which seems to work best: on odd iterations we run round-robin, and on each even iteration we run one step using the current best block (best in terms of potential decrease, and as observed and updated by the round-robin iterations). Further, as we will see, with decreasing frequency we run a step that is somewhat stronger than an ALL-BLOCKS iteration in terms of proving termination.

3. Optimization problems and more general feasibility systems

The methodologies described so far apply to problems **PACK(A,P)** only. In this section we outline how the same techniques can be applied to more general feasibility problems, and through them, to optimization problems.

As shown in [PST91], the paradigm of exponentially penalizing infeasibilities can be directly carried over to more general feasibility systems, yielding provably good algorithms. A covering problem is that of finding an approximately feasible solution to a system $S = \{Ax \geq e,\ x \in P\}$ or to prove that S is empty, where $A \geq 0$ has m rows. Given $x \in P$,

[PST91] assigns to it the penalty $\sum_i e^{-\alpha a_i^T x}$, and shows that an algorithm substantially similar to that described above finds an ϵ-optimal solution to the problem

$$\mathbf{COV(A,P):} \qquad \begin{array}{rcl} \mu^*_{A,P} = \max & & \mu \\ \text{s.t.} & & \\ Ax & \geq & \mu e \\ x & \in & P, \end{array}$$

in $O(\frac{n \log m}{\epsilon^2})$ Frank-Wolfe steps over P, for $\alpha = O(\frac{\log m}{\epsilon})$.

In fact, as shown in [PST91], the requirement that $A \geq 0$ in can be (substantially) relaxed to the assumption that $Ax \geq 0$ for all $x \in P$, thus encompassing a much wider class of problems, while retaining the same complexity bound.

Finally, even the assumption $Ax \geq 0$ can be avoided, through the simple expedient of adding a new variable, s, and replacing (in the packing case) the system $Ax \leq e$ with $Ax + \beta s \leq (1+\beta)e$ and $x \in P$ with $x \in P$, $s = 1$, where $\beta > 0$ is chosen so that $Ax + \beta e \geq 0$ for all $x \in P$. Using such a choice, if $x \in P$ is such that $Ax + \beta e \leq (1 + \frac{\epsilon}{1+\beta})(1+\beta)$ then $Ax \leq (1+\epsilon)e$, and, in principle, we have reduced **PACK(A,P)** to a similar problem with an additional variable, but at the possibly nontrivial cost of sharpening the required tolerance to $\frac{\epsilon}{1+\beta}$. This trick can be extended to handle systems $\{Ax \leq b, x \in P\}$ for general A and b (see [PST91]).

The techniques we have covered so far in this section are a necessary prelude for what we axe really interested in, which is to solve optimization problems. Suppose we have a linear program of the form

$$\mathbf{(LP)} \qquad c^* = \min\left\{c^T x : Ax \leq b, x \in P\right\},$$

where we may (now) assume $b > 0$, and that the feasible region is nonempty. We may also assume $c^* > 0$, using a trick we discussed in the previous paragraph.

To find an ϵ-approximate solution to this linear program, we consider feasibility systems of the form

$$c^t x \leq B \tag{2.38}$$
$$Ax \leq b \tag{2.39}$$
$$x \in P \tag{2.40}$$

and perform binary search over the "budget" parameter $B > 0$. For a given choice for B we can apply the potential reduction techniques we have discussed in this Chapter; thus we will either find an ϵ-feasible

solution to (2.38)-(2.40), or conclude that this system is infeasible. In the former case B becomes a new (ϵ-approximate) upper bound on the value of the linear program, while in the latter, B becomes a new (strict) lower bound. Thus the outcome of the binary search will be $\hat{x} \in P$ with $A\hat{x} \leq (1+\epsilon)b$ and $c^T\hat{x} \leq (1+\epsilon)c^*$.

Of course, given that we are solving a linear program, we would also like a duality proof of the near-optimality of $\hat{x}$. Equivalently, we would like an LP duality interpretation of the infeasibility of a system of the form (2.38)-(2.40). The algorithmic techniques for packing problems that we covered starting in Section 1 are less than satisfactory in this context – recall the that the termination condition in Section 1.3 is potential-based. On the other hand, the algorithm in [PST91] is more transparently based on linear programming duality, and we will now outline how their algorithm certifies that a system (2.38)-(2.40) is infeasible.

One way to show this is to prove that λ^*, the minimum value of $\lambda \geq 0$ such that the system

$$c^T x \leq \lambda B \tag{2.41}$$

$$Ax \leq \lambda b \tag{2.42}$$

$$x \in P, \tag{2.43}$$

is feasible, satisfies $\lambda^* > 1$. In [PST91] the following approach is used: let $\pi_0 > 0$ be a scalar, and $\pi > 0$ be a vector of the same dimension as b. Define

$$\bar{\lambda} = \frac{\min_{x \in P} \left\{ (\pi_0 c^T + \pi^T A)x \right\}}{\pi_0 B + \pi^T b}. \tag{2.44}$$

Then a simple analysis shows that $\lambda^* \geq \bar{\lambda}$. Thus, if $\bar{\lambda} > 1$ we have that (2.38)-(2.40) is infeasible, and, in fact, that $\bar{\lambda}B > B$ is a lower bound on **LP.** The algorithm in [PST91], when terminating with an infeasibility proof, produces precisely such a vector $\pi_0, \pi_1, \cdots, \pi_m$, where each π_i is an exponential penalty.

To obtain a better understanding of the relationship between this result and the techniques used in [GK94], it is useful to first scale constraint (2.38) by B^{-1} and row i of (2.39) $(1 \leq i \leq m)$ by b_i^{-1}, thereby casting (2.41)-(2.43) into the form **PACK(M,P).** In this case, (2.44) can be viewed as a weak duality (or Farkas Lemma) proof that $\lambda^* \geq \bar{\lambda}$ using dual variables $\frac{\pi_i}{\sum_j \pi_j}$ (compare this expression, for example, to (2.13)).

Later we will show that when $\bar{\lambda} > 1$, the bound $\bar{\lambda}B$ is *not* best possible; that is to say, we can manufacture from the π a different vector of duals that proves a *strictly* better lower bound than $\bar{\lambda}B$. This issue,

and others related to the efficient implementation of the exponential potential function method to approximately solve linear programs, are discussed in the computational section of this monograph.

4. Width, revisited

The width parameter that we have discussed in previous sections plays a fundamental role in the convergence rate of potential function methods, both from theoretical and numerical standpoints. If not properly handled, it can lead to implementations with very poor convergence properties.

The starting point for these difficulties is the first-order nature of the algorithms we have been discussing. The reader is probably familiar with the "zig-zagging" behavior of first-order algorithms for nonlinear programming problems; that is not the problem we allude to. Instead, first-order procedures can produce sequences of very small steps. In a worst-case scenario, the step-lengths become numerically indistinguishable from zero, while the iterates are still far from optimality. Thus, effectively, the algorithm converges to an incorrect point. This is the so-called "jamming" or "stalling" effect. Now recall than in the above procedures we are attempting to (approximately) minimize potential in lieu of achieving feasibility for a linear system. In case of stalling, an algorithm will converge to a point that is potentially quite infeasible for the system we are interested in.

In terms of the exponential potential function, it is useful to look at the critical section of the proof we presented. This is the set of equations beginning with (2.22), continuing through with (2.25), (2.26) and (2.26), and culminating with (2.29).

We are running a first-order method: thus at each iteration we approximate the potential function with its current gradient. In order for this approach to succeed, the potential function should approximately behave like its gradient in a neighborhood that is "large": this will permit the step-length to be large. Equation (2.25) is the key. Here the potential decrease due to a constraint i is of the form $\pi_i(e^{\alpha\tau m_i^T(y^t-x^t)}-1)$, where τ is the step-length, x^t is the current iterate, and y^t is the solution to the step finding linear program, i.e. $y^t - x^t$ is the step-direction. Critically, $\pi_i m_i^T y^t$ is the contribution, from row i, to the inner product between y^t and the gradient of Φ at x^t.

Thus, we would like $\pi_i(e^{\alpha\tau m_i^T(y^t-x^t)} - 1)$ to behave like $\tau\alpha\pi_i m_i^T(y^t - x^t)$: we know that y^t optimizes the current linearization and thus will make the second expression as small as possible. But we know that for $z \geq 0$, $e^z - 1$ is approximately z only for z small. Consequently, all

other factors being fixed, a choice of y^t such that $m_i^T y^t$ is large makes it necessary to choose τ *small.* Let us repeat this statement in slightly modified form: an increase in $m_i^T y^t$ necessitates a proportional decrease in τ. The largest possible value of $m_i^T y$, over all i and $y \in P$, is precisely the width. Thus, in the worst case step-length is inversely proportional to width. Conversely, an algorithm that maintains low width will achieve correspondingly larger step-lengths.

From an experimental standpoint, controlling width is an important and easily observable requirement. From a theoretical standpoint, the more recent algorithms that use the exponential potential function framework (see next chapter) can be viewed as exercising careful control over width.

Finally, we note that to some degree width-related problems are a fixture of Lagrangian relaxation schemes. The fact that we use an explicit (and carefully chosen) potential function makes width control a more deliberate and more carefully calibrated pursuit.

5. Alternative potential functions

The proofs we presented above heavily relied on using an exponential potential function. But it should be clear that (with some effort) other "rapidly increasing" potential functions should suffice to at least prove polynomial-time convergence (after all, we achieved this end with the rational barrier function in Chapter 1). For example, [GK96] analyzes a Karmarkar-like logarithmic barrier function, and shows that under certain conditions better bounds for this function than for the exponential function.

On the other hand, the quadratic penalty function (popular in the context of penalty methods, see next chapter for a brief discussion), does not seem to easily lead to provably low complexity approximation algorithms.

6. A philosophical point: why these algorithms are useful

As we have seen, the algorithms based on the exponential penalty function (and the flow deviation method) achieve a a desired accuracy ϵ while requiring, **in the worst-case,** a number of iterations that grows proportional to ϵ^{-2} (and polynomial in the overall size of the problem). This is in marked contrast to interior point methods for linear programming, whose dependence on ϵ grows as $\ln(1/\epsilon)$.

Thus, the exponential penalty methods are not even polynomial-time algorithms, and, asymptotically, they are clearly inferior to standard

methods. In addition, a nonlinear programmer might even pinpoint the critical failing of these algorithms, which leads to their asymptotic behavior: they are first-order methods. Why are we pursuing this line of research?

One justification, employed in e.g. [PST91], is that for *particular* classes of linear programs, and small, but constant ϵ (say $\epsilon = 10^{-3}$) the exponential penalty algorithms have a better provable asymptotic behavior. Another justification is that "the proof is in the pudding": we are, after all, comparing worst-case asymptotic estimates, and in the end what really matters is the actual experimental behavior of the competing algorithms (see Chapter 4 for some rather stunning successes of the exponential penalty methodology).

But there is a much more immediate advantage of the algorithms we have described: they are *thrifty*. They effectively leverage the block-angular structure of problems, whenever it exists, and in very large-scale instances this capability constitutes the essential feature that separates "usable" algorithms from the rest. Simply put, standard linear programming methods will use a forbiddingly prodigious amount of storage when applied on a very large problem instance – the ability to use one block at a time is central to an algorithm's ability to be at all runnable.

Clearly, this sort of motivation has always lurked behind the development of classical decomposition schemes (such as the Dantzig-Wolfe procedure, Benders' decomposition, and even column generation approaches) and of Lagrangian relaxation schemes. What is different about the exponential penalty methodology is that we are able to prove *something* about it, thus giving the field a measure of engineering precision. We would argue that as problem sizes dramatically grow, this type of precision will gain acceptance.

In summary, *even though* the exponential penalty methods are decomposition schemes that only look at a tiny part of a problem at a time, and even though they are first-order methods, they *are not too bad.* That is to say, the $O(\epsilon^{-2})$ bound is not a measure of how good the algorithms are: it is simply a statement that they are not wildly inefficient.

Chapter 3

RECENT DEVELOPMENTS

In this chapter we survey several exciting research developments that have taken place in the last few years regarding approximation of linear programs. First, Neal Young's work on derandomization (perhaps a more apropos term would be deconstruction) algorithms for covering and packing problems that provides a "natural" justification for the use of an exponential potential function. This work in turn gave rise to streamlined, and theoretically faster, ϵ-approximation algorithms for covering or packing problems (see [GK98], [F00]) which avoid an explicit use of an exponential potential function although at a fundamental level still use it. This work also has resulted on lower bounds on the complexity of Frank-Wolfe type algorithms [KY98].

Another field that has been reinvigorated is that of subgradient-based algorithms. On the computational side, Barahona and Anbil's work on the "volume" algorithm [BA00] has provided new momentum to classical Lagrangian relaxation methods for linear programming.

1. Oblivious rounding

Here we will describe the results in [Y95] on derandomizing algorithms. This work has had significant impact on recent developments on approximation algorithms for classes of linear programs.

To motivate the discussion, consider the classical (integral) set-covering problem: we are given an $0-1$ matrix A with m rows, and we want to solve the integer program

$$\begin{array}{lll} & \min & \sum_j x_j \\ \textbf{(SC)} & & \text{s.t.} \\ & Ax \geq e & (3.1) \\ & x_j \in \{0,1\} \quad \forall j, & (3.2) \end{array}$$

where e is the vector of m 1s. This problem is NP-hard; instead we next describe an approximation algorithm based on a very clever approach described in [Y95].

The central idea is to imagine that we have access to a very powerful computer that has, in fact, already solved **SC**. Alas, in order for us to actually obtain the optimal cover, we must query an oracle to get one element of the cover at a time – and, unfortunately, this oracle suffers from randomness: it may repeat its answers, randomly. Each call to the oracle is very expensive, and we would like to reduce the number of calls as much as possible. In fact, we would like to deconstruct the randomness of the oracle so as to obtain the optimal cover ourselves, without resorting to the oracle at all (or the powerful computer). As we will see next, we will fail at this task, but instead we will get a deterministic algorithm that computes a provably near-optimal cover.

Now the details. Let x^* be an optimal solution to this problem, and let $S^* = \left\{j : x_j^* = 1\right\}$. Suppose that there is an oracle such that each call to the oracle returns a **random** member of S^*, chosen from the uniform distribution. Clearly, after *enough* calls to the oracle we will be able to construct S^*; but can we provide some nontrivial estimate?

Consider an index i, $1 \leq i \leq m$. We know that there is at least one index $j \in S^*$ such that $a_{ij} = 1$. Thus, the probability that a call to the oracle will return a column that covers i is at least $1/|S^*|$. In other words, the probability that i is not covered is at most $1 - 1/|S^*|$. After T calls, the probability that i is still not covered is at most

$$(1 - 1/|S^*|)^T \tag{3.3}$$

Given that there are m rows to cover,

$$P(\text{not all rows covered after } T \text{ calls}) \leq m(1 - 1/|S^*|)^T < 1 \tag{3.4}$$

for $T = \lceil |S^*| \log m \rceil$. Thus, using this choice for T, after at most T oracle calls with positive probability we will have obtained S^* (which, incidentally, was not required to be an optimal cover in this analysis).

As planned above, we would like to decrease our reliance in the oracle. As a starting point, we would like to have a deterministic algorithm which outputs a column of $\hat{j}$ of A, with the following property:

- With positive probability, $T-1$ calls to the oracle will produce a set of columns that cover all those rows of A not covered by $\hat{j}$.

In other words, we obtain a hybrid algorithm that is a little less dependent on the oracle than the initial, all-oracle procedure.

To see how this is possible, we can rewrite the middle term in (3.4) as

$$\sum_{rows\ i} (1 - 1/|S^*|)^T. \tag{3.5}$$

For any given row i, the term in (3.5) corresponding to i is *at least*

$$\frac{(\text{no. of columns of } S^* \text{ not covering } i)}{|S^*|}(1 - 1/|S^*|)^{T-1} \quad , \tag{3.6}$$

and thus the sum in (3.5) is lower-bounded by

$$(1 - 1/|S^*|)^{T-1} \sum_{rows\ i} \frac{(\text{no. of columns of } S^* \text{ not covering } i)}{|S^*|} \quad = \tag{3.7}$$

$$(1 - 1/|S^*|)^{T-1} \sum_{j \in S^*} \frac{(\text{no. of rows not covered by } j)}{|S^*|}. \tag{3.8}$$

Now let us pretend for a moment that the expression in (3.8) equals the probability of failure for the all-oracle procedure, rather than an estimate for it. In this sense, we can interpret (3.8) as stating that the probability of failure, given the first choice $j \in S^*$ made by the oracle, is proportional to the number of rows not covered by j (since the oracle chooses columns of S^* with probability $1/|S^*|$). Thus if we could influence the oracle so as to favor those columns $j \in S^*$ that cover more rows we would increase our probability of success. [In fact, (3.8) can directly be interpreted as an upper bound on the expectation on the number of uncovered rows after T calls, with each potential choice for the first column returned by the oracle making an appropriate contribution.]

But we cannot, of course, control the oracle. Instead, let us try the next best approach: we will greedily pick, from among *all* columns, that column j_1 that covers a *maximum* number of rows. Given this deterministic choice, our probability of failure with $T-1$ ensuing oracle calls is at most

$$(\text{no. of rows not covered by } j_1) \times (1 - 1/|S^*|)^{T-1}, \tag{3.9}$$

which, by choice of j_1, is at most the last expression in (3.8), and thus is less than 1. Consequently, we have reached our goal: we now have an algorithm that, after carrying out a deterministic task, only needs

$T - 1$ oracle calls to achieve positive probability of success. [Actually, we have proved something stronger: the probability of failure is no more than the upper bound 3.4 on the failure probability for the all-oracle procedure.] In other words, if we remove from A those rows covered by j_1, we have that the $T - 1$ calls to the oracle will cover all rows of the resulting matrix with positive probability.

Note that we never used the fact that S^* is a minimum cover. Hence, we can proceed inductively: in the next step, we choose a column j_2 so as to cover a maximum number of rows of the current matrix, or, in other words, we choose j_2 so that, from among all columns, it covers the maximum number of rows not already covered by j_1. By induction, $T - 2$ oracle calls will cover all rows of A not covered by $j_1 \cup j_2$, with positive probability.

Continuing inductively, at step $t \geq 2$ of our hybrid deterministic/oracle procedure we choose a column j_t so as to cover a maximum number of rows not already covered by $j_1 \cup \cdots \cup j_{t-1}$. By the above analysis, the probability that with $T - t$ oracle calls we will cover the rows not covered by $j_1 \cup \cdots \cup j_t$ is positive.

Suppose $t = T - 1$. Then the probability that *one* oracle call will return a column that covers all remaining rows is positive. In other words, there exists a column that covers all remaining rows, and therefore one more application of our deterministic algorithm will find such a column. Success! We have entirely removed the oracle from our procedure, and in the process we have obtained a new proof of a classical result:

LEMMA 3.1 *([L75], [C79], [W82]) The greedy procedure finds a cover of size at most* $\lceil |S^*| \log m \rceil$. ■

1.1 Concurrent flows

The previous analysis was provided to motivate the idea of oblivious rounding. The name stems from the fact that the same methodology can be used to round fractional solutions to integral solutions. This can be done, for example, in the case of the set covering problem, with equal performance bound as above. Instead we will focus here on a more apropos example, the maximum concurrent flow problem introduced in Chapter 1.

We will seek to apply the same methodology – to create an appropriate oracle/random procedure that requires "few" calls to yield the *optimal* solution, yet is easy enough to deconstruct, obtaining a deterministic, provably good *approximation* algorithm.

In this setting it is most convenient to look at the problem in the minimum congestion format. We will first consider the *uniform* case,

and later return to the general case. To refresh our memory, we are given a network where each edge e has unit capacity, and a set of commodities to route, where commodity k $(1 \leq k \leq K)$ has origin s^k, destination t^k and demand amount d^k. The objective is to (fractionally) route all of the demands so as to minimize the maximum congestion. We will assume that the demands have been scaled so that $\sum_k d^k = 1$. As noted in Chapter 1 this only changes the value of the problem by a constant factor.

Let f denote an optimal flow, with maximum congestion $\lambda^* = \lambda(f)$. Suppose that for each commodity k, the flow between s^k and t^k can be decomposed into a set of paths $S(k)$ between s^k and t^k. We will denote the flow routed on a given path p by $f(p)$. Thus $\sum_{p \in S(k)} f(p) = d^k$.

1.1.1 The oracle

We can now describe the oracle: one call to the oracle returns, for each commodity k, a *random* path $p \in S^k$, chosen with probability $\frac{f(p)}{d^k}$.

In order to build a solution to the maximum concurrent flow problem, we call the oracle a large number T of times. Each time a path corresponding to a commodity k *is* returned, we route an additional d^k units of flow along this path. At the end of the T calls, we scale the overall flow by a factor of T^{-1}. Denote by x the resulting flow. We claim that given $0 < \epsilon < 1/2$, $\lambda(x) \leq (1+\epsilon)\lambda^*$ for large enough T, with positive probability.

To prove this fact, consider the following random variables:

- p_i^k, the path for commodity k that is returned by the oracle in call i, $1 \leq i \leq T$,
- $x_{e,i}^k$, the amount of flow of commodity k assigned to edge e in call i,
- $x_{e,i}$, the total amount of flow assigned to edge e in call i.

We have:

$$x_e = \frac{1}{T}\sum_i^T x_{e,i} \tag{3.10}$$

$$= \frac{1}{T}\sum_i^T (\sum_k x_{e,i}^k). \tag{3.11}$$

By construction, $x_{e,i}^k$ is set to d^k if $e \in p_i^k$ and zero otherwise. Thus, the expectation of each $x_{e,i}^k$ is f_e^k, and the expectation of $x_{e,i}$ is f_e. In other words, x_e is an average of T i.i.d. random variables, each with mean

$f_e = \sum_k f_e^k$, and hence the expectation of x_e itself is f_e. Now note that since we are dealing with unit capacities, $\max_e f_e = \lambda^*$, by definition.

Thus, the event that

$$\lambda(x) > (1+\epsilon)\lambda^*, \tag{3.12}$$

implies that for some edge e,

$$x_e > (1+\epsilon)\lambda^* \tag{3.13}$$

$$\geq (1+\epsilon)f_e. \tag{3.14}$$

In other words, for some edge e, x_e is larger than its own expectation by a constant factor.

In the probability literature, the study of such events falls under the heading of "large deviations". Generally speaking, the tightest bounds on probabilities of large deviations are obtained through the Central Limit Theorem [F], but such bounds can be algebraically cumbersome. A more flexible approach involves the use of "transforms" or "moment generating functions". For a (number valued) random variable X, this involves studying the behavior of $E[t^X]$ as a function of the parameter t. The study of this function has led to useful bounds (such as the Chernoff bound) on large deviation probabilities, asymptotically nearly as tight as those derived from the Central Limit Theorem.

To show that with high probability $\lambda(x) \leq (1+\epsilon)\lambda^*$. we will apply this technique using $t = 1+\epsilon$.

To this effect, we can rewrite (3.13) as

$$\sum_i x_{e,i} > T(1+\epsilon)\lambda^*, \tag{3.15}$$

which is equivalent to

$$\prod_{i=1}^{T} \frac{(1+\epsilon)^{x_{e,i}}}{(1+\epsilon)^{(1+\epsilon)\lambda^*}} > 1. \tag{3.16}$$

Since $(1+\epsilon)^{x_{e,i}} \leq 1+\epsilon x_{e,i}$ (because by assumption the sum of demands is 1, implying $x_{e,i} \leq 1$) the probability of (3.13) holding is *at most*

$$E[\prod_{i=1}^{T} \frac{1+\epsilon x_{e,i}}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}], \tag{3.17}$$

which, since for fixed e the $x_{e,i}$ are independent, equals

$$\prod_{i=1}^{T} \frac{1+\epsilon E[x_{e,i}]}{(1+\epsilon)^{(1+\epsilon)\lambda^*}} = \tag{3.18}$$

$$\prod_{i=1}^{T} \frac{1+\epsilon f_e}{(1+\epsilon)^{(1+\epsilon)\lambda^*}} = \left(\frac{1+\epsilon f_e}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}\right)^T \tag{3.19}$$

The logarithm of the multiplicand in (3.19) is at most

$$\epsilon f_e - (\epsilon - \frac{\epsilon^2}{2})(1+\epsilon)\lambda^*, \tag{3.20}$$

which, since $f_e \leq \lambda^*$, is at most

$$-(\frac{\epsilon^2}{2} - \frac{\epsilon^3}{2})\lambda^* \leq -\frac{\epsilon^2}{4}\lambda^*. \tag{3.21}$$

In summary, we now have:

LEMMA 3.2 *The probability that for a given edge e,* $x_e > (1+\epsilon)\lambda^*$ *is at most* $e^{-T\frac{\epsilon^2}{4}\lambda^*}$. ■

Finally, since there are m edges and the demands sum to 1, $\lambda^* \geq \frac{1}{m}$, and

LEMMA 3.3 *For* $T = \lceil \frac{4 \log m}{\lambda^* \epsilon^2} \rceil \leq \lceil \frac{4m \log m}{\epsilon^2} \rceil$ *the probability that* $\lambda(x) \leq (1+\epsilon)\lambda^*$ *is positive.* ■

1.1.2 The deterministic algorithm

We will next turn the T oracle calls into a deterministic algorithm which after T iterations has a flow guaranteed to have congestion at most $(1+\epsilon)\lambda^*$. We will proceed much in the same form as in the case of the set-covering problem.

The starting point is inequality (3.15), which describes the event that the final flow has excessively large congestion. We ask the question: in which way could we influence the outcome of the **first** oracle call, so that by the end of call T the probability of this event is as small as possible?

To proceed formally, a multicommodity flow vector where the flow of each commodity is carried by a single path will be called a *routing*. As introduced above, $S(k)$ is the set of paths for commodity k obtained in the flow decomposition of the *optimal* flow. Let S be the set of all routings obtained by specifying, for each k, one path in $S(k)$, and routing all the demand for commodity k on this path. For $r \in S$ let $x_e(r)$ denote the value of the flow on edge e assigned under routing r. We can rewrite (3.18) as:

$$\left(\sum_{r \in S}[1 + \epsilon x_e(r)P(r)]\right) \prod_{i=1}^{T-1} \frac{1+\epsilon E[x_{e,i}]}{(1+\epsilon)^{(1+\epsilon)\lambda^*}} \tag{3.22}$$

where $P(r)$ is the probability that the oracle chooses routing r. This expression equals

$$\left(\prod_{i=1}^{T-1} \frac{1+\epsilon E[x_{e,i}]}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}\right)\left(\sum_{r\in S} P(r)[1+\epsilon x_e(r)]\right) = \quad (3.23)$$

$$\frac{(1+\epsilon f_e)^{T-1}}{(1+\epsilon)^{T(1+\epsilon)\lambda^*}}\left(\sum_{r\in S} P(r)[1+\epsilon x_e(r)]\right) \leq \quad (3.24)$$

$$\frac{e^{-(T-1)\frac{\epsilon^2}{4}\lambda^*}}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}\left(\sum_{r\in S} P(r)[1+\epsilon x_e(r)]\right) \quad (3.25)$$

Thus, the probability that any edge has congestion at least $(1+\epsilon)\lambda^*$ is at most

$$e^{-(T-1)\frac{\epsilon^2}{4}\lambda^*}\left(\frac{1}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}\sum_{r\in S} P(r)\sum_e [1+\epsilon x_e(r)]\right) < 1, \quad (3.26)$$

where the inequality follows from the fact the double summation in (3.26) equals $1+\epsilon f_e$, and thus, as argued regarding (3.21), the expression within parentheses in (3.26) is at most $e^{-T\frac{\epsilon^2}{4}\lambda^*}$.

Now we can perform the same analysis as for the set-covering oracle. If we could cause the oracle to favor those routings $r \in S$ for which $\sum_e[1+\epsilon x_e(r)] = m + \epsilon\sum_e x_e(r)$ is small, then according to (3.26) we would increase the chances of success (or rather, the estimator (3.26) of the failure probability would be decreased). Thus, we would prefer the oracle to choose a routing $r \in S$ that minimizes $\sum_e x_e(r)$. Of course, we do not know the set S. Instead, let R denote the set of *all* routings (thus, each element of r specifies a path for each commodity that carries the full demand of that commodity). We will choose $r_1 \in R$ so as to minimize, amount all $r \in R$, $\sum_e x_e(r)$. Following this choice we will perform $T-1$ oracle calls.

An analysis similar to the above now shows that the probability of failure for this mixed algorithm is at most

$$\frac{e^{-(T-1)\frac{\epsilon^2}{4}\lambda^*}}{(1+\epsilon)^{(1+\epsilon)\lambda^*}}\left(\sum_e [1+\epsilon x_e(r_1)]\right) < 1, \quad (3.27)$$

where the inequality follows since by choice of r_1 the left-hand side of (3.27) is at most that of (3.26). Thus, we have successfully derandomized one step of the all-oracle procedure. Note that the computation of the routing r_1 (minimize the sum of flow amounts over all edges) breaks up into a separate shortest path problem for each commodity.

Proceeding inductively, suppose that after j deterministic steps we have chosen routings $r_1, \cdots r_j$ such that if we then were to run $T-j$ oracle steps (and then scale by T^{-1}) the resulting flow has positive probability of success. The same analysis as employed for the first deterministic step shows that if in step $j+1$ we choose a routing $r_{j+1} \in R$ so as to minimize

$$\sum_e \left(\prod_{i=1}^{j} [1 + \epsilon x_e(r_i)] \right) (1 + \epsilon x_e(r_{j+1})), \tag{3.28}$$

the probability of failure (congestion larger than $(1+\epsilon)\lambda^*$) after $T-j-1$ oracle calls is at most

$$\frac{e^{-(T-j-1)\frac{\epsilon^2}{4}\lambda^*}}{(1+\epsilon)^{(1+\epsilon)\lambda^*}} \left(\sum_e \prod_{i=1}^{j+1} [1 + \epsilon x_e(r_i)] \right), \tag{3.29}$$

and that this value is strictly less than 1. Consequently, after proceeding in this manner for T iterations, we will have completely derandomized the algorithm (using no knowledge of the optimal routing or of λ^*). The algorithm we have so far is

Algorithm Y

Step Y.0 For every edge e, set $w_e = 1$. For every commodity k and edge e, set $x_e^k = 0$.

Step Y.1 For $j = 1, 2, \cdots, T$ do:

(a) For each commodity k, let y^k denote the flow resulting by routing d^k units of flow along a shortest path from s^k to t^k using the metric w. For each edge e, reset $x_e^k \leftarrow x_e^k + y_e^k$.

(b) For each edge e, reset $w_e \leftarrow (1 + \epsilon \sum_k y_e^k) w_e$.

Step Y.2 Reset $x \leftarrow T^{-1}x$.

As a consequence of the above analysis, we now know:

LEMMA 3.4 *Algorithm Y finds a flow x with* $\lambda(x) \leq (1+\epsilon)\lambda^*$. ■

1.1.3 Handling general capacities

The previous analysis considered the unit capacity case. We will now show how to adapt the analysis and the algorithm presented in the last section so as to handle the general case. Here each edge e has capacity u_e, and the congestion of a flow x on an edge e is $\frac{\sum_k x_e^k}{u_e}$.

It would seem straightforward to replace, in the above equations, all quantities x_e (and $x_{e,i}$ and f_e) with x_e/u_e so as to obtain congestion amounts. A careful look at the analysis shows that this breaks down when going from equation (3.16) to (3.17). Here we previously needed $x_{e,i} \leq 1$; now we need $\frac{x_{e,i}}{u_e} \leq 1$. The astute reader might then say, this can be achieved without loss of generality by scaling all u_e by a suitably large factor. However, the result of this action will be to proportionally decrease λ^* – with the consequence that the number T of iterations proportionally increases (see Lemma 3.3).

The quantity $\frac{x_e}{u_e}$ can be as large as $\rho \doteq \frac{\sum_k d^k}{\min_e u_e}$; this ratio is the "width" parameter discussed in Chapter 2 (or rather, an upper bound for it). Thus, we can use instead of (3.16) the following to describe the event that the final flow produced by the oracle has congestion larger than $(1+\epsilon)\lambda^*$:

$$\prod_{i=1}^{T} \frac{(1+\epsilon)^{\frac{x_{e,i}}{\rho u_e}}}{(1+\epsilon)^{(1+\epsilon)\frac{\lambda^*}{\rho}}} > 1. \tag{3.30}$$

Proceeding as above, we obtain the following analogue of Lemma 3.2:

LEMMA 3.5 *The probability that for a given edge e,* $\frac{x_e}{u_e} > (1+\epsilon)\lambda^*$ *is at most* $e^{-T\frac{\epsilon^2}{4\rho}\lambda^*}$. ■

Further,

LEMMA 3.6 *For* $T = \lceil \frac{4\rho \log m}{\lambda^* \epsilon^2} \rceil$ *the probability that* $\lambda(x) \leq (1+\epsilon)\lambda^*$ *is positive.* ■

Finally, we can produce a deterministic algorithm as we did above for the unit capacity case. The only difference lies in Step 1(b) where the edge weight update formula becomes

$$w_e \leftarrow \left(1 + \epsilon \frac{\sum_k y_e^k}{u_e}\right) w_e. \tag{3.31}$$

With this change, we have, analogously to Lemma 3.4,

LEMMA 3.7 *Algorithm Y using weight update formula (3.31) finds a flow x with* $\lambda(x) \leq (1+\epsilon)\lambda^*$. ■

The number of iterations of this algorithm, T is proportional to ρ and inversely proportional to λ^*. Suppose we try to remove the dependence on λ^*, for example by scaling up the demands so that $\lambda* \geq 1$. However, this will result in an equal increase in ρ, with no change in running time.

This difficulty can be traced, again, to the way that (3.17) was derived from (3.16).

More precisely, on a given iteration the oracle (and its deterministic counterpart) might decide to route all commodities on a given edge; and as we saw above this introduces the dependence on ρ. We outline next a way of overcoming this difficulty, which borrows ideas from the work [PST91] and [GK94] we discussed in Chapter 2, and also from [GK96], [GK98] and [F00].

(a) Scale the demands by a common multiplier so that for $1 \leq k \leq K$, the maximum flow amount for commodity k and capacities u_e is at most d^k, and so that this bound is attained by at least one commodity. This guarantees $1 \leq \lambda^* \leq K$.

(b) The oracle we previously discussed routes, in each iteration, all flow for each commodity along a single path. These paths are chosen so that the expected value of the routing chosen by the oracle equals the vector f. This, in fact, is the **only** property of the oracle that the analysis above required (see eqs. (3.18)-(3.19)).

Let $\hat{\lambda}$ be an upper bound on λ^*. We now change the oracle, so that for each commodity k, it picks a random *flow* of d^k units, feasible for the capacities $\hat{\lambda}u_e$, chosen from a distribution with mean f^k.

Consequently, the total flow routed by the oracle on any edge e in a given iteration is at most $K\hat{\lambda}u_e$. As a result, the width ρ is replaced by K in (3.30), and T becomes $\lceil \frac{4\hat{\lambda}\log m}{\lambda^*\epsilon^2} \rceil$.

1 In the derandomization of the oracle, we now choose, for each commodity k, a minimum-cost flow, feasible for the capacities $\hat{\lambda}u_e$, that sends d^k units from s^k to t^k. This is different from the previous derandomization in that we do not choose a single path for each commodity. The rest of the deterministic algorithm, especially the edge-weight update formula, remains the same as before.

2 We obtain a flow with congestion at most $(1+\epsilon)\lambda^*$ in at most $K\lceil \frac{4\hat{\lambda}\log m}{\lambda^*\epsilon^2} \rceil$ minimum-cost flow computations.

3 The final step is to couple the algorithm we just described with a "powers of two" search for λ^*, using guesses $\hat{\lambda}$. If a guess $\hat{\lambda}$ for λ^* is valid (i.e. if $\lambda^* \leq \hat{\lambda}$) the above algorithm is correct; otherwise, it may produce a flow with congestion larger than $(1+\epsilon)\hat{\lambda}$. But in such a case we have proved that our guess $\hat{\lambda}$ is in fact a *lower bound* for λ^*. Thus we proceed with guesses $\hat{\lambda} = 1, 2, 4, \cdots$ (without exceeding K)

until we have estimated λ^* up to a factor of two (or perhaps $2(1+\epsilon)$). After that, one more run of the algorithm correctly finds a flow with congestion at most $(1+\epsilon)\lambda^*$. The total number of minimum-cost flow computations is $O(K \log K \lceil \frac{4 \log m}{\epsilon^2} \rceil)$.

1.1.4 Comparison with the exponential potential function method

For the sake of simplicity, let us assume that $\rho = O(K)$ (as was essentially the case in the previous section). At iteration h of the algorithm we just described, the weight assigned to an edge e is

$$\prod_{i=1}^{h} \left(1 + \epsilon \frac{x_{e,i}}{K u_e}\right) \tag{3.32}$$

and the total flow currently routed on e is

$$\sum_{i=1}^{h} \sum_{k} x_{e,i}^k = \sum_{i=1}^{h} x_{e,i}. \tag{3.33}$$

Further, by construction the total flow of commodity k that has been routed by iteration h is hd^k. Therefore, the flow y defined by $y_i^k \doteq h^{-1} \sum_{i=1}^{h} x_i^k$ routes exactly d^k units. For ϵ small, the weight assigned to an edge e now becomes approximately

$$e^{\epsilon \sum_{i=1}^{h} \frac{x_{e,i}}{K u_e}} = e^{\epsilon h \frac{y_{e,i}}{K u_e}} \tag{3.34}$$

Suppose $h \geq \frac{T}{2}$. Writing $\alpha = \frac{2 \log m}{\epsilon}$, the right-hand side of (3.34) is between $e^{\alpha \frac{y_{e,i}}{u_e}}$ and $e^{2\alpha \frac{y_{e,i}}{u_e}}$. In other words, the deterministic algorithm penalizes congestion very much like the exponential potential function method introduced in Chapter 2. It should be possible to prove a much deeper correspondence between the two algorithms.

2. Lower bounds for Frank-Wolfe methods

All the ϵ-approximation algorithms we have discussed so far have complexity that grows as ϵ^{-2} times a polynomial on the size of the problem. We may wonder if this dependence on ϵ^{-2} is the best we can do. A difficulty in this regard is that we do not only want to let $\epsilon \to 0$; we also want the problem size to go to infinity, ideally we should get an algorithm that does well on both accounts. As we will see, only partial progress has been made on proving satisfactory lower-bounds on the complexity for finding ϵ-approximate solutions. At the same time, the available results highlight the role of the width parameter.

An initial lower bound analysis is given in [GK96]. Consider the instance of the minimum congestion problem where the network consists of m parallel edges, all of capacity 1, between vertices s and t, and we wish to send 1 unit of flow from s to t. Thus the problem is

$$\lambda^* = \min_{x \in P} \{\max \{x_1, x_2, \cdots, x_m\}\}, \tag{3.35}$$

where

$$P = \left\{ x \in R_+^m : \sum_j x_j = 1 \right\}. \tag{3.36}$$

Note that $\lambda^* = m^{-1}$. [GK96] considers the question of how many problems over P must be solved, in order to obtain an $O(1)$ approximation to λ^* (in [GK96] the result is presented as concerning ϵ-approximations for "fixed" ϵ; but the result already applies for $O(1)$ approximation).

More precisely, the analysis in [GK96] considers the minimum length of a sequence $x^1, x^2, \cdots, x^T$ where each $x^i \in P$, that *any* algorithm requires in order to achieve $\lambda(x^T) \leq O(\lambda^*)$, where x^{i+1} is obtained from $x^1, x^2, \cdots, x^i$ and the solution of a linear program over the "restricted" set $P^i = \{x \in P \ x_j \leq \lambda(x^i), 1 \leq j \leq m\}$. (We define $P^0 = P$).

Some comments: it makes sense to prove lower bounds on the length of the sequence, since this is essentially the number of Frank-Wolfe of a potential-function like algorithm, i.e. the number of linear programs to be solved. [On the other hand, since we do expect that the λ^i will be monotonically decreasing, these linear programs may gradually become more difficult and the iteration count perhaps loses some relevance]. It also makes sense to modify the set P by progressively tightening bounds on the variables (we saw this in action in our analysis of the oblivious rounding algorithm for the minimum congestion problem, and in Chapter 2, for instance).

Next we provide a sketch of a proof that $\Omega(\log m)$ iterations are needed, under the assumption that each solution of a linear program over a P^i yields an extreme point of P^i.

To prove this, we will show that if $t = o(\log m)$, $|supp(x^t)| = o(m)$, from which it follows that $m^{-1} = o(\lambda(x^t))$ (here *supp* denotes support). This is proved by induction. At the first iteration, all flow is placed on a single edge, and $\lambda(x^1) = 1$. The assumption regarding the P^i shows that $|supp(x^2)| \leq 2$, thus $\lambda(x^2) \geq 1/2$. Thus $|supp(x^3)| \leq 4$ and continuing inductively we obtain the desired result.

2.1 Lower bounds under the oracle model

Klein and Young [KY98] consider the minimum number of iterations needed to find an ϵ-approximate solutions to packing problems of the form **PACK(A,P)** introduced in Chapter 2, under the assumption that we can *only* access P through an "oracle" that solves linear programs over P (and returns the solution vector).

Thus, P cannot be modified, in particular, we cannot impose tight(er) upper bounds on variables. We feel that both models (can or cannot modify P) are of importance. Certainly, we have seen that from a theoretical viewpoint, the ability to tighten bounds on variables yields Frank-Wolfe algorithms that require fewer iterations – in particular, this technique was essential in avoiding a dependency on the width parameter. From a practical standpoint the running time improvement can be substantial, as we will discuss in Chapter 4. On the other hand, there may be settings where the pure oracle model is meaningful. For example, P may be a polyhedral set with exponentially many constraints, such that a combinatorial procedure can efficiently solve linear programs over P, but not so over P with restricted variable bounds.

One consequence of using the oracle model is that at each iteration the support of the current solution grows by one unit. To motivate the technique used in [KY98], consider the following example.

$$\lambda^* = \min \quad \lambda$$

s.t.

$$x_1 + x_2 \leq \lambda \tag{3.37}$$
$$x_2 + x_3 \leq \lambda \tag{3.38}$$
$$x_1 + x_3 \leq \lambda \text{ and} \tag{3.39}$$
$$x \in P = \left\{ x \geq 0 : \sum_j x_j = 1 \right\} \tag{3.40}$$

We have $\lambda^* = 2/3$. Consider a hypothetical Frank-Wolfe algorithm that operates by solving linear programs over P (only). After one iteration, only one variable is positive, and $\lambda = 1$. After two iterations, two variables are positive (at most), and (still) $\lambda = 1$. Thus, we need at least three iteration to obtain a nontrivial approximation to λ^*.

To handle the general case, [KY98] consider problems of the form

$$\lambda^*(A, P) = \min \quad \lambda$$

(R) s.t.

$$Ax \leq \lambda e \tag{3.41}$$

$$x \in P = \left\{ x \geq 0 : \sum_j x_j = 1 \right\} \tag{3.42}$$

where A is a $0-1$ $m \times n$ matrix, such that each entry is an independent random variable equal to 1 with some fixed probability $p > 0$. They show that if B is any submatrix of A containing all the rows but with "few" columns, then $\lambda^*(B,P)$ is significantly larger than $\lambda^*(A,P)$ – larger than by a factor of $1+\epsilon$, if m is of the order of n^2 and p is appropriately chosen. This proves the desired result (provided in more detail below) since as argued in the small example above, when running a Frank-Wolfe procedure that solves linear programs over P the support of the current vector has cardinality no larger than the iteration count.

Now to prove the resulting relating $\lambda^*(B,P)$ to $\lambda^*(A,P)$, note that a value $\lambda^*(M,P)$, where M is $0-1$ and dense (but randomly chosen), has positive probability of being high if M has enough rows. To fix ideas (and as a generalization of the simple example given above), say that M has c columns and $r = c^2$ rows, and that each row of M has only one zero (randomly chosen). For a subset S of the columns, with $|S| = c-1$, the probability that a row of M has a zero in S is $1 - 1/c$. Thus, the probability that every row of M has a zero in S is approximately $e^{-r/c} = e^{-c}$. Hence, the expected number of subsets S of the columns, with $|S| = c-1$, and such that *every* row of M has no zero in S, is at least $c(1-e^{-c}) > c-1$. Since there are c subsets S with $|S| = c-1$ we conclude that there is *some* matrix M as described such that for *every* subset S of columns with $|S| = c-1$, *some* row of M only contains 1*s* in S. Consequently, $\lambda^*(M,P) \geq 1 - 1/c$.

Now returning to the main result, Klein and Young showed that if A is randomly chosen as described above, then with high probability *every* submatrix of A with few columns contains a dense submatrix, which as we just argued produces the desired result. The theorem in [KY98] is:

THEOREM 3.8 *Consider an instance of problem* **(R)** *where* $m = n^2$ *and the entries in A are equal to* 1 *with probability p. Let* $\epsilon > 0$ *be such that* $\frac{1}{p\epsilon^2}$ *grows as (at most) a power of m slower than* 1/2. *Then with high probability the problem has width* $\rho = 1/p$, *and any procedure that computes an* ϵ-*approximate solution to* **(R)** *in the oracle model needs to solve at least* $\rho\epsilon^{-2}\log m$ *linear programs under P.* ■

Note that this lower bound is essentially attained (at least up to poly-logarithmic factors) by some of the algorithms in Chapter 2. At the same time, note that the product $\rho\epsilon^{-2}$ appears both in the lower bound and in the assumption on ϵ; thus the overall bound can be viewed as a $m^{1/2}\log m$ lower bound on the number of iterations although this is

somewhat misleading. The range on ρ and ϵ that the theorem allows is limited; an open area of research concerns proving stronger bounds outside the range permitted by the theorem.

Finally, note that the oracle model (i.e. P remains fixed) is essential toward including the width in the complexity lower bound; when we allow our algorithm to modify P (as when we adjust variable bounds, which we did in the case of algorithm **GK** in Chapter 2, the net effect is to replace width with a combinatorial quantity (e.g. number of columns).

3. The algorithms of Garg-Könemann and Fleischer

The algorithmic developments related to the oblivious rounding technique provided the starting point for a new line of research which still continues, and which has produced the most efficient algorithms for various flow-type problems, including the maximum concurrent flow problem. The ideas described in Section 1 led to new algorithms in [GK98], later improved and extended in [F00]

These algorithms are similar to Algorithm Y in that they employ essentially the same broad scheme, in particular the weight-update formula (3.31) and the idea of gradually increasing the flow of each commodity (and later scaling appropriately). However, the key difference lies in how these algorithms route flow, which can be viewed as a sophisticated technique for avoiding width problems.

Consider, for example, the maximum concurrent flow problem. Recall that Algorithm Y, when handling a given commodity, would route all of the demand of that commodity along a shortest path. This causes the width parameter to appear in our analysis of the algorithms in Section 1 (we could only bypass this issue by using minimum-cost flows).

In contrast, the algorithm for the maximum concurrent flow problem given in [GK98], [F00] routes flow much more carefully. Roughly speaking, the modification to step **Y.1**(a), when considering a given commodity k, is as follows. We compute a shortest path (for the current metric) between s^k and t^k. Then route as much demand as possible along this path, without exceeding any capacity along it, and up to d^k units. Next, we reset the edge-weights. If there is some left-over demand to route (i.e. if the bottleneck capacity on the shortest path is smaller than d^k) then a new shortest path path is computed, and used to route as much of the left-over demand as possible, and so on until d^k units are routed. In more detail,

i. Set $\bar{d}^k = d^k$.

ii. Compute a shortest path P between s^k and t^k; let $u(P)$ denote the minimum capacity on P.

iii. Route $z = \min\left\{\bar{d}^k, u(P)\right\}$ along P.

iv. Reset $\bar{d}^k \leftarrow \bar{d}^k - z$. If $\bar{d}^k > 0$, go to **ii**, else quit.

An additional difference lies in the explicit treatment of dual variables (and of the dual problem). With these changes, the total complexity (not just the iteration count) of the ϵ-approximation algorithm in [F00] is $O^*(\epsilon^{-2}m(k+m))$ (a similar bound is achieved in [GK98]). We refer the reader to [GK98], [F00] for complete details. Recently, Karakostas [Ka02] has elaborated on the ideas in [GK98], [F00], obtaining an algorithm for the maximum concurrent flow problem that converges (up to logarithmic factors) in time $O^*(\epsilon^{-2}m^2)$.

The strategy we just described for routing flow of one commodity can be related to the exponential potential function method with optimal (line-search) step-lengths. Using this method, if a shortest path computation for commodity k produces a path whose minimum capacity is much smaller than d^k, then the step-length computation will effectively reduce the amount of flow of commodity k that we (re)route along this path. The results in [F00] suggest the following alternative to the "round-robin" strategy described in Chapter 2, and which involves choosing a threshold parameter γ (described below): once we start routing a given commodity k, we should stay with it, repeatedly rerouting it until we achieve a cumulative step-length of γ, and only then move to the next commodity in the round-robin order. The spirit of this strategy is to reroute flows at a uniform rate among all commodities, while controlling width, resulting in potential decreases that can be uniformly spread out among commodities. Further improvements (see [F00]) rely on the combinatorial structure of the problem.

The quantity γ equals $\epsilon \log_{1+\epsilon}^{-1}\left(\frac{m}{1-\epsilon}\right)$, which for ϵ small approximately equals $\frac{\epsilon^2}{\log m}$. This was exactly the step-length used in Chapter 2, see eq. (2.36). The critical difference is that we now "gradually" achieve this step-length, while carefully controlling width. A full explanation of the algorithms in [GK98], [F00] in terms of potential functions is an interesting topic for further research.

3.1 The Luby-Nisan algorithm and related work

Here we will briefly mention the algorithm due to Luby and Nisan [LN93], which preceded Young's work and shares some elements with it and with the techniques described in the previous section.

The problem considered in [LN93] is the "positive" linear program

$$\min\left\{c^T x \;:\; Ax \geq b,\; x \geq 0\right\} \tag{3.43}$$

where $c \geq 0$, $b \geq 0$ and $A \geq 0$. It is seen that this linear program is equivalent to one of the form

$$\min\left\{\sum_j x_j \;:\; Ax \geq \mathbf{e}\;,\; x \geq 0\right\} \tag{3.44}$$

where $A \geq 0$ is possibly different from the A in (3.43), and $\mathbf{e}$ is the vector of 1s. Let A be $m \times n$, and note that the dual of (3.44) is $\max\left\{\sum_i y_i \;:\; A^T y \leq \mathbf{e}\;,\; y \geq 0\right\}$.

To obtain an approximate solution to (3.44), [LN93] uses a primal-dual algorithm, which maintains a primal vector x and a dual vector y. Initially, $y_i = 1$, $1 \leq i \leq m$, and $x_j = 0$, $1 \leq j \leq n$.

The typical iteration proceeds as follows:

1. Let $B \subseteq \{1, \cdots, n\}$ denote the set of columns j such that $y^T A$ is "large" (appropriately defined).

2. We update the primal variables as follows. For $j \notin B$ x_j, is left unchanged. For $j \in B$ we reset

$$x_j \;\leftarrow\; x_j + \lambda, \tag{3.45}$$

for appropriately defined $\lambda > 0$ (the same λ is used for all $j \in B$).

3. We update the dual variables as follows: for each i, we reset

$$y_i \;\leftarrow\; y_i e^{-\lambda \sum_{j \in B} a_{ij}}. \tag{3.46}$$

Note that the dual update is exponential in the change of the quantities Ax.

Steps **1**, **2** and **3** are repeated until $\sum_j y_j$ is appropriately small. The final step is to make the solutions feasible. To that effect, we compute $\alpha = \max_i \left\{\sum_j a_{ij} x_j\right\}$ (using the final value of x) and we output the primal solution $\bar{x} \doteq \alpha^{-1} x$. A similar scaling step (but not quite identical, see [LN93]) is used to obtain a dual feasible solution.

In [LN93] it is shown that this primal-dual pair proves ϵ-optimality of v, provided that the set B in step **1**, the quantity λ in step **2**, and the termination condition (in addition to a few other details) are appropriately defined as a function of ϵ. Further, the number of iterations is approximately $O(\frac{\log m \log n}{\epsilon^4})$.

Of more interest here is the relationship of the algorithm to the standard exponential potential function method. At a given iteration t, let x be the current primal vector, and set $\bar{x}^t = \alpha x$. Thus $\bar{x}^t$ is stated in the same scale as the final solution $\bar{x}$. Then it is seen that (3.46) is equivalent to:

$$y_i = y_i e^{-\alpha \sum_j a_{ij} \bar{x}_j^t}, \tag{3.47}$$

i.e. we are using exponential penalties. What is more, the quantity α can be shown to be of the order of $\frac{\log m}{\epsilon}$. Consequently, the penalties employed by the Luby-Nisan algorithm are exactly those used by the exponential potential function method. In fact, the quantity λ in **2** is chosen so as to avoid high width (more precisely: a large fraction of the constraints will see small width using the perturbation in **2**).

It seems likely that a more careful analysis of the Luby-Nisan algorithm will show a close correspondence with the standard exponential potential function method. We note that the algorithm works for pure "covering" linear programs (or, using the dual, for "packing" problems). Recent work by Young [Y01] has generalized these ideas (together with ideas from [GK98], [F00]) to handle simultaneous covering and packing problems.

4. Lagrangian Relaxation, Non-Differentiable Optimization and Penalty Methods

Non-differentiable optimization techniques encompass a wide array of algorithms that can be used to tackle difficult linear programs, and can also take advantage of special problem structure, e.g. block-angular constraints. Even though the field is rich in deep theoretical results, traditionally, the focus of researchers working on these topics has largely been different from the theoretical computer science emphasis on complexity analysis. As a result, even though convergence results for non-differentiable optimization techniques abound, usually these are not easily translatable into provably good bounds for convergence to approximate solutions. Potentially, this may be just a matter of focus or semantics. We discuss these techniques briefly, as a full survey would be outside the scope of this monograph.

The classical non-differentiable optimization tool is subgradient-based Lagrangian relaxation, which has long been popular among optimization practitioners, because of its ease of implementation, general applicability and (potential) computational success. Even though this technique is popular, it is also frequently seen as lacking robustness – different problem instances may require vastly different algorithmic parameters. But this may just be a symptom of inadequate theoretical understanding of

this technique and not of an inherent weakness. As far as we know, there are no theoretical results on Lagrangian Relaxation that are as sharp as some of the theorems we covered in this monograph.

During the last fifteen or twenty years far more sophisticated non-differentiable optimization approaches have been developed. We discuss some of these next.

4.0.1 Bundle and cutting-plane methods

Some interesting results are presented in [LNN95] concerning Bundle-type methods (for example, the Level method). These can be classified as "cutting-plane" algorithms (see below), and are rather more complex (and implementationally far less trivial) than the standard subgradient-based Lagrangian Relaxation. [LNN95] describes cutting-plane algorithms whose running time to achieve error ϵ grow as $O(\epsilon^{-2})$; however these running times also depend exponentially on other problem data. In addition, at each iteration these algorithms add a new inequality to the formulation, and each iteration entails the solution of a quadratic program. Nevertheless, the convergence analyses are quite interesting. As far as we know the results in [LNN95] are the best of this type that can be proved for Bundle methods.

In general, cutting-plane methods are applicable to problems with nonlinear objective, or non-differentiable objective, and linear constraints. For example, in the case of the minimum-congestion problem, we are interested in optimizing a min-max function subject to linear constraints. Cutting-plane methods approximate the objective by algorithmically introducing inequalities that are added to a working formulation. Part of the art (and theory) of these methods lies in adroitly choosing such cutting planes in order to guarantee convergence. See, for example, [GGSV96].

4.0.2 Penalty methods

A different approach is that of penalty methods. Such methods replace a linear program

$$\text{(LP):} \qquad \min \left\{ c^T x \,:\, Ax \leq b \right\} \tag{3.48}$$

with a sequence of usually unconstrained problems of the form

$$\min_x G(x) \;\doteq\; c^T x + \gamma \sum_i f([a_i^T x - b_i]^+), \tag{3.49}$$

where f is a "penalty" function, γ is a parameter which typically is gradually increased, and $z^+ = \max\{0, z\}$. The case where f is quadratic

has received much attention, since in that case the Hessian of $G(x)$ can be explicitly computed. See, for example, [FG99].

Cominetti and San Martin [CM94] (also see Cominetti and Dussault [CM94]) consider the case where f is *exponential*; more precisely their unconstrained problem is of the form

$$U(\delta): \qquad \min_x c^T x + \delta \sum_i e^{\frac{a_i^T x - b_i}{\delta}}. \tag{3.50}$$

They show that there is an optimal solution x^* to (LP) the solution to $U(\delta)$ is of the form

$$x(\delta) \doteq x^* + d^*\delta + O(e^{-\frac{\mu}{\delta}}), \tag{3.51}$$

where d^* is a fixed vector and $\mu > 0$ is a fixed parameter. Thus, (see [CM94]) algorithms can be devised that gradually let $\delta \to 0^+$ and thus achieve convergence to x^*.

It is difficult to gauge how close these results are to proving polynomial-time approximate convergence: the convergence theorems in [CM94], [CM94] are all in terms of *absolute* errors. Also note that by (3.51), for δ small $c^T x$ approximately equals $c^T x^* + (c^T d^*)\delta$. Hence, in order to guarantee *relative* error ϵ we would have to choose δ dependent on the optimal objective value and on the quantity $c^T d^*$, which might be difficult to estimate.

There are other (significant?) ways in which the exponential penalty approach in [CM94] differs from the exponential potential function method we have discussed in prior sections. Notably, there is no notion of a "budget" in the unconstrained problem $U(\epsilon)$ and it is not clear how the rate at which δ is decreased reflects (or is affected by) problem size. Nevertheless, the similarities are undeniable, and perhaps a common ground could be established.

4.0.3 The Volume algorithm

Recently, Barahona and Anbil [BA00] have presented an interesting variation on the subgradient method for Lagrangian Relaxation which includes two key ingredients. First, the algorithm in [BA00] maintains a primal vector $\hat{x}$; at each iteration when a Lagrangian is solved (with solution y, say), $\hat{x}$ is updated using a formula of the form

$$\hat{x} \leftarrow (1-\tau)\hat{x} + \tau y, \tag{3.52}$$

where $0 \leq \tau$ is chosen so to minimize (for example) the total square violation of the relaxed inequalities by (the new) $\hat{x}$. Thus, rule (3.52) can be

seen as an implementation of (half of) a Frank-Wolfe procedure. Second, it is the vector $\hat{x}$ that is used to compute the new Lagrange multipliers (and not y) through the standard subgradient update technique. There is some theoretical justification for the use of a formula such as (3.52) to update primal variables. This is the (very elegant) "volume" theorem proved in [BA00] and outline below. Finally, this is an approach that seems to enjoy some computational success.

In [BA00], the authors study optimization problems **DCG** of the form,

$$\begin{aligned} z^* = \min \quad & z \\ \text{s.t.} \quad & \\ z + a_i^T x \quad & \leq \quad b_i, \quad 1 \leq i \leq m, \end{aligned} \tag{3.53}$$

for appropriate vectors a_i and b. This problem can be seen as the dual of a "column generation" problem; i.e. the dual variables to constraints (3.53) are the column weights in a column generation formulation. Thus, it is of interest to study the structure of optimal dual variables to **DCG.** The following theorem is proved in [BA00].

THEOREM 3.9 *Let $\bar{z} < z^*$ with $z^* - \bar{z}$ small. Let $I \subseteq \{1, 2, \cdots, m\}$ be the subset of inequalities (3.53) that are active at optimality. Then for $i \in I$, the optimal dual variable for constraint i is proportional to the measure of the set*

$$\begin{aligned} z + a_i^T x \quad &= \quad b_i, \\ z + a_j^T x \quad &< \quad b_j, \quad \forall j \in I - i, \\ \bar{z} \leq z \leq z^* & \end{aligned} \tag{3.54}$$

i.e. the measure of the face arising from constraint i, between $\bar{z}$ and z^*.

Note that one of the constraints in the dual to **DCG** is that the dual variables are nonnegative and sum to 1. Also note that the measure of the face that corresponds to constraint i is proportional to the volume of the polyhedron P_i located "between" face i and the hyperplane $z = \bar{z}$. Consequently, we may view the i^{th} optimal dual variable as representing the probability a randomly chosen point near the optimum lies in P_i. In [BA00] this idea is used to motivate an elaboration to the standard subgradient-based Lagrangian relaxation method which, in addition, produces primal solutions. Some similar ideas can be found in earlier work of Wolsey (unpublished).

To complete this section we theorize that it may be possible to view, at least approximately, Lagrange multipliers obtained through the familiar subgradient optimization formula as the gradient of an appropriate (smooth) potential function. This conjecture, if valid, might lead to a more satisfactory convergence analysis than now available.

Chapter 4

COMPUTATIONAL EXPERIMENTS USING THE EXPONENTIAL POTENTIAL FUNCTION FRAMEWORK

In this chapter we will describe our experiments using the exponential potential function framework. The focus of our work has been to produce a general-purpose implementation of an algorithm for approximately solving block-angular linear programs. All of the data sets for which we present numerical results include a routing component. While routing is not a real-life synonym for block-angularity, it is certainly the case that many practical models involve networks and commodities to route, in addition to other features. Further, our problem instances are all derived from or motivated by problems in networking and telecommunications – in future work we plan to test our implementation on problems arising in logistics.

Nevertheless, the instances we consider are fairly disparate and in many cases include non-routing features. Our implementation handles all these instances through a common interface, without requiring any a-priori combinatorial information: the input simply consists of the values of the coefficients in the constraint matrix, and the block structure.

More precisely, we consider a block-angular linear program **(LP)**

$$z^* = \min \quad c^T x$$

$$Ax \leq b \tag{4.1}$$

$$x \in P = P^1 \times P^2 \times \cdots \times P^K \tag{4.2}$$

where the P^k are polyhedra over which it is "easy" to optimize a linear function. For convenience in the presentation, we also assume that $b_i > 0$ for every i.

As discussed in Chapter 2 the block-angular structure can be leveraged by the methodology we described to obtain a theoretically fast approximation algorithm – as measured by the number of iterations. In

our implementation we have placed special emphasis on minimizing the number of iterations, but little or no attention is paid to making the individual iterations especially fast. In fact, a low-level but salient feature of our implementation is that it treats each block P^k as a general linear system, with block iterations solved by calling a general-purpose LP solver. This deliberate algorithmic strategy, we believe, adds robustness to the implementation while focusing attention on the most critical issue: keeping the iteration count low. Finally, the key design ingredients in our system is that it should be "choice" and "parameter" free from the point of view of a user – our implementation does not take any algorithmic options from the user.

0.1 Remarks on previous work

Previous experiments using the exponential potential function methodology (see [LSS93], [GOPS98], [GK94]) focused on pure multicommodity flow problems and concurrent flow problems, resulting overall in quite substantial improvements over general-purpose linear programming codes. We will describe some of the features of these implementations later. [A00] presents results using the ideas of [GK98] as applied to a problem in VLSI design.

A reading of the first three references shows three related areas where further improvements are possible. We comment on these next.

1 A central component of these implementations is that they exploit special structure of the polyhedra P^k so that each block iteration runs fast. In fact, the algorithms discussed in these references can only be applied to multicommodity flow problems, where linear programs with feasible set P^k are either shortest-path problems or (single commodity) minimum-cost flow problems. Accordingly, all of these implementations rely, rather heavily, on using a fast minimum-cost flow subroutine.

   However, reducing the overall number of block iterations is arguably a much more important algorithmic objective than making each iteration fast. This is certainly the case if the number K of blocks is large and if all the blocks are roughly of the same size. The second assumption is generally true for multicommodity flow problems, and the first one is true for *real-world* multicommodity flow problems . Should this scenario arise, then each block accounts for a diminishing fraction of the overall problem size (as, say, both the number of nodes and the number of commodities increase) and a theoretical improvement in the per-block running time will yield decreasing returns. At the same time, a reading of [LSS93], [GOPS98], [GK94]

same time, a reading of [LSS93], [GOPS98], [GK94] shows that the actual number of block iterations can become quite large, and that it can grow rather rapidly with problem *size*. These observations, in our view, underscore the importance for achieving small iteration count. Several features of our implementation seek to address this point, and achieve experimental success.

2 There is an additional price to be paid for having an implementation that requires the blocks to have a particular structure. Or, to put it differently, there are great advantages to an implementation that treats each block as an arbitrary linear program. As we argue below, there are many instances of practical importance where by carefully modifying the blocks we obtain a new system which is much more effectively handled by the exponential potential function method. Starting from a multicommodity flow problem, for example, the new blocks will *not* be minimum-cost flow problems. The per-block iteration time becomes comparatively more expensive, but both the overall iteration count and the precision of the algorithm are greatly enhanced. As it turns out, even though our implementation solves the blocks as general LPs, it actually seems to achieve faster running times, especially for large problems.

   There is also a strictly practical consideration that favors the use of a general-purpose block solver. It is fairly common for a practical model to significantly "change" during its lifetime. What starts out as a pure multicommodity flow sub-model may change, so that, for example, only certain commodity-dependent paths can be used to route flow. It is quite critical that an optimization module should nimbly adapt to such (sudden) changes, even at the cost of reduced performance.

3 An additional weakness of the basic framework, which is hinted at by the numerical results in the above references, and which is quickly found out in experimentation, is that the lower bounds proved by the algorithm can improve rather slowly. We describe below a theoretical algorithmic construct that yields provably stronger lower bounds than those obtained in [PST91] and which in some cases allow us to (numerically) reach provable optimality.

In addition to focusing on the above issues, our implementation exploits an inherent strength of the methodology which, while theoretically difficult to explain, can be clearly observed in experiments – the algorithm quickly zeroes in on "good" columns.

0.2 Outline of a generic implementation

For completeness, we next provide an outline of the basic exponential potential function algorithm, as applied to linear program **LP**. This largely follows the ideas in Chapter 2. In later sections we describe how we have modified this basic implementation. In summary, the algorithm approximately solves a sequence of feasibility systems, each consisting of the original constraint system plus one "budget" constraint made up by the objective row, and whose right-hand side value is a "guess" on the optimal objective value. The algorithm applies binary search on this guess so as to (approximately) optimize.

More precisely, given $\epsilon > 0$, the algorithm seeks to find a vector $x \in P^1 \times P^2 \times \cdots \times P^K$ which violates constraints (4.1) with maximum relative error at most ϵ and whose objective value is at most $(1+\epsilon)$ times the value of **LP**, i.e. x is an ϵ-feasible, ϵ-optimal vector. To this end, given a scalar B, the algorithm considers feasibility systems $FEAS(B)$ of the form.

$$c^T x \leq B \tag{4.3}$$
$$Ax \leq b \tag{4.4}$$
$$x \in P^1 \times P^2 \times \cdots \times P^K, \tag{4.5}$$

and either finds an ϵ-feasible solution to this system or determines that it is infeasible. The set of iterations that accomplish this goal is called a B-phase. Postponing a detailed description of a B-phase, the main algorithm is as follows:

Main Algorithm

Step 0. Initialization. We start with $\underline{B} = -\infty$ and a valid upper bound $\bar{B}$ on the value of **LP**.

Step 1. Set $B = \mu\underline{B} + (1-\mu)\bar{B}$, where $0 < \mu < 1$ is an appropriately chosen parameter. Run a B-phase to handle problem $FEAS(B)$:
(a) If $FEAS(B)$ is infeasible, set $\underline{B} = B$.
(b) If $FEAS(B)$ is ϵ-feasible, set $\bar{B} = B$. As we will see in Section 3, this step is modified in our implementation, so that the solutions that yield our upper bounds to **LP** are fully feasible, not just ϵ-feasible.

Step 2. If $\bar{B} \leq (1+\epsilon)\underline{B}$, exit. Otherwise, go to Step 1.

END

A smaller initial value $\bar{B}$ than the choice in **Step 0** can be used, provided $FEAS(\bar{B})$ is known to be ϵ-feasible. This goal is achieved by running

the exponential potential function method on the system (4.1), (4.2) directly. This description of the Main Algorithm is a bit simplified; in our implementation the value ϵ is gradually approximated (i.e. Steps 1 and 2 are overlaid with ϵ-scaling).

Next we describe the *B*-phases. Each *B*-phase starts with some $x \in P$ which is δ-feasible to *FEAS*(*B*) for some $\delta > 0$.

B-Phase

Step I. Set $\delta = \max\left\{\frac{\delta}{2}, \epsilon\right\}$.

Step II. Core Procedure. Run exponential potential reduction procedure to determine whether *FEAS*(*B*) has a δ-feasible solution or is infeasible.

(a) If *FEAS*(*B*) is infeasible, exit;

(b) Otherwise, if $\delta > \epsilon$ go to Step 1, and if $\delta = \epsilon$, exit.

END

The core procedure (Step II) embodies the key algorithmic components of the exponential potential framework. Given a parameter $\alpha > 0$, this procedure iterates through vectors x^h, $t = 0, 1, \cdots$ contained in P. We write $x^{h,k}$ for the restriction of x^h to P^k, $1 \leq k \leq K$, and we denote by m the number of rows in A. At iteration h we proceed as follows:

Block Iteration

1. Choose a block k, $1 \leq k \leq K$.

2. For $1 \leq i \leq m$, set

$$\pi_i = \frac{1}{b_i} e^{\alpha \frac{\sum_j a_{ij} x_j^h}{b_i}} \tag{4.6}$$

and let $g_{h,k}^T$ denote the restriction of the (row) vector $\pi^T A$ to the coordinates in block k.

3. Solve the linear program

$$\min\left\{g_{h,k}^T x : x \in P^k\right\}$$

with solution v^h.

4. For appropriate $0 \leq \tau \leq 1$, set

$$x^{h+1,k} = \tau v^h + (1 - \tau) x^{h,k} \tag{4.7}$$

and $x^{h+1,\hat{k}} = x^{h,\hat{k}}$ for all $\hat{k} \neq k$.

As discussed in Chapter 2, the choice of stepsize τ and the block index k are important aspects of the overall algorithm. Further, in the above description we have left out the lower bounding method, i.e. how to detect case (a) of Step II of the B-Phase procedure – we will describe how we prove lower bounds later.

The rest of this chapter is organized as follows. In Section 1 we analyze some low-level issues that arise when implementing the exponential potential method. In Section 2 we discuss a general method for improving lower bounds obtained by a Lagrangian procedure.

1. Basic Issues.

When implementing the generic method described above, one must make a number of low-level decisions; namely choosing the parameter μ used in the binary search in the main algorithm, choosing the stepsize τ in each block iteration, choosing the parameter α in the exponential function, and choosing the block k to be used in each block iteration.

As we have seen in Chapter 2, there are theoretically correct choices in all four cases. Further, these issues have already been considered in previous experimental work. In this section we describe our choices, primarily because they differ, sometimes markedly, from those of prior authors. At the same time, with a couple of exceptions which will be noted, none of our choices should be viewed as hard-and-fast rules. Rather, they are the outcome of our experimentation, reflecting a compromise between speed and numerical stability.

1.1 Choosing a block

As discussed in Chapter 2, the choice of block k to be used in each block iteration has important theoretical ramifications. Radzik's "round-robin" approach is theoretically sound and easy to implement, and was implemented e.g. in [GOPS98]. At the same time, recent theoretical work ([GK98], [F00]) and common sense suggest making a choice that in some regard guarantees a large potential decrease.

The round-robin approach should work well on large uniform problems (such as random multicommodity flow problems) but might have difficulty handling problems with marked differences between blocks. On the other hand, a "best-block" approach might quickly reduce the disparities between blocks, in the sense that all block choices may end up yielding roughly the same potential decrease.

In our implementation, odd iterations use the round-robin approach, and even iterations implement a best-block approach, as follows: at any iteration we keep track of a certain block $\hat{k}$ which, when last used, yielded

a potential decrease Δ. If this is an even iteration, we run block $\hat{k}$ and we update Δ. If this is an odd iteration, we run the block k chosen by the round-robin approach, which results in a potential decrease Γ, and if $\Gamma > \Delta$ we reset $\hat{k} \leftarrow k$ and $\Delta \leftarrow \Gamma$.

Experimentally, this combined approach has resulted in increased robustness and visible (if small) reductions in the number of iterations. Some practical problem instances we have looked at contained vastly different blocks (in particular, one block that was different from, and much more meaningful than all the other blocks) in which case the best-block approach was essentially mandatory.

1.2 Choosing τ

From a theoretical standpoint, the stepsize τ to be used in each block iteration can be given a pre-assigned value that guarantees convergence, as discussed in Chapter 2. However, this choice may be overly conservative, and it may be difficult to implement if it contains an explicit estimate for the width (as it is in the algorithm in [PST91]). Some basic experimentation quickly proves the benefits of choosing τ so as to yield a maximum potential decrease, i.e. by carrying out a line-search to minimize $\sum_i e^{\alpha \frac{\sum_j a_{ij}(x_j^h + \tau(v^h - x^h))}{b_i}}$, where x^h is the current iterate and v^h is the vector computed in Step 3 of the current block iteration.

It might at first appear to the reader that an approximate line-search would suffice. Our experiments convinced us otherwise. In fact, our overall implementation (of the Main Algorithm) often failed to converge unless the line-searches were carried out to relatively high accuracy.

In [GOPS98] the authors use Newton's method to drive the line-search. Further, each line search was "warm-started" using the stepsize computed in the last block iteration using the same block. It should be noted that computing the optimum τ entails minimizing a function of the form $\sum_i \gamma_i e^{\beta_i \tau}$, where the γ_i, β_i are parameters, and thus each iteration of Newton's method is relatively expensive (it involves all rows). In addition, the β_i may be very small or very large numbers, possibly resulting in (a) a slowly converging Newton's method, or more important, (b) a Newton's method that fails to converge or to correctly converge due to numerical instabilities. These numerical issues are synergistic with the choice of the parameter α, see below. Furthermore, the stepsize procedure has to behave properly in the event that the optimal stepsize is zero, or numerically zero – note that zero stepsize, in the case of an iteration using all the blocks, indicates that the current iterate minimizes potential.

The successful experience in [GOPS98] notwithstanding, we were unable to obtain an accurate and reliable algorithm by using Newton's method only. In our implementation we combine Newton's method with binary search. We first run binary search to estimate τ to some desired estimate (two digits of accuracy). Then we run Newton's method, starting with the estimate produced by the binary search, to compute an estimate $\hat{\tau}$ with three digits of accuracy. Next, we compute the potential decrease using stepsize $\hat{\tau}$. If this decrease is either negative (indicating numerical error) or if this decrease is inferior to that obtained using the value of τ computed last time we used the same block, then we rerun Newton's method, this time requiring five digits of accuracy, and obtaining a new estimate $\bar{\tau}$. This estimate is evaluated, and if it does not yield a decrease in the potential function then the stepsize procedure returns with a stepsize of 0. If a complete cycle of round-robin block iterations yields a stepsize of 0 for every block, then our implementation runs one iteration using all blocks at the same time, and if the resulting stepsize is 0 the code terminates.

1.3 Choosing μ

Here we discuss the choice of the parameter μ, used in Step 1 of the Main Algorithm, in the binary search for the optimum objective value. From a theoretical standpoint, the choice $\mu = 0.5$ should be "good enough". However, our experiments indicate that this may not be a robust choice. In fact, on selected "difficult" problems our code will not converge when using $\mu = 1/2$.

To see why this may happen, consider a problem **(LP)** where the matrix A has a large number of rows, and consider an iteration of Step 1 of the Main Algorithm, resulting in case (b). At the end of this iteration we will therefore have a δ-feasible point x^h with $c^T x^h = \bar{B}$, for some $\delta > 0$. In the next B-phase the (new) budget constraint will be of the form $c^T x \leq B \ (= \frac{1}{2}(\underline{B} + \bar{B}))$. Consequently, at the very start of this B-phase the relative violation of the budget constraint (i.e. $c^T x^h / B$) is much higher than all the other constraint violations. This has two adverse consequences. First, the nature of the exponential potential function is such that the algorithm will try to immediately even out the infeasibilities among the various constraints, which will typically result in a larger average infeasibility than prior to the budget change. Thus, the effort that was expended in the prior execution of Step 1 has to some degree gone to waste. Second, if α (the parameter in the exponential function) is large, there may be loss of precision in the evaluation of the penalty function and in the computation of the stepsize.

Yet another reason why the choice $\mu = 1/2$ may be undesirable is that, when the difference between $\underline{B}$ and $\bar{B}$ is large (as might be the case with a difficult problem instance), the system $FEAS(\frac{1}{2}(\underline{B} + \bar{B}))$ may be infeasible. The attempt to find an approximately feasible solution to $FEAS(\frac{1}{2}(\underline{B} + \bar{B}))$ is a "mistake": in a sense, in order to prove infeasibility we may have to go "past" the desired value of ϵ. From an experimental point of view, it is quickly observed that such mistakes are computationally costly.

In summary, it appears that a choice $\mu > 1/2$ may be preferable. In our implementation, we use $\mu = 9/10$. One practical benefit arising from this choice is that, after two "mistakes" we will have a proved 1% error bound, i.e. $\bar{B} \leq (1.01)\underline{B}$.

1.4 Choosing α

We have seen in Chapter 2 that setting α of the form $\frac{c \log m}{\epsilon}$, where $c > 0$ is a small parameter (say $c = 3$) and m *is* the number of constraints, yields a provably good algorithm. In practice this choice may not be ideal, primarily because it may prove too large – in the sense that the exponential function becomes badly behaved. [GOPS98] describes an adaptive rule for controlling α. Here we describe a describe a different procedure.

One way to observe why a large value of α may impede practical convergence is to consider, is to recall that the exponential potential reduction method is a first-order method, and therefore may be slow to converge when the Hessian matrix of the potential function is ill-conditioned [Lu]. Consider a feasibility system

$$Hx \leq d \tag{4.8}$$
$$x \in P, \tag{4.9}$$

(where H has M rows and $d > 0$) for which we seek an ϵ-feasible solution. and the system

$$z \leq \mathbf{1} \tag{4.10}$$
$$(x, z) \in \hat{P} \tag{4.11}$$

where 1 is the vector of m 1's, and

$$\hat{P} = \left\{(x, z) : x \in P, z_i = \frac{h_i x}{d_i}, \ 1 \leq i \leq M\right\} \tag{4.12}$$

where h_i is the i^{th} row of H. For $x \in P$, z is simply the vector of relative violations of $Hx \leq d$. The two systems are equivalent in that the

exponential potential reduction method will generate the same sequence of iterates in both cases. Consider the second system at iteration k. Then it can be seen that the Hessian of the potential function is proportional to $diag\{e^{\alpha z_1}, \cdots, e^{\alpha z_M}\}$. Consequently, we can expect that if there is a large disparity between violations of different constraints, and if α is large, then a steepest-descent algorithm for minimizing the potential function may converge slowly. [To be more precise, one should look at the *projected* Hessian (projected to $\hat{P}$) but we would expect that matrix to be no less ill-conditioned.] For ϵ small enough, $\alpha = \frac{\log M}{\epsilon}$ could certainly be quite large.

We are left with a difficult choice: which piece of mathematics do we believe? The working answer is that we believe both, but not at the same time. We use the following rules for setting α.

1 Initially we expect classical first-order algorithm behavior (i.e. rapid convergence to a reasonably accurate solution). Here we use $\alpha = \frac{3 \log m}{\epsilon}$. This rule is also used every time that we shift to a new value of ϵ (e.g. when using ϵ-scaling).

2 As soon as the code determines that we are "close" to the optimum, α is reduced. More precisely, our code includes periodic calls to a procedure that will be described below, which we will refer to as the *NX* procedure. *NX* takes as input the current iterate x^h and returns a new iterate $x^{h+1} \in P$. When a call to this procedure is successful, (a) x^{h+1} has better objective value and usually much smaller potential than x^h, and (b) we have a tighter lower bound. When (a) holds we reduce α by 0.9, unless the lower and upper bounds are within 1% of each other, in which case we reduce α by 0.1. The logic here is that the basic algorithm may have converged far enough that we are entering the stage where a first-order method for minimizing potential may be ineffective, in which case bad conditioning of the Hessian would play a negative role.

3 When α is too small, the minimizer of the potential function will be potentially deep in the relative interior of the feasible region. Essentially, then, we may be wasting iterations converging to a point that is suboptimal for **LP.** The algorithm detects this situation when there is an iteration that (a) worsens the objective value and (b) produces a feasible iterate (maximum violation less than $1.0e^{-10}$). In such a situation, α is increased by a factor of 2.

In our experiments, these rules provided a reasonably robust trade-off.

2. Improving Lagrangian Relaxations

According to our experiments, the issue that emerges as most critical when implementing the exponential potential framework, is how to prove tight lower bounds quickly. Indeed, it appears that the basic framework is not always able to fulfill this function effectively, in practice.

In Chapter 2 we saw how the algorithm in [PST91] proves lower bounds for **LP** (eqs. (2.41)-(2.44)). For convenience, we briefly outline the approach here. Recall that z^* denotes the value of **LP**. If we can prove that $\lambda > 1$ whenever the system

$$\begin{aligned} c^T x &\leq \lambda B && (4.13)\\ Ax &\leq \lambda b && (4.14)\\ x &\in P, && (4.15) \end{aligned}$$

is feasible, then we have proved that $z^* > B$. More precisely, is $\pi_0 > 0$ is a scalar, and $\pi > 0$ is a an m-vector, then defining

$$\bar{\lambda} = \frac{\min_{x \in P} \left\{ (\pi_0 c^T + \pi^T A)x \right\}}{\pi_0 B + \pi^T b}. \qquad (4.16)$$

we have that $\bar{\lambda}B$ is a lower bound on z^*. This is the approach used in [PST91], with exponential penalties playing the roles of the π_i. Even though this approach is theoretically sound (e.g. the algorithm in [PST91] is a fully polynomial-time approximation scheme, and it relies on this bound to yield the termination condition) in practice the bound may not be as tight as desired. Furthermore, the computation of the bound is expensive as it involves optimizing over all blocks. Thus, having gone through the effort to compute this lower bound, it may pay off to invest some additional computational time to tighten it.

First, note that (4.16) can be rewritten as

$$\bar{\lambda}(\pi_0 B + \pi^T b) - \pi^T b = \min_{x \in P} \left\{ (\pi_0 c^T + \pi^T A)x - \pi^T b \right\} \qquad (4.17)$$

which, since $\pi_0 > 0$, can be restated as

$$\bar{\lambda}B + (\bar{\lambda} - 1)\frac{\pi^T b}{\pi_0} = \min_{x \in P} \left\{ (c^T + \frac{\pi^T}{\pi_0} A)x - \frac{\pi^T}{\pi_0} b \right\} \qquad (4.18)$$

The function minimized in the right-hand side of (4.18) is a Lagrangian, which therefore is a lower bound on the value of LP3. Since $\bar{\lambda} > 1$, and $\pi > 0$, $b \geq 0$ and $\pi_0 > 0$ we have that this lower bound is *strictly stronger* than $\bar{\lambda}B$.

In fact, we do not have to stop here, and a much stronger bound can be frequently obtained. To this effect, we describe next a technique for tightening Lagrangians. Suppose that we assign dual variables $v \geq 0$ to constraints (4.1) of **LP.** The resulting Lagrangian is:

$$L(x,v) = (c^T + v^T A)x - v^T b \tag{4.19}$$

$$\min\{L(x,v) : x \in P\} \leq z^* \tag{4.20}$$

Suppose $\hat{x}$ minimizes the Lagrangian in (4.20) and let $L^*(v)$ be the resulting optimal value. The observation to be made is that if

$$c^T\hat{x} > L^*(v) \tag{4.21}$$

then we should be able to obtain a stronger lower bound on z^* than $L^*(v)$. In essence, when this situation arises the multipliers v are poor approximations to the optimal dual variables for constraints (4.14).

When this situation arises a simple procedure can be applied which will usually yield superior dual variables. For *fixed* v, and any real U denote:

(**LP**(v,U))

$$F_v(U) = \min \quad L(x,v)$$
$$\text{s.t.}$$
$$c^T x \leq U \tag{4.22}$$
$$x \in P \tag{4.23}$$

The intuition here is that if U were an *upper* bound on z^* then constraint (4.22) would be redundant. Now we have

LEMMA 4.1 *For any* U, $\min\{F_v(U), U\} \leq z^*$.

Proof. Note that $F_v(z^*) \leq z^*$, since with $U = z^*$, the linear program **LP(v,U)** is simply a relaxation of LP. Now let $K = \min\{F_v(U), U\}$, and assume by contradiction that $K > z^*$. Then $F_v(U) \leq F_v(K) \leq F_v(z^*) \leq z^*$, ∎

As a result of this lemma, we are led to the problem of finding U^v that maximizes $\min\{F_v(U), U\}$ (we remind the reader that v is fixed). Since $F_v(U)$ is a continuous, monotonely nonincreasing function of U, we have that U^v exists, is unique and satisfies $F_v(U^v) = U^v$.

One approach to computing U^v would be to apply binary search (over U) to $F_v(U)$. However, computational experience suggests **LP(v,U)** may be difficult to solve by direct methods even if we have a fast algorithm for solving linear programs over P. [GK96] describes a technique

that we could use to approximately solve $LP(v,U)$. However, an alternative approach is that of (simultaneously) searching for U^v and the corresponding optimal dual variable for constraint (4.22), and this turns out to provide a much better insight.

Following this approach, for given U and $\gamma \geq 0$ (this value being a guess for the *negative* of the dual variable for (4.22)), we have that

$$\min_{x \in P} \left\{ L(x,v) + \gamma c^T x - \gamma U \right\} \leq F_v(U), \tag{4.24}$$

and consequently

$$\min_{x \in P} \left\{ L(x,v) + \gamma c^T x \right\} \leq F_v(U) + \gamma U, \tag{4.25}$$

or equivalently,

$$\min_{x \in P} \left\{ L(x, \frac{v}{1+\gamma}) \right\} \leq \frac{F_v(U) + \gamma U}{1+\gamma}. \tag{4.26}$$

Since $F_v(U^v) = U^v$, we have that for any $\gamma \geq 0$,

$$\min_{x \in P} \left\{ L\left(x, \frac{v}{1+\gamma}\right) \right\} \leq U^v. \tag{4.27}$$

Finally,

LEMMA 4.2 *Let γ^v be the negative of the optimal dual variable for constraint (4.22) in $LP(v,U^v)$. Then*

$$\begin{aligned} \max_{\gamma \geq 0} \min_{x \in P} \left\{ L\left(x, \frac{v}{1+\gamma}\right) \right\} &= \min_{x \in P} \left\{ L\left(x, \frac{v}{1+\gamma^v}\right) \right\} = U^v \\ &= \max_{U} \min \left\{ F_v(U), U \right\}. \end{aligned} \tag{4.28}$$

Proof. This follows from the fact that U^v and γ^v satisfy (4.24) with equality. ■

In other words, we must maximize (as a function of nonnegative γ) the left-hand side of (4.27), which we denote $g_v(\gamma)$. Let v^j, $j \in J$ be the set of extreme points of P. We have that

$$g_v(\gamma) = \min_{j \in J} \left\{ c^T v^j + \frac{v^T}{1+\gamma}(b - Av^j) \right\}. \tag{4.29}$$

For each j, the quantity in the right-hand side of the above equation is, as a function of γ, either constant, or strictly increasing, or strictly decreasing. Thus, $g_v(\gamma)$ is a unimodal function of γ, and we can maximize it using binary search over γ, provided that an upper bound on γ^v is

known. Here, each iteration in the binary search involves the solution of a linear program over P.

In our implementation we use this Lagrangian strengthening procedure. Note than an invocation of this procedure could be rather expensive – it entails several optimizations over all blocks P^k. Thus, we use it sparingly, in conjunction with a different procedure described in the next section. Nevertheless, we obtain quite visible improvements on the lower bound. At the same time, we wish to stress that it is not clear whether similar improvements should be expected if the procedure were to be used in conjunction e.g. with standard subgradient optimization method, which tend to produce "good" dual variables: instead, the procedure should be viewed as a method for refining "poor" dual variables.

3. Restricted Linear Programs

Linear programming models arising from practical applications can be extremely large, but in many cases this is due to the mathematical intricacy of the modeling, as opposed to the the actual size of the underlying "real" problem. To put it differently, a model may try to capture all possible combinations of several events, while in fact only a small number of combinations will be realized in a good solution. In addition, variables are linked through non-random logical constraints, e.g. variable upper-bound inequalities, with the consequence that if a certain variable is zero, then a number of other variables will also be zero (consider, for example, location models). In other words, the model may contain a very large number of variables, but only a small fraction of these will be positive at optimality.

Should this situation arise, it is clear that a substantial speedup would ensue if an algorithm could effectively "guess" the support of a near-optimal solution. What is quite surprising is that even when the support Z^* of the optimal solution is relatively large the linear program *restricted* to the columns in Z^* can be solved much faster (sometimes astoundingly faster) than the original problem. This situation certainly arises with routing or network design linear programs, as we will see later in Table 4.10.

Of course, in order to leverage the advantage of a small Z^*, an algorithm would have to discover a near-optimal support. **The exponential potential function framework, together with effective width control, seems to rapidly find near-optimal supports.** By this we mean that after a small number of round-robin rounds the set of variables with positive values will be (a) small and (b) near-optimal (if infeasible). A theoretical explanation for this effect seems difficult. It

may also be a general property of Lagrangian-type algorithms. In any case, it is a property that we exploit to **accelerate** our algorithm.

We make use of the following procedure, which is periodically called (more on this below), interspersed with block iterations, and which we use to modify **Step 1(b)** of the Main Algorithm so that the solutions output by our implementation are feasible (not simply ϵ-feasible). For convenience, we assume that all variables are nonnegative. In the following description, $\underline{B}$ and $\bar{B}$ denote the current lower and upper bounds on the value of **LP.**

Procedure NX

nx.0. After block iteration h, $Z(h) = \left\{j : x_j^h \geq \theta \sum_j x_j^h\right\}$ and $\theta > 0$ is a parameter. [In our implementation we used $\theta = 1.0e^{-10}$]. Let **LP(h)** be **LP,** restricted to columns in $Z(h)$, but containing all constraints.

nx.l. Solve **LP(h).** If infeasible, quit *NX.* Otherwise, let y^h be its optimal solution and π^h the vector of optimal dual variables corresponding to inequalities (4.1).

nx.2. Set $x^{k+1} = \beta y^k + (1 - \beta)x^k$, where $0 \leq \beta \leq 1$ is chosen so that x^{k+1} minimizes potential in the segment $[x^k, y^k]$.

nx.3. Compute $B^k = \max_{\gamma \geq 0} \min_{x \in P} \left\{L(x, \frac{\pi^k}{1+\gamma})\right\}$. Where $\underline{B}$ is the current lower bound on the value of **LP**, we reset $\underline{B} \leftarrow \max\left\{B^k, \underline{B}\right\}$.

END

An important point regarding this procedure is that it is *not* intended as a crossover step. In fact, typically, x^{k+1} will be far from optimal. On the other hand, given that point y^k is feasible (and, frequently, $c^T y^k < c^T x^k$) the potential at x^{k+1} will be far smaller than at x^k. In other words, *NX* will typically accelerate convergence. To put it differently, we are using a subroutine call to an "exact" LP solver to enhance the performance of our approximate LP solver.

A call to this procedure can be relatively expensive primarily because of **nx.3**, but typically not, in our testing, because of **nx.l.** Nevertheless, the benefits of the Lagrangian strengthening procedure and of using optimal duals arising from the "good" set $Z(h)$ are compounded, yielding to sometimes dramatically improved bounds in a single iteration. In our implementation, *NX* is called just after block iteration *tK/2*, where $t = 1, 2, 3, \cdots$, provided the maximum relative infeasibility is "not too large" (in our implementation, at most $3e^{-02}$) and otherwise the call is

skipped. The rationale behind these rules are (a) after many repeated calls, the marginal utility of the procedure tends to diminish, and (b) if the current iterate is very infeasible the support $Z(h)$ may be quite poor – in particular, **LP(h)** may be infeasible.

Another important implementation detail concerns the choice of algorithm for **nx.1.** The default rule is that we use the barrier algorithm – in our testing it provided the best mix of speed and reliability. And we use it without crossover: by far this was the better choice with regards to **nx.3.** We apply these default rules until the lower and upper bounds we have proved for **LP** are "close" (less than $1.0e^{-02}$), in which case we use the dual simplex method, warm-started at the current iterate (for the primal solution) and using the dual variables from the last call to *NX* (for the dual solution).

Finally, in our implementation we use the solutions produced by *NX* as the only source for upper bounds – consequently, our upper bounds are "feasible", at least to the tolerance of the underlying linear programming solver.

In Section 5.6 we discuss experimental data supporting the use of Procedure *NX*.

4. Tightening formulations

A type of constraint that tends to abound in many practical model is the "variable upper bound" or VUB constraint. This is a constraint of the general form,

$$\sum_{k \in K} f_j^k - x_j \leq u_j, \tag{4.30}$$

where we have such an inequality for each j in some index set J. Network design problems are a typical example: here the f_j^k are commodity flows and the x_j is a capacity variable (there may be many capacity variables but usually there are far fewer of those than there are commodities). It is essentially folklore that linear programs with many VUBs can be quite difficult (sometimes attributed to degeneracy).

These inequalities can also be difficult for the exponential potential approach to handle. Normally, the indices $k \in K$ give rise to "natural" blocks in the formulation. Thus, constraint (4.30) would be one of those that we view as a linking constraint. In addition, we would use another block to handle all variables x_j. As a consequence, when performing a block iteration, the step direction may substantially violate constraint (4.30). In other words, we may experience the *width* difficulty analyzed in Chapter 2. The theoretically correct remedy we described there (i.e., bounding the variables), while helpful, certainly falls short

in practical applications (e.g. network design models with thousands of commodities). A little reflection shows that the problem is caused by our separating the f^k variables from the x variables. This suggests modifying our formulation so that the x variables not only appear in a separate block but also appear in *every* block $k \in K$. In addition, for each $k \in K$ the constraint

$$f_j^k - x_j \leq u_j, \tag{4.31}$$

is added to those already in block k. After this transformation we no longer have a "block-angular" formulation; nevertheless each block iteration is still valid in the context of the original formulation.

The intuitive advantage of this transformation, in the case where $|K|$ is large is clear: should we use an approach such as the round-robin block iteration method on the *original* formulation, we only change the variables x very infrequently. Thus the infeasibilities of constraints (4.30) may improve very slowly, leading to small step sizes and overall slow convergence (and, on large examples, numerical stalling of the algorithm). By using inequalities (4.31) we frequently refresh the x. Yet another advantage of this approach is that in e.g. network design models it is the x_j that appear significantly in the objective vector c of **LP** (and usually the f^k do not or are not as important). Thus the additional urgency to frequently update the x_j. There is a price, of course, to using (4.31) – the model becomes larger (but by at most a factor of two, in terms of nonzeros). Moreover, the block LPs potentially become more difficult – but this difficulty does not seem to be borne out in our testing, however. We will return to these issues in the section on numerical experience.

Finally, we point out that it is quite easy to detect VUBs (4.30) and to automatically reformulate a problem to include inequalities (4.31).

5. Computational tests

In this section we present numerical results with several classes of problems. In all of our experiments, a problem was input as a file using the LP format, with the rows of each block appearing consecutively and delimited by the keyword BLOCK. All linear programs, regardless of any implicit combinatorial nature, were solved as general linear programs using CPLEX [ICP] (various releases). Block LPs were solved using the dual simplex method. As stated above, the LPs encountered in the *NX* procedure were solved either using the barrier method or the dual simplex method. In our implementation, we set $\epsilon = 1.0e^{-03}$ although this precision was not always achieved (but was in many cases exceeded).

The numerical experiments listed below were carried out either on a 660 MHz ev5 Alpha computer, with 1GB of memory and 8MB of cache,

or on a 754 MHz ev6 Alpha computer, with 2GB of memory and 8MB of cache.

5.1 Network Design Models

Network design problems are mixed-integer programs arising in several practical contexts, in particular, telecommunications. Frequently, these are quite difficult mixed-integer programs (see [BCGT96]). The cut-and-branch and branch-and-cut approaches have been proven the most effective in handling these problems by several researchers. On the other hand, these approaches lead to some very difficult continuous relaxations in the case of realistic large networks (e.g. 300 nodes).

We have already provided a description of these problems in Section 2.2.1, including the polyhedral strengthening of the basic formulation (1.53 - 1.56) via the so-called cutset-inequalities (1.57). Some of the examples we consider below include "survivability" modeling features, and here we have some inequalities of the form

$$\sum_{(i,j)\in C^1} x_{ij} + \sum_{(i,j)\in C^2} \frac{r+1}{r} x_{ij} \geq k \qquad (4.32)$$

where $r > 0$ is an integer (see [BM00], [MW97]).
All these examples involve real data (demands and costs) from the New York City LATA. On these problems, we observed that CPLEX's barrier code was far superior to the dual simplex method, whenever the problem was large enough, as was the case for all the instances listed below.

Inst	m	n	nonz	Barr	Barr 1% sec	ϵ-app 1% sec	ϵ-app 0.1% sec
netd1	31317	270760	793143	1097	681	17	22 ($9.6e^{-4}$)
netd2	31363	270806	799783	1563	1383	81	165 ($6.0e^{-6}$)
netd3	31746	270866	837132	3697	628	21	30 ($2.0e^{-4}$)
netd4	32496	427085	1256593	16775	10000	34	40 ($6.5e^{-4}$)
netd5	32866	427180	1284466	17870	11047	387	1802 ($5.0e^{-4}$)
netd6	32496	427085	1256593	25231	10336	1836	3000 ($3.9e^{-4}$)
netd7	33003	427215	1316879	44688	12305	47	56 ($5.6e^{-4}$)
netd8	33218	427215	1329112	51140	11976	59	68 ($5.6e^{-4}$)
netd9	33966	588618	1777359	89577	73166	343	494 ($1.0e^{-3}$)

Table 4.1. Network design problems

In Table 4.1, "n", "m" and "nonz" are, respectively, the number of columns, constraints and nonzeros after the linear program has been

preprocessed by CPLEX. "Barr sec" is the time that the barrier code required to solve the problem to default tolerances, in seconds, **excluding** the time required to compute the initial Cholesky factorization. "Barr 1% sec" is the time, in seconds, that the barrier code ran until the half-sum of the "primal value" and "dual value" (as listed in CPLEX's output) was within 1% of the eventual optimum. "ϵ-app 1%" is the time it took our code to prove a lower bound $\underline{B}$ and an upper bound $\bar{B}$ such that $\bar{B} \leq 1.01\underline{B}$. "$\epsilon$-app 0.1%" indicates the time required to get a 0.1% error bound, with the quantity in parenthesis showing the actual relative error proved at that point.

As stated above, the dual method (incl. using steepest edge pivoting) was inferior to the barrier code, particularly in the more difficult instances. For example, on instance **netd9**, after several days, dual simplex had computed a bound that was more than 20% away from the optimum. Also note that the time required by the barrier code to compute the initial ordering of columns is nontrivial; on **netd9** this was of the order of 800 seconds.

5.2 Minimum-cost multicommodity flow problems

There is an abundant bibliography concerning algorithms for multicommodity flow problems; see [GOPS98]. These problems arise in the telecommunications industry, for example. Nevertheless, it is difficult to obtain "real" large-scale instances, and researchers in this area have traditionally relied on random problem generators.

In [GOPS98] the authors describe computational results with a code for multicommodity-flow problems which is based on the exponential potential reduction method. To test their code, they used the generators "RMFGEN", "MULTIGRID" and "TRIPARTITE". The first two are multicommodity variations of well-known single-commodity network flow problems. The third one [GOPS98] generates problems that are difficult for the exponential potential reduction method. In this section, we test our code on instances produced by these three generators provided to us by the authors of [GOPS98] and on instances we obtained using these generators. It is important to note that the behavior of each generator is consistent in terms of the difficulty of the instances it produces; thus, we report on a sample of the problems only.

As noted before, block iteration reduces to a minimum-cost flow problem. In [GOPS98] these were handled using a network-simplex code, warm started. However, we stress that we solve these problems as general linear programs. In spite of this fact, we obtain nontrivial speed-ups over the code in [GOPS98]. We attribute this fact to the decreased it-

eration count, which itself is due to the better lower bounds and to the *NX* procedure.

The next two tables describe results with TRIPARTITE instances, the last two of which we generated and were not run using the code in [GOPS98]. In this table, the column headed " ϵ-approx its" shows the number of block iterations solved in this run. For completeness, in table 4.3 we show statistics for the first two runs using a recent version of the code in [GOPS98], run on a SUN Ultra 2 (approximately twice as slow as our machine). We note that on TRIPARTITE problems the Barrier method appears to be much more effective than the dual simplex; however, it does require significantly more memory. The 1%-approximation CPLEX running times using the barrier method follow the same convention as in Table 4.1; in the case the dual simplex method we report the run time when the method was used with a 1% optimality tolerance.

Inst	m	n	nonz	Barr. sec	Barr. 1% sec	ϵ-app 1% sec	ϵ-app 1% its
tp1	3838	25632	75232	29	17	18	941
tp2	18741	127007	368526	237	176	509	2831
tp3	37541	312034	917288	1126	875	1287	3272
tp4	54769	783817	2322024	11624	9687	2370	5191

Table 4.2. TRIPARTITE instances.

Inst	GOPS 1% sec	GOPS 1% its
tp1	338	66072
tp2	6772	180368

Table 4.3. Sample comparison with [GOPS98].

The next table describes two instances created by RMFGEN, and an instance created by MULTIGRID. The first problem was run using the Barrier method; the other two using the steepest-edge Dual simplex method because of memory limitations.

Minimum-cost multicommodity flow problems have attracted a great deal of attention, generating a rich literature on this subject that is simply too abundant to properly reference here. In addition, there is an abundance of special-purpose codes for these problems. Many of

these codes are essentially refined implementations of traditional linear-programming algorithms (e.g. simplex or interior point) where the explicit knowledge of the matrix structure is used to speed up or short-cut iterations. This can result in sometimes substantial improvements over commercial LP solvers. Nevertheless (and not surprisingly) we have observed that several of these codes also scale with problem size at the same rate as commercial LP solvers. As a result, on large problem instances our implementation is considerably faster (at least as an approximation algorithm).

Another category is that of Lagrangian-based methods. In principle, such methods might provide more competition (after all, our implementation can be viewed as a Lagrangian method). There are very many implementations of Lagrangian relaxation; some comparisons have shown our method to be much more effective; the lack of an adequate "library" of problems makes a scientific comparison difficult.

5.3 Maximum concurrent flow problems

The maximum concurrent flow problem, described in Chapters 1 and 2 provided the original motivation for the development of exponential function methods. As such, we would expect our algorithm to perform well on these problems. In fact, the exponential potential method, as used in the references we have provided, tackles the problem in implicit form: rather than using (say) formulation **LP0** (1.5 - 1.7) it simply keeps track of edge penalties, which are used to reroute commodities, resulting in an update of the edge penalties, and so on.

Our implementation cannot operate this way because it only receives inputs as linear programs. For these experiments, we use the "minimum congestion" formulation:

Inst	m	n	nonz	CPLEX sec	CPLEX 1% sec	ϵ-app 1% sec	ϵ-app 1% its
rmfgen1	27352	116403	257443	187	133	115	3292
rmfgen2	156891	670052	2009281	9940	4868	1252	31840
mg1	271039	818943	2414742	> 50000	1200	230	1923

Table 4.4. RMFGEN and MULTIGRID instances

$$\begin{aligned} &\min \quad \lambda \\ &\text{s.t.} \\ &\sum_k f_{ij}^k - \lambda u_{ij} \leq 0 \quad \forall\,(i,j) && (4.33)\\ &N^k f^k = b^k, \quad k = 1,\cdots,K && (4.34)\\ &0 \leq f^k \qquad\qquad k = 1,\cdots,K, && (4.35) \end{aligned}$$

Using the techniques of Section 4, for each edge (i,j) inequality 4.33 gives rise to the inequality

$$f_{ij}^k - \lambda u_{ij} \leq 0, \tag{4.36}$$

for each commodity k.

Our implementation appears to handle these problems exceptionally well. In fact, we can solve many of these instances with as much as six digits of accuracy. The following table displays running times *RMFGEN* instances.

Inst	m	n	nonz	Dual sec	ϵ-approx sec	ϵ-approx proved error
RMF20	40620	163181	489540	326	92.17	10^{-7}
RMF22	59323	241242	723723	809	135.0	10^{-7}
RMF15	90980	382721	1148160	3332	94.31	10^{-7}
RMF21	132968	565445	1696332	7058	286.5	10^{-7}
RMF13	160143	689056	2067177	9434	399.4	10^{-6}
RMF23	234446	1019054	3057169	40368	708.3	10^{-7}
RMF19	251201	1095606	3286819	66112	510.4	10^{-7}
RMF25	356669	1569446	4708372	137865	1227.3	10^{-6}
RMF26	481236	2131038	6393176	455541	1781.0	10^{-7}

Table 4.5. Maximum concurrent flow problems

Figure 4.1 displays the running time of our algorithm as a function of the number of columns for the instances in Table 4.5.

5.4 More sophisticated network design models

We conclude this section by describing our experience with some very comprehensive network design instances that were provided to us by AT&T Laboratories. Even though we cannot make this data publicly available, we can however describe the underlying model in some detail. As we will see, the pure routing component of this model is only a very small fraction of the overall picture; the success of our implementation,

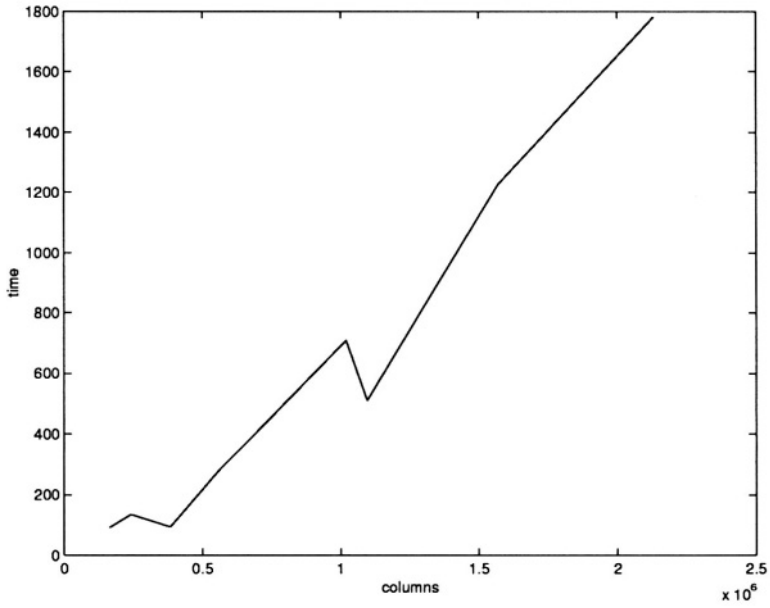

Figure 4.1. Time as a function of column count for potential function method on *RMFGEN* concurrent flow problems

in our opinion, underscores the potential usefulness of the exponential potential approach to complex, large, difficult, real-life models.

The AT&T models we considered involve installing capacities in the edges of a network so that multicommodity demands can be routed. In addition,

- There are several categories of demands, which are hierarchically organized, as are the capacity types. Somewhat simplifying the model, demand of type 1 can be routed on capacities of types $1,2,3,\ldots$; demand of type 2 can be routed on capacities of types 2,3, …, and so on.
- We can "lose" demand, at a cost.
- Any given commodity can only be routed on certain edges, which are part of the input.
- Capacities are not bought for one edge at a time; rather, we buy capacity for a "system" that comprises a subset of edges (e.g. a tree, a path, a cycle). From a modeling standpoint, this gives rise to variable upper-bound inequalities where the same capacity variable appears in several inequalities.

- The capacities themselves have to be packed into a higher category of devices, which must also be bought, and which also involve subsets of edges of the underlying network (but different subsets than those used for the capacities themselves).
- There are constraints specifying minimum ratios between capacity amounts of the different hierarchies installed on any given edge of the network.

Overall, this gives rise to a very complex formulation where the pure routing component is quite nonstandard, and where the constraints involving capacity variables only form a nontrivial subset of the model. In addition, the formulation contains many coefficients with very large or very small value.

We were given two problem instances, both corresponding to a network with roughly 500 nodes.

1 The first instance, *SONET4*, has 475162 columns, 117477 rows, and 1884499 nonzeros. This problem was solved by CPLEX (Barrier) in 1640 seconds, and has value 1.621107. Table 4.6 shows the behavior of our implementation. Almost immediately, our code had a feasible solution that was correct to four digits and the bulk of the effort was to find a matching lower bound; after less than five minutes the lower bound was within 1.12% of the upper bound.

2 The second instance, *SONET5*, has 2984075 columns, 337678 rows, and 11903433 nonzeros. With the variable upper-bound strengthening we employ, the number of rows that our implementation handles swells to over 900000. This instance required five hours using CPLEX's Parallel Barrier method, with four CPUs employed, and consuming over 4GB of memory. In contrast, the first call to *NX* by our method yielded a solution with proved relative error 1.32×10^{-4} (at 130 seconds of running time). See Table 4.7.

Time	LB	UB
31.5	1.591	1.621
277.3	1.603	1.621
1199	1.6206	1.62114
1283	1.62109	1.62111

Table 4.6. Behavior of approximation algorithm on $SONET4$

Parallel Barrier Time	ϵ-approx LB at 130 sec.	ϵ-approx UB at 130 sec.
5 hours	1.59567	1.59588

Table 4.7. Behavior of approximation algorithm on *SONET*5

5.5 Empirical trade-off between time and accuracy

As described in Chapter 2, the running time of the exponential potential method might grow proportional to ϵ^{-2}. Is this worst-case behavior observed in practice? Our personal bias is that a more relevant issue is how the algorithm scales with *problem size* while maintaining ϵ fixed at a realistically practical value (in the telecommunications applications, $\epsilon = 10^{-2}$ would be ample). Nevertheless, in this section we present some empirical analyses of the behavior of our implementation as ϵ becomes small. To some degree, these experiments are imperfect, and we cannot make them better – our implementation will in general be unable to attain an arbitrarily small ϵ, due to the difficult numerical behavior of the exponential function (and not because of a fundamental underlying weakness of the overall algorithmic approach).

In Table 4.8 we study instance **rmfgen2** (from Table 4.4). Here "lower bound", "upper bound" and "relative error", are, respectively, the tightest lower bound proved so far, the best upper bound obtained by the algorithm, and the relative gap between the two. Recall that our implementation proves upper bounds through finding feasible solutions, so the only source of error in this table concerns objective value. Figure 4.2 displays time as a function of ϵ^{-1} for this problem instance.

In Table 4.9 and Figure 4.3 we study instance **netd9** (from Table 4.1). Figures 4.2 and 4.3 appear to show a *sub*linear dependence of running time on accuracy. This observation was also made in [GOPS98]. Clearly, further theoretical work is needed to understand this behavior.

It is worth noting the relatively vertical section of the plot for **rmfgen2**. Comparing with the entries in Table 4.8 we see that at this point the algorithm had an upper bound that was close to the optimum and was slowly improving, while the lower bound was essentially fixed, and that this stage of the algorithm accounts for a large fraction of the overall running time. These facts underscore our observation that lower bound generation is the weakest ingredient of the exponential potential approach.

Time	lower bound	upper bound	relative error
433	2.858729e+05	3.958275e+05	0.384627574
603	3.049973e+05	3.647225e+05	0.195822061
697	3.146053e+05	3.510820e+05	0.115944328
784	3.207860e+05	3.469756e+05	0.081641967
874	3.309232e+05	3.425384e+05	0.035099383
958	3.309232e+05	3.415940e+05	0.032245548
1051	3.356333e+05	3.408308e+05	0.015485651
1142	3.356333e+05	3.408308e+05	0.015485651
1260	3.378108e+05	3.406015e+05	0.008261133
1359	3.392114e+05	3.405609e+05	0.003978345
1461	3.392114e+05	3.405378e+05	0.003910246
1555	3.392114e+05	3.405357e+05	0.003904055
1651	3.392114e+05	3.405299e+05	0.003886957
1743	3.392114e+05	3.405207e+05	0.003859835
1835	3.392114e+05	3.405194e+05	0.003856002
1931	3.392114e+05	3.405191e+05	0.003855118
2037	3.392114e+05	3.405191e+05	0.003855118
2140	3.392114e+05	3.405172e+05	0.003849517
2255	3.392114e+05	3.405172e+05	0.003849517
2371	3.396824e+05	3.405172e+05	0.002457590
2506	3.398923e+05	3.405155e+05	0.001833522
2647	3.403687e+05	3.405140e+05	0.000426890

Table 4.8. Behavior of approximation algorithm on instance **rmfgen2**

5.6 Hitting the sweet spot

The *NX* routine in our implementation clearly plays a critical role. An important empirical issue concerns the relative size of the support of the optimal solution in practical instances, and how this affects our implementation.

In Table 4.10 we show, for a variety of instances, the overall number of columns, the number of columns in the restricted linear program solved in the last call to *NX*, the sum of running times required to solve *all* restricted LPs ("restricted LP cum. sec."), the total number of calls to *NX*, ("*NX* count") and the running time needed by CPLEX to solve the original LP ("full sec."). Since the original LP and the restricted LPs are in some cases solved using different methods (e.g. CPLEX's barrier may be used for the *NX* LP, and CPLEX's dual for the overall LP) the comparison between running times is strictly for qualitative purposes.

As we can see from this table, in general the calls to *NX* yield LPs that are solved quickly. Instances *netd*9 and *SONET*5 show standard behavior: the restricted linear programs arising here are fairly small,

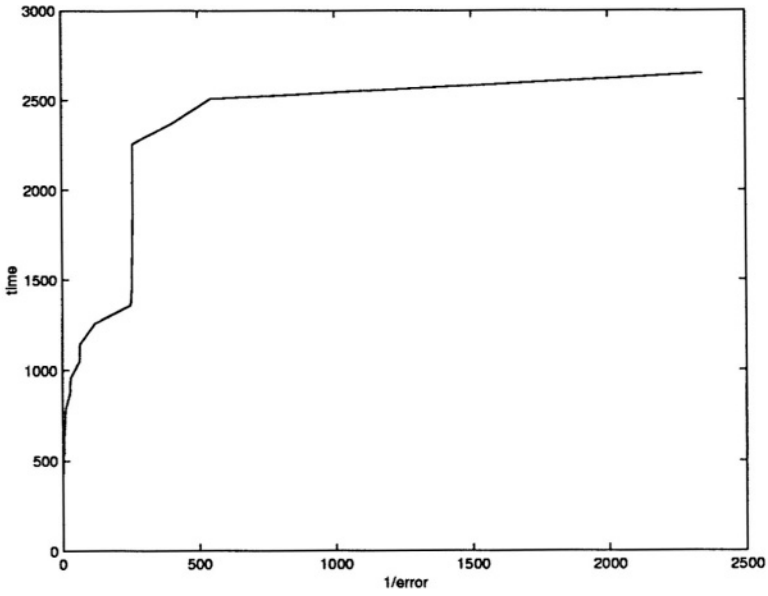

Figure 4.2. Time as function of ϵ^{-1} for instance **rmfgen2**

Time	lower bound	upper bound	relative error
59	2.52791e+05	2.75242e+05	0.088810379
79	2.52791e+05	2.71684e+05	0.074737511
99	2.52791e+05	2.70827e+05	0.071349342
130	2.65215e+05	2.69837e+05	0.017427741
157	2.65429e+05	2.69739e+05	0.016237840
183	2.65550e+05	2.69687e+05	0.015575545
211	2.65550e+05	2.69663e+05	0.015486673
242	2.66441e+05	2.69634e+05	0.011985750
276	2.66522e+05	2.69611e+05	0.011590033
310	2.66687e+05	2.69560e+05	0.010772929
345	2.66856e+05	2.69500e+05	0.009910581
374	2.66856e+05	2.69499e+05	0.009905710
405	2.67302e+05	2.68850e+05	0.005791196
436	2.68144e+05	2.68703e+05	0.002081337
477	2.68144e+05	2.68626e+05	0.001796417
482	2.68337e+05	2.68626e+05	0.001077375
488	2.68357e+05	2.68626e+05	0.001001648
493	2.68358e+05	2.68626e+05	0.000999783

Table 4.9. Behavior of approximation algorithm on instance **netd9**

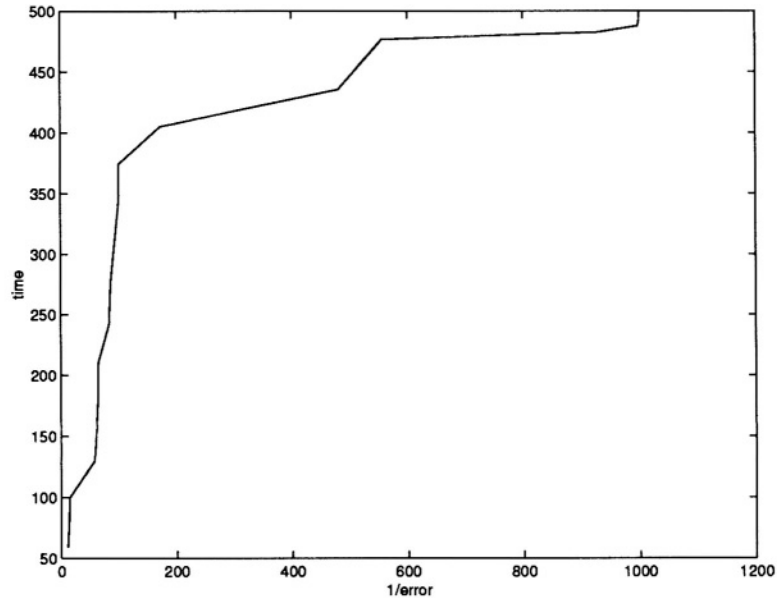

Figure 4.3. Time as function of ϵ^{-1} for instance **netd9**

Instance	Columns in full LP	Columns in last restricted LP	restricted LP cum. sec.	*NX* count	full sec.
netd9	588518	5989	75	17	89577
SONET5	2984075	21839	12	2	5 hours
RMF26	2131038	711793	1562	2	455541

Table 4.10. Speedup of restricted LP in *NX*

and their solution times are insignificant compared to that of the full problem. But consider *RMF*26. Here the last call to *NX* produced a restricted linear program with about one-third of all the variables. At over seven hundred thousand variables, this linear program is not small. More important, even though it comprises a large fraction of the overall problem, it is solved nearly 500 times faster. Thus, the near-optimal nature of the restricted set of variables acts to reduce the complexity of the problem far beyond what could be expected by the relative decrease in problem size.

As an additional point regarding *RMF*26, notice (Table 4.5) that the calls to *NX* dominate in this instance the overall running time of our

implementation – the "magic" of the exponential potential function is that it quickly singles out the "right" one-third of the variables.

6. Future work

The *NX* routine plays a critical role in our implementation. Thus, it would be of great interest to develop a better theoretical understanding for its success. A first step would be to describe how fast the exponential potential method converges to a near-optimal *support*.

In addition, the issue of reducing the number of iterations (and the related issue of proving stronger lower bounds) remains quite critical, in particular when considering more general classes of linear programs. One possible venue of research concerns the full integration of a *NX*-like step into the algorithm proper. By this we mean that, every so often, we would carry out a more expensive step which is on the other hand more accurate than the generic algorithm, and which serves to accelerate potential reduction. *NX* is an example of this strategy; but perhaps the approach of fully solving the "restricted" LP is excessive. We have experimented with partially solving the restricted LP, with mixed results. Part of the difficulty here is that, in general, a black-box call to a generic LP solver may be a poor way to approximately solve a linear program (the convergence tends to be rather abrupt). An intriguing approach would be that of interleaving potential reduction steps with pivoting-like steps.

Another crucial future area of work concerns the generalization of the approach to broader classes of linear programs, in particular, linear programs without block structure. We could view such problems as consisting of a single block – the theory applies, of course, but also predicts a small step size, which we have observed in practice, with the result that the algorithm jams prematurely. A somewhat more promising approach is that of viewing each variable as constituting a separate block.

Finally, an important direction of research concerns extremely large problems, large enough that each individual block iteration is fairly expensive. Here we need a better understanding of the effect of coarsely solving block LPs on convergence rate.

APPENDIX - FREQUENTLY ASKED QUESTIONS

Q: When I need to solve a linear program, I simply run my favorite LP code. It is fast, robust, and mature. Why do I need to bother with approximation algorithms?

A: Clearly an enormous amount of progress has been achieved in the field of traditional linear programming. What motivates the study of approximation schemes is that very large problems arising in practice overstrain the capabilities of even the best LP codes. And we can expect larger problems to continue to arise. Thus, approximation algorithms simply are a venue for delivering some useful information on extremely difficult problems, in practicable time.

Q: But why the emphasis on provably good algorithms? Are you simply just trying to write more papers?

A: The central issue here is that, apparently, "seat of the pants" data might suggest that there is not enough correlation between the provably good performance of a linear programming algorithm and its actual behavior in practice. The classical pieces of evidence in favor of this view are the poor convergence of the Ellipsoid Method, and the very fast convergence of the Simplex Method. Thus, in our insistence on provably good bounds, are we engaging in gratuitous theoretical research?

Clearly, this is a complex issue. It reflects a schism between two competing communities: the theoretical computer science community, and the traditional mathematical programming community. We view the theoretical component of the research on approximate algorithms for linear programming as an integral part of a program that includes experimental testing as a critical component. Thus, theory not only guides implementation but is also informed by it. Over time, this may result in far more complex, theoretically faster algorithms which also do much better in practice. In order for this to happen, the algorithms community will probably need to expand its chest of tools so as to incorporate methodology of nonlinear programming, and will need to become more fully involved in experimental evaluation.

At the same time, as problem sizes become massive, experimentation with and tuning of algorithms will become much more challenging. It will be difficult to argue that a particular methodology is "good' solely on the basis of experimental data that only samples a tiny fraction of

the problem spectrum. Further, applications may well place a premium on predictability of codes. A "provably good" label with the appropriate focus may become a *requirement* in practice.

Q: You speak of "provably good" algorithms, but a convergence rate of ϵ^{-2} is laughable.

A: Newton's method – embodied in the Interior Point methods for Linear Programming – provably achieves a convergence rate of $\ln(\frac{1}{\epsilon})$, and its practical convergence, in particular on very difficult linear programming instances, can be nothing short of wondrous. But this comes at a steep computational linear algebra price. On a very large-scale model each iteration of Newton's method frequently becomes too slow to be practical. In addition, the method can require an unrealistic amount of storage. An important research objective in the nonlinear programming community has always been that of developing algorithms that exhibit Newton-like convergence behavior, while in also producing fast iterations and requiring modest amounts of storage. This is a goal that should also be considered by the algorithms community (see above response). The challenge is that large problems are large enough that we can only afford sophisticated linear algebra computations on submatrices that are tiny compared to the overall model.

Over the last few decades, computational science has periodically been punctuated by visible advances in computational machinery. This made possible the leveraging of increasingly complex mathematical methodology on ever larger problems. But at any one point in time, the largest applications at that time were beyond the capabilities of the current machinery. Such is the case right now (see e.g. Table 0.1). Of course, a technological revolution could soon come about that would provide a quantum improvement on our computing capabilities. In order for such a revolution to be significant in the context of massive linear programs, it would have to be truly stupendous: in terms of today's e.g. large-scale routing problems, one would need to be able to carry out Cholesky factorization of dense matrices with hundreds of millions of rows and columns. And if we have to wait, say, five to ten years for such an improvement in computing, then the size of "large" applications will probably have increased as well. Think about future applications of optimization to biology, cryptography, next-generation networking and (above all) the design, manufacture and control of the computational devices that would deliver this computing revolution.

Hopefully, the optimization community will have more than heuristic methods to offer in the solution of such future gargantuan problems. It is also possible that the optimization community will not participate in the solution of such problems, or that the problems may not even be viewed as having an optimization component, but we hope that this is an unlikely event.

Q: But I am a practical person. What *do* I get from these approximation algorithms?

A: See e.g. Tables 4.1, 4.5, 4.6 and 4.7. They show the behavior of one piece of code, run unchanged and with no runtime options of three vastly different problem classes. Certainly, other nontraditional methods may achieve experimental success. Our admittedly preliminary testing shows that many such methods scale poorly with problem size. Of course, "the proof is in the pudding" and the next decade promises to be quite exciting in this field.

References

[A00] C. Albrecht, Provably good global routing by a new approximation algorithm for multicommodity flow. In *Proceedings of the International Conference on Physical Design (ISPD)*, San Diego, CA, (2000), 19 – 25.

[AMW98] D. Alevras, M. Grötschel and R Wessäly, Cost-efficient network synthesis from leased lines, *Annals of Operations Research* **76** (1998), 1 – 20.

[Ba96] F. Barahona, Network design using cut inequalities, *SIAM J. Opt.* **6** (1996), 823 – 837.

[BA00] F. Barahona and R. Anbil, The Volume Algorithm: producing primal solutions with a subgradient method, *Mathematical Programming* **87** (200), 385 – 399.

[B96] D. Bienstock, Experiments with a network design algorithm using ϵ-approximate linear programs (1996).

[BCGT96] D. Bienstock, S. Chopra, O. Günlük and C. Tsai (1996), Minimum Cost Capacity Installation for Multicommodity Network Flows, *Math. Programming* **81** (1998), 177 – 199.

[BM00] D. Bienstock, G. Muratore, Strong inequalities for capacitated survivable network design problems, *Math. Programming* **89** (2000), 127 – 147.

[BR00] D. Bienstock, O. Raskina, Asymptotic analysis of the flow deviation method for the maximum concurrent flow problem (2000), to appear, *Math. Programming*.

[B00] R. Bixby, personal communication.

[C79] V. Chvatal, A greedy heuristic for the set-covering problem, *Math. of Operations Research* **4** (1979), 233 – 235.

[CM94] R. Cominetti and J.-P. Dussault, A stable exponential penalty algorithm with superlinear convergence, *J. Optimization Theory and Applications* **83**:2 (1994).

[CM94] R. Cominetti and J. San Martín, Asymptotic analysis of the exponential penalty trajectory in linear programming, *Mathematical Programming* **67** (1994), 169 – 187.

[Cou43] R. Courant, Variational methods for the solution of problems of equilibrium and vibration, *Bull. Amer. Math. Soc.* **49** (1943), 1 – 43 .

[DS69] S.C. Dafermos and F.T. Sparrow, The traffic assignment problem for a general network, *Journal of Research of the National Bureau of Standards - B*, **73B** (1969).

[D00] R. Daniel, personal communication.

[DW] G.B. Danzig and P. Wolfe, The decomposition algorithm for linear programming, *Econometrica* **29** (1961), 767 – 778.

[FM68] A.V. Fiacco and G.P. McCormick, *Nonlinear Programming: Sequential Unconstrained Optimization Techniques*, Wiley, New York (1968).

[F] W. Feller, *An introduction to probability theory and its applications*, Wiley, New York (1966).

[FG99] A. Frangioni and G. Gallo, A Bundle Type Dual-Ascent Approach to Linear Multicommodity Min Cost Flow Problems, *INFORMS JOC* **11** (1999) 370 – 393.

[GGSV96] J.-L. Goffin, J. Gondzio, R. Sarkissian and J.-P. Vial, Solving nonlinear multicommodity flow problems by the analytic center cutting plane method, *Mathematical Programming* **76** (1996) 131 – 154.

[ICP] ILOG CPLEX, Incline Village, NV.

[F00] L.K. Fleischer, Aproximating Fractional Multicommodity Flow Independent of the Number of Commodities, *SIAM J. Disc. Math.*, **13** (2000), 505 – 520.

[FW56] M. Frank and P. Wolfe, An algorithm for quadratic programming, *Naval Res. Logistics Quarterly* **3** (1956), 149 – 154.

[GK98] N. Garg and J. Könemann, Faster and simpler algorithms for multicommodity flow and other fractional packing problems, *Proc. 39th Ann. Symp. on Foundations of Comp. Sci.* (1998) 300-309.

[FGK71] L. Fratta, M. Gerla and L. Kleinrock, The flow deviation method: an approach to store-and-forward communication network design, *Networks* **3** (1971), 97 – 133.

[GOPS98] A. Goldberg, J. Oldham, S. Plotkin and C. Stein, An Implementation of a Combinatorial Approximation Algorithm for Minimum Multicommodity Flow, IPCO 1988, R.E. Bixby, E.A. Boyd, R.Z. Rios-Mercado, eds., *Lecture Notes in Computer Science* **1412**, Springer, Berlin, 338 – 352.

[GK94] M.D. Grigoriadis and L.G. Khachiyan (1991), Fast approximation schemes for convex programs with many blocks and couping constraints, *SIAM Journal on Optimization* **4** (1994) 86 – 107.

[GK95] M.D. Grigoriadis and L.G. Khachiyan, An exponential-function reduction method for block-angular convex programs, *Networks* **26** (1995) 59-68.

[GK96] M.D. Grigoriadis and L.G. Khachiyan, Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}KNM)$ time, *Mathematical Programming* **75** (1996), 477 — 482.

[GK96] M.D. Grigoriadis and L.G. Khachiyan, Coordination complexity of parallel price-directive decomposition, *Math. Oper. Res.* **21** (1996) 321 – 340.

[GV] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore and London (1996).

[Ka02] , G. Karakostas, Faster Approximation Schemes for Fractional Multicommodity Flow Problems, *Proc. 13th Ann. Symp. on Discrete Algorithms* (2002).

[KPST90] 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, *Proc. 22nd Ann. ACM Symp. on Theory of Computing* (1990), 310 – 321.

[KY98] P. Klein and N. Young, On the number of iterations for Dantzig-Wolfe optimization and packing-covering approximation algorithms, Proceedings IPCO 1999, 320 – 327.

[LMPSTT91] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos and S. Tragoudas, Fast approximation algorithms for multicommodity flow problems, *Proc. 23nd Ann. ACM Symp. on Theory of Computing* (1991), 101-111.

[LR98] T. Leighton and S. Rao, An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms, *Proc. FOCS 29* (1988), 422 – 431.

[LSS93] T. Leong, P. Shor and C. Stein, Implementation of a Combinatorial Multicommodity Flow Algorithm, DIMACS Series in Discrete mathematics and Theoretical Computer Science **12** (1993), 387-405.

[LNN95] C. Lemarechal, A. Nemirovskii and Y. Nesterov, New variants of bundle methods, *Math. Programming* **69** (1995), 111 – 148.

[LLR94] N. Linial, E. London and Y. Rabinovich, The geometry of graphs and some of its algorithmic applications, *Proc. FOCS 35* (1994), 577 – 591.

[L75] L. Lovász, On the ratio of optimal integral and fractional covers, *Discrete Mathematics* **13** (1975), 383 – 390.

[LN93] M. Luby and N. Nisan, A parallel approximation algorithm for positive linear programming, *Proc. 24th Ann. ACM Symp. on Theory of Computing* (1993), 448 – 457.

[Lu] D. Luenberger, *Introduction to Linear and Nonlinear Programming*, Addison-Wesley, Menlo Park (1973).

[MMV91] T. Magnanti, P. Mirchandani and R. Vachani, Modeling and solving the capacitated network loading problem, Working Paper OR 239-91, MIT (1991).

[MW97] T. Magnanti and Y. Wang, Polyhedral Properties of the Network Restoration Problem-with the Convex Hull of a Special Case (1997), to appear, *Math. Programming.*

[OK71] K. Onaga and O.. Kokusho, On feasibility conditions of multicommodity flows in networks, *IEEE Transactions on Circuit Theory*, **18** (1971), 425 – 429.

[OR88] J.B. Orlin, A faster strongly polynomial minimum cost flow algorithm, *Operations Research* **41** (1993), 338 – 350.

[PK95] S. Plotkin and D. Karger, Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows, In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, (1995), 18-25.

[PST91] S. Plotkin, D.B. Shmoys and E. Tardos, Fast approximation algorithms for fractional packing and covering problems, *Math. of Oper. Res.* **20** (1995) 495-504. Extended abstract: *Proc. 32nd Annual IEEE Symp. On Foundations of Computer Science*, (1991), 495-504.

[R95] T. Radzik, Fast deterministic approximation for the multicommodity flow problem, *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms* (1995).

[R01] O. Raskina, Ph. D. Dissertation, Dept. of IEOR, Columbia University (2001).

[S91] R. Schneur, Scaling algorithms for multicommodity flow problems and network flow problems with side constraints, Ph.D. Thesis, MIT (1991).

[SC86] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley (1986).

[SM91] F. Shahrokhi and D.W. Matula, The maximum concurrent flow problem, *Journal of the ACM* **37** (1991), 318 – 334.

[STODA94] M. Stoer and G. Dahl, A polyhedral approach to multicommodity survivable network design, *Numerische Mathematik* **68** (1994), 149 – 167.

[V90] P.M. Vaidya, An algorithm for linear programming which requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ arithmetic operations, *Math. Programming* **47**, 175-201.

[W82] L. Wolsey, An analysis of the greedy algorithm for the submodular set covering problem, *Combinatorica* **2** (1982), 385 – 393.

[W97] S. Wright, Primal-Dual interior point methods, Siam (1997).

[Y95] N. Young, Randomized rounding without solving the linear program, in *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms* (1995), 170-178.

[Y01] N. Young, Sequential and parallel algorithms for mixed packing and covering, to appear, *Proc. 42nd Annual IEEE Symp. On Foundations of Computer Science*(2001).

Index