

Chapter Seven

Lossless Compression Algorithms

7.1 Introduction

the emergence of multimedia technologies has made *digital libraries* a reality. Nowadays, libraries, museums, film studios, and governments are converting more and more data and archives into digital form. Some of the data (e.g., precious books and paintings) indeed need to be stored without any loss.

As a start, suppose we want to encode the call numbers of the 120 million or so items in the Library of Congress (a mere 20 million, if we consider just books). Why don't we just transmit each item as a 27-bit number, giving each item a unique binary code (since $2^{27} > 120,000,000$)?

The main problem is that this "great idea" requires too many bits. And in fact there exist many coding techniques that will effectively reduce the total number of bits needed to represent the above information. The process involved is generally referred to as *compression*

7.2 Basics of Information Theory

Information theory is defined to be the study of efficient coding and its consequences. It is the field of study concerned about the storage and transmission of data. It is concerned with source coding and channel coding.

Source coding: involves compression *Channel coding*: how to transmit data, how to overcome noise, etc Data compression may be viewed as a branch of information theory in which the primary objective is to minimize the amount of data to be transmitted

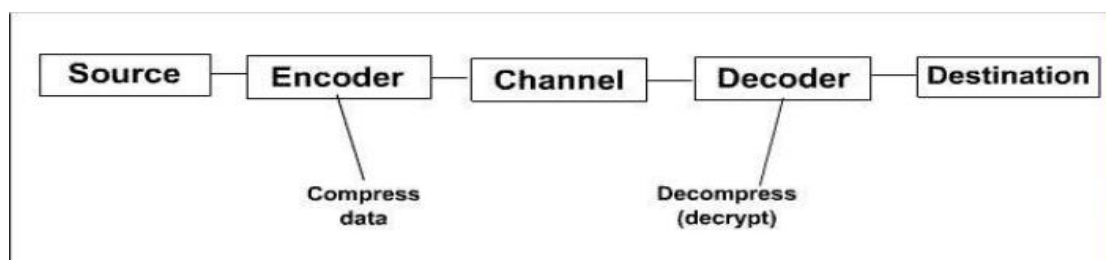


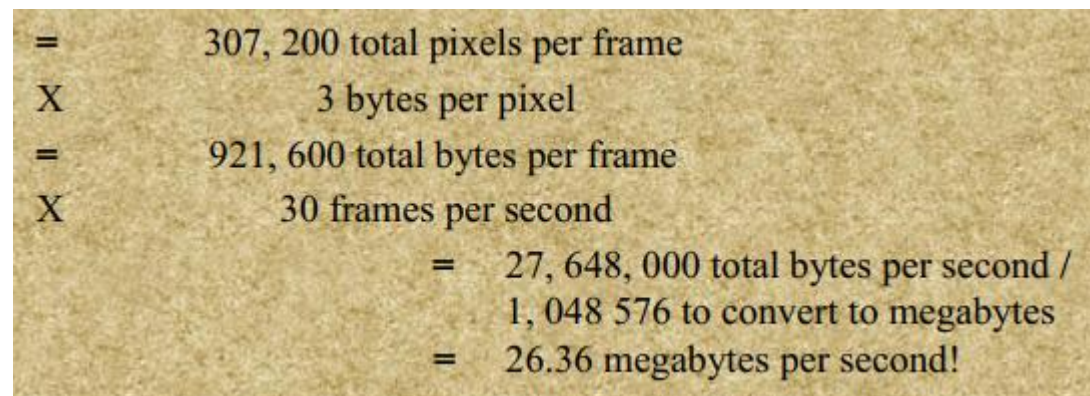
Fig Info,nvh++rmation coding and transmission

Need for Compression

With more colors, higher resolution, and faster frame rates, you produce better quality video, but you need more computer power and more storage space for your video. Doing some simple calculations (see below) it can be shown that with 24-bit color video, with 640 by 480 resolutions, at 30 fps, requires an astonishing 26 megabytes of data per second! Not only does this surpass the capabilities of the many home computer systems, but also overburdens existing storage systems.

640 horizontal resolution

X= 480 vertical resolution



= 307, 200 total pixels per frame
X 3 bytes per pixel
= 921, 600 total bytes per frame
X 30 frames per second
= 27, 648, 000 total bytes per second /
1, 048 576 to convert to megabytes
= 26.36 megabytes per second!

The calculation shows space required for video is excessive. For video, the way to reduce this amount of data down to a manageable level is to compromise on the quality of video to some extent. This is done by lossy compression which forgets some of the original data.

Compression Algorithms

Compression methods use mathematical algorithms to reduce (or compress) data by eliminating, grouping and/or averaging similar data found in the signal. Different Although there are various compression methods, including Motion JPEG, only MPEG-1 and MPEG-2 are internationally recognized standards for the compression of moving pictures (video).

A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (of bits, for example) which contains the same information but whose length is as small as possible.

Data compression has important application in the areas of data transmission and data storage. The proliferation of computer communication networks is resulting in massive transfer of data over communication links. Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Lossless compression is a method of reducing the size of computer files without losing any information. That means when you compress a file, it will take up less space, but when you decompress it, it will still have the exact same information. The idea is to get rid of any redundancy in the information, this is exactly what happens is used in ZIP and GIF files. This differs from lossy compression, such as in JPEG files, which loses some information that isn't very noticeable. Why use lossless compression?

You can use lossless compression whenever space is a concern, but the information must be the same. An example is when sending text files over a modem or the Internet. If the files are smaller, they will get there faster. However, they must be the same as that you sent at destination. Modem uses LZW compression automatically to speed up transfers.

There are several popular algorithms for lossless compression. There are also variations of most of them, and each has many implementations. Here is a list of the families, their variations, and the file types where they are implemented:

Family	Variations	Used in
Running-Length	none	
Huffman	Huffman AdaptiveHuffman Shannon-Fano	MNP5 COMPACT SQ
Arithmetic	none	
LZ78 (Lempel-Ziv 1978)	LZW(Lempel-ZivWelch)	GIF v.42bis compress
LZ77 (Lempel-Ziv 1977)	LZFG	ZIP ARJ
	LHA	

Table lossless coding algorithm families and variations

Variable Length Encoding

Claude Shannon and R.M. Fano created the first compression algorithm in the 1950's. This algorithm assigns variable number of bits to letters/symbols.

Shannon-Fano Coding

Let us assume the source alphabet $S=\{X_1,X_2,X_3,\dots,X_n\}$ and

Associated probability $P=\{P_1,P_2,P_3,\dots,P_n\}$

The steps to encode data using Shannon-Fano coding algorithm is as follows:

Order the source letter into a sequence according to the probability of occurrence in nonincreasing order i.e. decreasing order.

ShannonFano(sequence s)

If s has two letters

Attach 0 to the codeword of one letter and 1 to the codeword of another;

Else if s has more than two letter

Divide s into two subsequences S1, and S2 with the minimal difference between probabilities of each subsequence;

extend the codeword for each letter in S1 by attaching 0, and by attaching 1 to each codeword for letters in S2;

ShannonFano(S1);

ShannonFano(S2);

Example: Suppose the following source and with related probabilities

$S=\{A,B,C,D,E\}$

$P=\{0.35,0.17,0.17,0.16,0.15\}$

Message to be encoded="ABCDE"

The probability is already arranged in non-increasing order. First we divide the message

into AB and CDE. Why? This gives the smallest difference between the total probabilities of the two groups.

$S_1=\{A,B\}$ $P=\{0.35,0.17\}=0.52$

$S_2=\{C,D,E\}$ $P=\{0.17,0.17,0.16\}=0.46$

The difference is only $0.52-0.46=0.06$. This is the smallest possible difference when we

divide the message.

Attach 0 to S1 and 1 to S2.

Subdivide S1 into sub groups.

$S_{11}=\{A\}$ attach 0 to this $S_{12}=\{B\}$

attach 1 to this

Again subdivide S2 into subgroups considering the probability again.

S21={C} P={0.17}=0.17

S22={D,E} P={0.16,0.15}=0.31

Attach 0 to S21 and 1 to S22. Since S22 has more than one letter in it, we have to subdivide it.

S221={D} attach 0

S222={E} attach 1

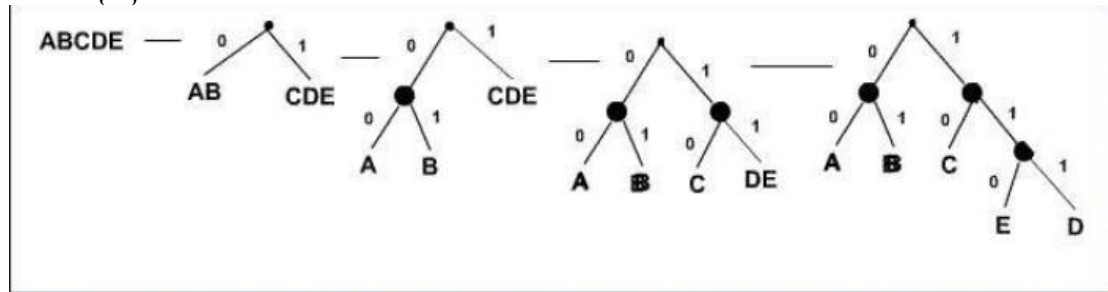


Fig Shannon-Fano coding tree

The message is transmitted using the following code (by traversing the tree)

A=00 B=01

C=10 D=110 E=111

Instead of transmitting ABCDE, we transmit 000110110111

Dictionary Encoding

Dictionary coding uses groups of symbols, words, and phrases with corresponding abbreviation. It transmits the index of the symbol/word instead of the word itself.

There

are different variations of dictionary based coding:

LZ77 (printed in 1977)

LZ78 (printed in 1978)

LZSS

LZW (Lempel-Ziv-Welch)

LZW Compression

LZW compression has its roots in the work of Jacob Ziv and Abraham Lempel. In 1977,

they published a paper on "sliding-window" compression, and followed it with another

paper in 1978 on "dictionary" based compression. These algorithms were named LZ77

and LZ78, respectively. Then in 1984, Terry Welch made a modification to LZ78 which

became very popular and was called LZW

The Concept

Many files, especially text files, have certain strings that repeat very often, for example " the ". With the spaces, the string takes 5 bytes, or 40 bits to encode. But what if we were to add the whole string to the list of characters? Then every time we came across " the ", we could send the code instead of 32,116,104,101,32.

This would take less no of bits.

This is exactly the approach that LZW compression takes. It starts with a dictionary of all the single character with indexes 0-255. It then starts to expand the dictionary as information gets sent through. Then, redundant strings will be coded, and compression has occurred.

The Algorithm:

LZWEncoding()

Enter all letters to the dictionary;

Initialize string s to the first letter from the input;

While any input is left read

symbol c; if s+c exists in the

dictionary s = s+c;

else

output codeword(s); //codeword for s

enter s+c to dictionary;

s =c;

end loop output

codeword(s);

Example: encode the ff string “aababacbaacbaadaa”

The program reads one character at a time. If the code is in the dictionary, then it adds the character to the current work string, and waits for the next one. This occurs on the first character as well. If the work string is not in the dictionary, (such as when the second character comes across), it adds the work string to the dictionary and sends over the wire the works string without the new character. It then sets the work string to the new character.

Example:

Encode the message aababacbaacbaadaaa using the above algorithm

Encoding : Create dictionary of letters found in the message

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a 2
			b 3
			c
		4	d

S is initialized to the first letter of message a ($s=a$)

Read symbol to c , and the next symbol is a ($c=a$)

Check if $s+c$ ($s+c=aa$) is found in the dictionary (the one created above in step 1).

It is not found. So add $s+c(s+c=aa)$ to dictionary and output codeword for $s(s=a)$.

The code for a is 1 from the dictionary. Then initialize s to c ($s=c=a$).

Encoder		Dictionary	
Input($s+c$)	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
aa	1	5	aa

Read the next letter from message to c ($c=b$)

Check if $s+c$ (ab) is found in the dictionary. It is not found. Then, add $s+c$ ($s+c=ab$)

into dictionary and output code for c ($c=b$). The codeword is 2. Then initialize s to c

($s=c=b$)

Encoder		Dictionary	
Input($s+c$)	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
aa	1	5	aa
ab	1		

		6	ab
--	--	---	----

Read the next letter to c ($c=a$).

Check if $s+c$ ($s+c=ba$) is found in the dictionary. It is not found.

Then add $s+c$ ($s+c=ba$)

to the dictionary. Then output the codeword for s ($s=b$). It is 2.

Then initialize s to c ($s=c=b$)

<u>Encoder</u>		<u>Dictionary</u>	
Input(s+c)	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
	1	5	aa
aa ab	1	6	ab
ba	2	7	ba

Read the next message to c (c=a). Then check if s+c (s+c=ab) is found in the dictionary.

It is there. Then initialize s to s+c (s=s+c=ab).

Read again the next letter to c (c=a). Then check if s+c (s+c=aba) is found in the dictionary. It is not there. Then transmit codeword for s (s=ab).

The code is 6. Initialize s to c(s=c=a)

<u>Encoder</u>		<u>Dictionary</u>	
Input(s+c)	Output	Index	Entry
		1	a b
		2	c d
		3	aa
		4	ab
aa	1	5	ba
ab	1	6	aba
ba	2	7	
aba	6	8	

Again read the next letter to c and continue the same way till the end of message. At last you will have the following encoding table.

<u>Encoder</u>		<u>Dictionary</u>	
Input(s+c)	Output	Index	Entry
		1	a b c
		2	d aa
		3	ab
	1	4	ba
	1	5	aba
	2	6	ac
	6	7	cb
aa ab	1	8	baa
ba aba	3	9	acb
ac cb	7	10	baad
baa	9	11	da
acb	11	12	aa
baad	4	13	
da aaa	5	14	
a	1	15	

Table encoding string

Now instead of the original message, you transmit their indexes in the dictionary. The code for the message is *1126137911451*

Decompression

The algorithm:

LZWDecoding()

Enter all the source letters into the dictionary;

Read priorCodeword and output one symbol corresponding to it;

While codeword is still left

read Codeword;

PriorString = string (PriorCodeword);

If codeword is in the dictionary

Enter in dictionary PriorString + firstsymbol(string(codeword));

output string(codeword);

else

Enter in the dictionary priorString +firstsymbol(priorString);

Output priorString+firstsymbol(priorstring); priorCodeword=codeword;

end loop

The nice thing is that the decompressor builds its own dictionary on its side, that matches

exactly the compressor's dictionary, so that only the codes need to be sent.

Example:

Let us decode the message *1126137911451*.

We will start with the following table

<u>Encoder</u>		<u>Dictionary</u>
Input	Output	IndexEntry
		1 a 2
		b 3
		c
		4 d

Read the first code. It is 1. Output the corresponding letter a

<u>Encoder</u>		<u>Dictionary</u>
Input	Output	IndexEntry
		1 a 2 b
		3 c
		4 d
1	a	

Read the next code. It is 1 and it is found in the dictionary. So add aa to the dictionary and output a again.

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
1	a		aa
1	a	5	

Read the next code which is 2. It is found in the dictionary. We add ab to dictionary and output b.

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
1	a		
1	a	5	aa
2	b	6	ab

Read the next code which is 6. It is found in the dictionary. Add ba to dictionary and output ab

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
1	a		
1	a	5	aa
2	b	6	ab
6	ab	7	ba

Read the next code. It is 1. 1 is found in the dictionary. Add aba to the dictionary and output a

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
1	a		
1	a	5	aa
2	b	6	ab
6	ab	7	ba
1	a	8	aba

Read the next code. It is 3 and it is found in the dictionary. Add ac to dictionary and output c.

Continue like this till the end of code is reached. You will get the following table:

<u>Encoder</u>		<u>Dictionary</u>	
Input	Output	Index	Entry
		1	a
		2	b
		3	c
		4	d
1		5	aa
1		6	ab
2	a a	7	ba
6	b	8	aba
1	ab	9	ac
3	a c	10	cb
7	ba	11	baa
9	ac	12	acb
11	baa	13	baad
4	d	14	da
5	aa	15	aaa
1	a		

The decoded message is aababacbaacbaadaaa

Huffman Compression

When we encode characters in computers, we assign each an 8-bit code based on an ASCII chart. But in most files, some characters appear more often than others. So wouldn't it make more sense to assign shorter codes for characters that appear more often and longer codes for characters that appear less often? D.A. Huffman published a paper in 1952 that improved the algorithm slightly and it soon superceded Shannon-Fano coding with the appropriately named Huffman coding.

Huffman coding has the following properties:

Codes for more probable characters are shorter than ones for less probable characters. Each code can be uniquely decoded

To accomplish this, Huffman coding creates what is called a *Huffman tree*, which is a binary tree.

First count the amount of times each character appears, and assign this as a *weight/probability* to each character, or node. Add all the nodes to a list.

Then, repeat these steps until there is only one node left:

- Find the two nodes with the lowest weights.
- Create a parent node for these two nodes. Give this parent node a weight of the sum of the two nodes.
- Remove the two nodes from the list, and add the parent node.

This way, the nodes with the highest weight will be near the top of the tree, and have shorter codes

Algorithm to create the tree

Assume the source alphabet $S=\{X_1, X_2, X_3, \dots, X_n\}$ and

Associated Probabilities $P=\{P_1, P_2, P_3, \dots, P_n\}$

Huffman()

For each letter create a tree with single root node and order all trees according to the probability of letter of occurrence;

while more than one tree is left take two trees t_1 , and t_2 with the lowest probabilities p_1 ,

p_2 and create a tree with

probability in its root equal to p_1+p_2 and with t_1 and t_2 as its subtrees;

associate 0 with each left branch and 1 with each right branch;

create unique codeword for each letter by traversing the tree the root to the leaf containing the probability corresponding to this letter and putting all encountered 0s and

1s together;

Example: Suppose the following source and related probability

$S=\{A,B,C,D,E\}$

$P=\{0.15,0.16,0.17,0.17,0.35\}$

Message="abcde"

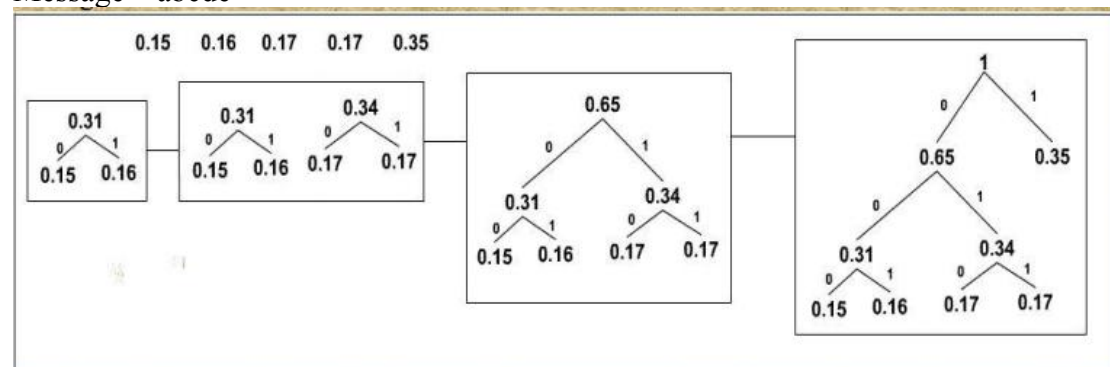


Fig Huffman tree

To read the codes from a Huffman tree, start from the root and add a 0 every time you go left to a child, and add a 1 every time you go right. So in this example, the code for the character b is 01 and the code for d is 110.

As you can see, a has a shorter code than d . Notice that since all the characters are at the leaves of the tree, there is never a chance that one code will be the prefix of another one

(eg. a is 01 and b is 011). Hence, this unique prefix property assures that each code can be uniquely decoded

The code for each letter is:

$a=000$ $b=001$ $c=010$ $d=011$

$e=1$

The original message will be encoded to:

abcde=0000010100111

To decode the message coded by Huffman coding, a conversion table had to be known by the receiver. Using this table, a tree can be constructed with the same path as the tree used for coding. Leaves store the same path as the tree used for coding. Leaves store letters instead of probabilities for efficiency purpose.

The decoder then can use the Huffman tree to decode the string by following the paths according to the string and adding a character every time it comes to one

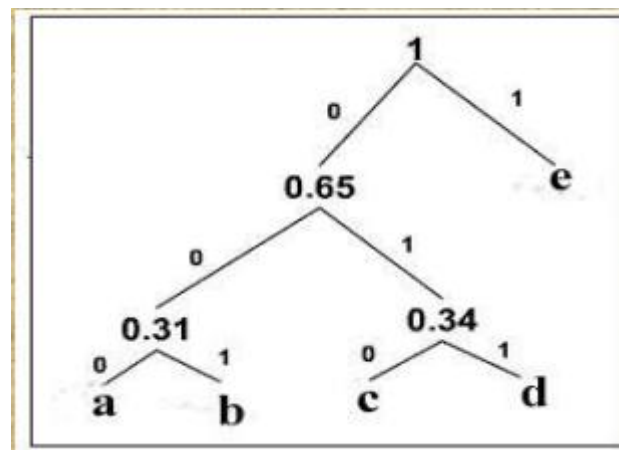


Fig Huffman tree

The Algorithm

Move left if you get 0

Move right if you get 1

If you get letter (reach leaf node) output that letter.

Go back and start from root again with the remaining code.

Using this algorithm and the above decoding tree, let us decode the encoded message 0000010100111 at destination.

0-move left

0-move left again

0-move left again, and we have reached leaf. Output the letter on the leaf node which is a.

Go back to root.

0-move left

0-move left

1-move right, and we have reached the leaf. Output letter on the leaf and it is b.

Go back to root.

0-move left

1-move right

0-move left, and we reach leaf. Output letter found on the leaf which is c.

Go back to root.

0-move left

1-move right

1-move right, and we reach leaf. Output letter on leaf which is d.

Go back to root.

1-move right, and we reach leaf node. Output the letter on the node which is e. Now we have finished i.e. no more code remains. Display the letters output as message.

Abcde

How can the encoder let the decoder know which particular coding tree has been used?

Two ways:

i) Both agree on particular Huffman tree and both use it for sending any message ii)

The encoder constructs Huffman tree afresh every time a new message is sent and sends the conversion table along with the message. This is more versatile, but has additional overload—sending conversion table. But for large data, there is the advantage. It is also possible to create tree for pairs of letters. This improves performance.

Example:

$S=\{x, y, z\}$

$P=\{0.1, 0.2, 0.7\}$

To get the probability of pairs, multiply the probability of each letter.

$xx=0.1*0.1=0.01$ $xy=0.1*0.2=0.02$ $xz=0.1*0.3=0.07$

$yx=0.2*0.1=0.02$ $yy=0.2*0.2=0.04$ $yz=0.2*0.7=0.14$

$zx=0.7*0.1=0.07$ $zz=0.7*0.7=0.49$ $zy=0.7*0.2=0.14$

Using these probabilities, you can create Huffman tree of pairs the same way as we did previously.

Arithmetic Coding

The entire data set is represented by a single rational number, whose value is between 0 and 1. This range is divided into sub-intervals each representing a certain symbol.

The number of sub-intervals is identical to the number of symbols in the current set of symbols and the size is proportional to their probability of appearance.

For each symbol in the original data a new interval division takes place, on the basis of the last subinterval.

Algorithm:

ArithmeticEncoding(message)

CurrentInterval=[0,1); //includes 0 but not 1

while the end of message is not reached

read letter X_i from message;

divide the CurrentInterval into SubInterval $IR_{\text{CurrentInterval}}$;

CurrentInterval=SubInterval $_i$ in CurrentInterval;

Output bits uniquely identifying CurrentInterval;

Assume the source alphabet $s=\{X_1, X_2, X_3, \dots, X_n\}$ and associated probability of

$P=\{p_1,$

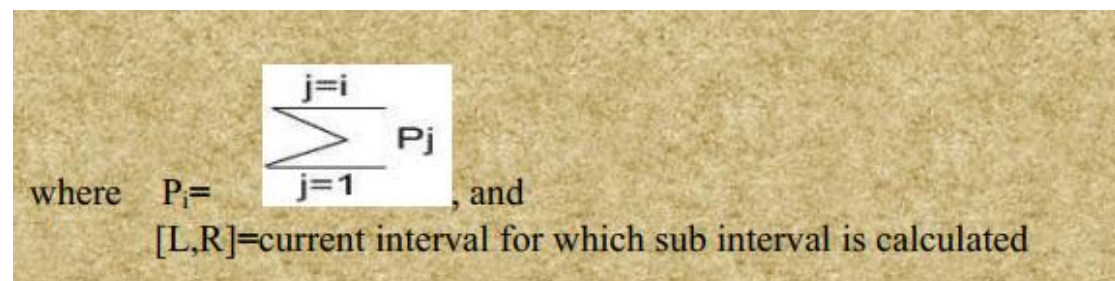
$p_2, p_3, \dots, p_n\}$

To calculate sub interval of current interval $[L,R]$, use the following formula

$IR_{[L,R]}=\{[L, L+(R-L)*P_1], [L+(R-L)*P_1, L+(R-L)*P_2], [L+(R-L)*P_2,$

$L+(R-L)*P_3], \dots,$

$[L+(R-L)*P_{n-1}, L+(R-L)*P_1]\}$



Cumulative probabilities are indicated using capital P and single probabilities are indicated using small p.

EXAMPLE:

Encode the message *ABBC#* using arithmetic encoding.

$s=\{a,b,c,\#\}$ $p=\{0.4,0.3,0.1,0.2\}$

At the beginning CurrentInterval is set to $[0,1)$. Let us calculate subintervals of $[0,1)$.

First let us get cumulative probability P_i

$P_1=0.4$

$P_2=0.4+0.3=0.7$

$P_3=0.4+0.3+0.1=0.8$

$P_4=0.4+0.3+0.1+0.2=1$

Next calculate subintervals of $[0,1)$ using the formula given above.

$$IR[0,1]=\{[0,0+(1-0)*0.4],[0+(1-0)*0.4, 0+(1-0)*0.7], [0+(1-0)*0.7, 0+(1-0)*0.8], [0+(1-0)*0.8, 0+(1-0)*1]\}$$

$$IR[0,1]=\{[0,0.4],[0.4,0.7],[0.7,0.8],[0.8,1)\} - \text{four subintervals}$$

Now the question is, which one of the SubIntervals will be the CurrentInterval? To determine this, read the first letter of the message. It is a. Look where a is found in the source alphabet. It is found at the beginning. So the next CurrentInterval will be $[0,4)$ which is also found at the beginning in the SubIntervals.

Again let us calculate the SubIntervals of CurrentInterval $[0,0.4)$. The cumulative probability does not change i.e. the same as previous.

$$IR[0,0.4]=\{[0,0+(0.4-0)*0.4],[0+(0.4-0)*0.4, 0+(0.4-0)*0.7],[0+(0.4-0)*0.7, 0+(0.4-0)*0.8], [0+(0.4-0)*0.8, 0+(0.4-0)*1]\}$$

$$IR[0,0.4]=\{[0,0.16],[0.16,0.28],[0.28,0.32],[0.32,0.4)\}.$$

Which interval will be the next CurrentInterval? Read the next letter from message. It is b. B is found in the second place in the source alphabet list. The next CurrentInterval will be the second SubInterval i.e $[0.16,0.28)$.

Continue like this till there is letter left in the message. You will get the following result:

$$IR[0.16,0.28]=\{[0.16,0.208],[0.208,0.244],[0.244,0.256],[0.256,0.28)\}.$$
 Next

$$IR[0.208,0.244]=\{[0.208,0.2224],[0.2224,0.2332],[0.2332,0.2368],[0.2368,0.242)\}.$$

Next

$$IR[0.2332,0.2368]=\{[0.2332,0.23464],[0.23464,0.23572],[0.23572,0.23608],[0.23608, 0.2368)\}$$

We are done because no more letter remained in the message. The last letter read was #.

It is the fourth letter in source alphabet. So take the fourth SubInterval as

CurrentInterval

i.e $[0.23608, 0.2368]$. Now any number between the last CurrentInterval is sent as the message. So you can send 0.23608 as the encoded message or any number between 0.23608, and 0.2368.

Diagrammatically, calculating SubIntervals look like this

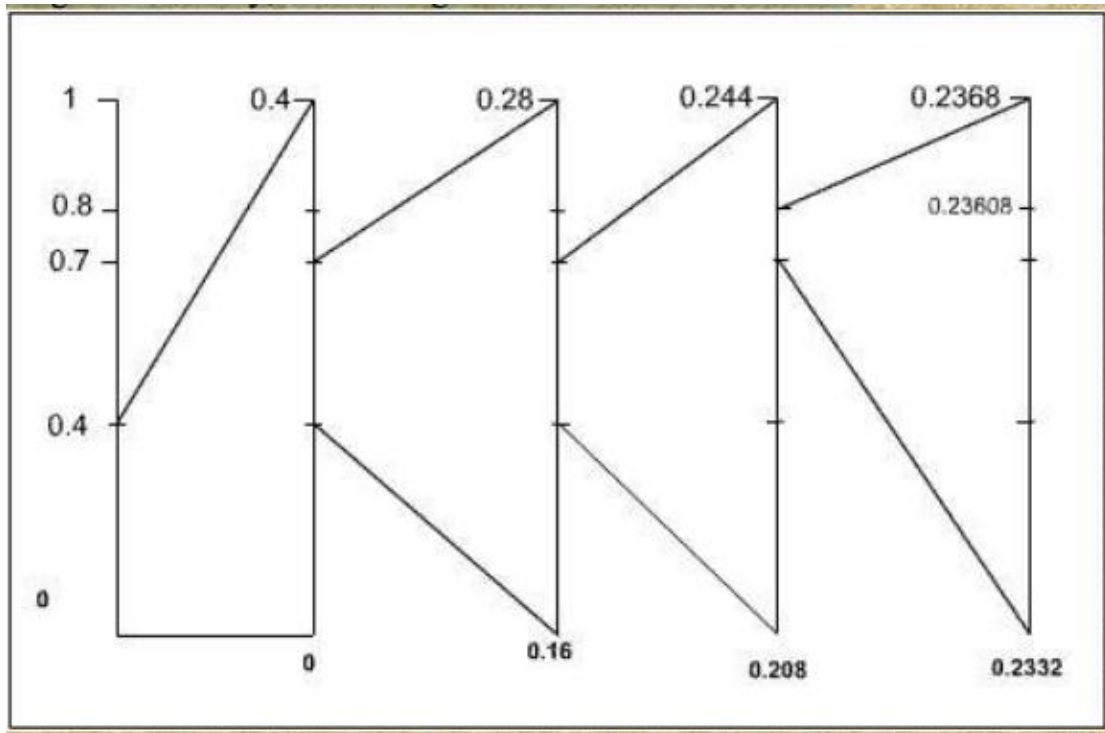


Fig sub interval and current interval

DECODING Algorithm:

ArithmeticDecoding(codeword)

CurrentInterval=[0, 1];

While (1)

Divide CurrentInterval into SubIntervals $IR_{currentInterval}$;

Determine the SubInterval_i of CurrentInterval to which the codeword belongs;

Output letter X_i *corresponding to this SubInterval;*

If end of file Return;

CurrentInterval=*SubInterval_i in* $IR_{currentInterval}$; *End*

of while

Example:

Decode 0.23608 which we previously encoded.

To decode the source alphabet and related probability should be known by destination.

Let us use the above source and probability.

$s=\{a,b,c,\#\}$ $p=\{0.4,0.3,0.1,0.2\}$

First set CurrentInterval to [0,1], and then calculate SubInterval for it. The formula to calculate the SubInterval is the same to encoding. The cumulative probabilities are:

$$P1=0.4$$

$$P2=0.4+0.3=0.7$$

$$P3=0.4+0.3+0.1=0.8$$

$$P4=0.4+0.3+0.1+0.2=1$$

$$IR[0,1]=\{[0,0+[1-0]*0.4],[0+[1-0]*0.4,0+[1-0]*0.7],[0+[1-0]*0.7,0+[1-0]*0.8],[0+[1-0]*0.8,0+[1-0]*1]\}$$

$IR[0,1]=\{[0,0.4],[0.4,0.7],[0.7,0.8],[0.8,1]\}$. Now check in which SubInterval the encode

message falls. It falls in the first SubInterval i.e [0,0.4]. Output the first letter from source

alphabet. It is a. Set CurrentInterval to [0,0.4]

$$IR[0,0.4]=\{[0,0+(0.4-0)*0.4],[0+(0.4-0)*0.4,0+(0.4-0)*0.7],[0+(0.4-0)*0.7,0+(0.4-0)*0.8],[0+(0.4-0)*0.8,0+(0.4-0)*1]\}$$

$IR[0,0.4]=\{[0,0.16],[0.16,0.28],[0.28,0.32],[0.32,0.4]\}$. Again check where 0.23608 falls.

It falls in the second SubInterval i.e [0.16,0.28]. Set CurrentInterval to this SubInterval.

Output the second letter from source alphabet. It is b.

$IR[0.16,0.28]=\{[0.16,0.208],[0.208,0.244],[0.244,0.256],[0.256,0.28]\}$. 0.23608 falls in

the second SubInterval. Output the second letter from source alphabet. It is b.

$IR[0.208,0.244]=\{[0.208,0.2224],[0.2224,0.2332],[0.2332,0.2368],[0.2368,0.242]\}$. falls

in the third SubInterval. Output the third letter from source alphabet. It is c.

$IR[0.2332,0.2368]=\{[0.2332,0.23464],[0.23464,0.23572],[0.23572,0.23608],[0.23608,0.2368]\}$. 0.23608 falls in the fourth SubInterval. Output fourth letter which is #. Now

end of the message has been reached.

Disadvantage: arithmetic precision of computer is soon suppressed and hence large message can't be encoded

Implementation of Arithmetic Coding

To solve the above disadvantage, arithmetic coding is implemented as follows

```
Algorithm:

OutputBits()
{
    While(1)
    If CurrentInterval [0,0.5)
        Output 0 and bitcount 1s; //and here shows concatenation Bitcount=0;
    Else if CurrentInterval [0.5,1)
        Output 1 and bitcount 0s;

        Bitcount=0;
    Subtract 0.5 from left and right bounds of CurrentInterval;
    Else if CurrentInterval [0.25,0.75)
        Bitcount++;
    Subtract 0.25 both left and right bounds of CurrentInterval;
    Else Break;
    Double left and right bounds of CurrentInterval;
}

FinishArithmeticCoding()
{ bitcount++;
  if lowerbound of CurrentInterval
  <0.25 output 0 and bitcount 1s; else
  output 1 and bitcount 0s;
}

ArithmeticEncoding(message)
{
    CurrentInterval=[0,1];
    Bitcount=0;
    While the end of message is not reached
    { read letter  $X_i$  from message;
      divide CurrentInterval into SubInterval  $IR_{CurrentInterval}$ ;

      CurrentInterval=SubIntervali in  $IR_{CurrentInterval}$ ;
      OutputBits();
    }
    FinishArithmeticEncoding();
}
```

Example:

Encode the message abbc#.

$s = \{a, b, c, \#\}$ $p = \{0.4, 0.3, 0.1, 0.2\}$

CurrentInterval	input	output	bitcount	Subintervals			
[0,1)	a		0	[0,0.4)	[0.4,0.7)	[0.7,0.8)	[0.8, 1)
[0,0.4)		0					
[0,0.8)	b			[0,0.32)	[0.32, 0.56)	[0.56,0.64)	[0.64,0.8)
[0.32,0.56)		-	1				

[0.14,0.62)	b			[0.14,0.332)	[0.332,0.476)	[0.476,0.524)	[0.524,0.62)
[0.332,0.476)		01	0				
[0.664,0.952)		1					
[0.328,0.904)	c			[0.328,0.5584)	[0.5584,0.7312)	[.7312, .7888)	[.7888, .904)
[.7312, .788)		1					
[.4624, .5776)		-	1				
[.4248, .6552)		-	2				
[.3496, .8104)	#			[.3496, .53392)	[.53392, .67216)	[.67216, .71824)	[.71824, .8104)
[.71824, .8104)		100	0				
[.43648, .6208)		-	1				
[.37296, .7416)		-	2				
[.24592, .9832)		0111					

The final code will be 001111000111 from the output column of table.