Data Compression Algorithms and Techniques

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Computer Science & Engineering

under the Supervision of

Prof. Dr. Satya Prakash Ghrera

By

Ankit Srivastava- Roll no:111255

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "Data Compression Algorithms and Techniques", submitted by Ankit Srivastava in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Prof. Dr. Satya Prakash Ghrera

Professor, Brig(Retd) Head, Dept.of CSE.

Signature:

Date:

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

I am highly indebted to respected Dr. Satya Prakash Ghrera for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards the Project Co-ordinator Dr. Hemraj Saini for their kind co-operation and encouragement which help me in completion of this project. I would like to express my special gratitude and thanks to industry persons for giving me such attention and time. My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Date: 12/05/2015 Ankit Srivastava

CONTENTS

Chapter Name **Page No.**

1. Introduction to Data compression **10-15**

1.1 Lossy and Lossless Data Compression

1.2 Data Compression and Information Theory

1.3 Modelling and Coding

2. Lossless Data Compression **16-22**

2.1 Statistical Model for Compression

2.2 Dictionary Schemes

2.3 liv and Zempel

2.4 Other Lossless Compression

2.4.1 Burrows Wheeler

3. Data Encoding Techniques **23-34**

3.1 Probability Coding

3.1.1 prefix codes

3.1.2 arithmetic coding

3.2 Applications

3.2.1 Run Length Codes

3.2.2 Context coding

3.2.3 Context coding PPM

4. Huffman and Arithmetic Coding **35-41**

4.1 Huffman Algorithm

4.2 Compression with Arithmetic Coding

5. Context based Modeling for Data Compression **42-50**

5.1 Bascs for context based modeling

5.2 Methods for Blending

5.2.1 Introduction

5.2.2 Escape strategy

5.2.3 Exclusion principle

5.2.4 Memory limitations

5.3 context modeling for order 2

5.3.1 blending Strategy

5.3.2 self organizing list

5.3.3 Frequency distribution

5.3.4 Escape strategy

5.4 encoding the model

6. Prediction with partial matching **51-60**

6.1 introduction

6.2 PPM Algorithm

6.2.1 PPMC

6.2.2 Performance of PPMC

6.3 longer context

6.3.1 context Tries

7. Results **61-63**

8. References **64**

LIST OF FIGURES

1. Figure 1.1 A Statistical Model with a Huffman Encoder. Page 11

2. Fig 2.1 General Adaptive Compression Page 17

3. Fig 2.3 General Adaptive Decompression page 17

4. Fig. 3.1 Group 3 Run Length Huffman Codes page 28

5. Figure 3.2 Example of PPM page 30

6. Fig 3.4 Shannon Fano Code page 33

7. Fig 4.1 The Huffman Tree After Two Passes page 37

8. Fig 4.2 next pass page 37

9. fig 6.1 compression ratios varies with context length page 57

10 fig 6.2 context trie for string page 60

List of Tables

1. Table 3.1 page 25
2. Table 3.2 page 26
3. Table 6.1 page 55
4. Table 6.2 page 56

Abstract

Under this Project Report on “Data Compression Algorithms sand Techniques”, the aim is to survey of various Data Compression techniques known till date and understanding their working and how they actually contribute towards compression of data, since data compression is being widely used these days in almost every field related with computer science. Either it be data communication ,data storage or the expensive cost associated with the consumption of expensive resources like disc space or connection bandwidth data with lesser bytes is needed everywhere. Within this project work although thorough study has been done on various Data Compression Algorithms but as far as implementation is concerned we have limited our work for the Algorithms that provide Lossless Data Compression.

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. The purpose of this paper is to present and analyze a variety of data compression algorithms. A simple characterization of data compression is that it involves transforming a string of characters in some representation (into any form) 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.

Many data processing applications require storage of large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, 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.

Similarly, compressing applicable to half of its original size is equivalent to doubling the capacity of the storage medium. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.

Chapter 1: INTRODUCTION TO DATA COMPRESSION

1. 1 Lossy and Lossless Data Compression

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand.

The task of compression consists of two components, an encoding algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation. We distinguish between *lossless algorithms*, which can reconstruct the original message exactly from the compressed message, and *lossy Algorithms*, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters.

all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of the messages – *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in "typical" images.

Compression is therefore all about probability. When discussing compression algorithms The *model* component somehow captures the probability distribution of the messages by knowing or discovering something about the structure of the input. The *coder* component then takes advantage of the probability biases generated in the model to generate codes. It does this by effectively lengthening low probability messages and shortening high-probability messages. A model, for example, might have a generic “understanding” of human faces knowing that some “faces” are more likely than others (*e.g.*, a teapot would not be a very likely face). The coder would then be able to send shorter messages for objects that look like faces. This could work well for compressing teleconference calls. The models in most current real-world compression algorithms, however, are not so sophisticated, and use more mundane measures such as repeated patterns in text. Although there are many different ways to design the model component of compression algorithms and a huge range of levels of sophistication, the coder components tend to be quite generic—in current algorithms are almost exclusively based on either Huffman or arithmetic codes. Lest we try to make to fine of a distinction here, it should be pointed out that the line between model and coder components of algorithms is not always well defined.

It turns out that information theory is the glue that ties the model and coder components together. In particular it gives a very nice theory about how probabilities are related to information content and code length. As we will see, this theory matches practice almost perfectly, and we can achieve code lengths almost identical to what the theory predicts.

Another question about compression algorithms is how does one judge the quality of one versus another. In the case of lossless compression there are several criteria I can think of, the time to compress, the time to reconstruct, the size of the compressed messages, and the generality—*i.e.*, does it only work on Shakespeare or does it do Byron too. In the case of lossy compression the judgment is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on your application one might be more important than another and one would want to pick your algorithm appropriately.

Perhaps the best attempt to systematically compare lossless compression algorithms is the Archive Comparison Test (ACT) by Jeff Gilchrist.

It reports times and compression ratios for 100s of compression algorithms over many databases. It also gives a score based on a weighted average of runtime and the compression ratio.

1.2. Data Compression Using Information Theory

Data compression is perhaps the fundamental expression of Information Theory. Information Theory is a branch of mathematics that had its genesis in the late 1940s with the work of Claude Shannon at Bell Labs. It concerns itself with various questions about information, including different ways of storing and communicating messages. Data compression enters into the field of Information Theory because of its concern with redundancy. Redundant information in a message takes extra bit to encode, and if we can get rid of that extra information, we will have reduced the size of the message. Information Theory uses the term entropy as a measure of how much information is encoded in a message. The word entropy was borrowed from thermodynamics, and it has a similar meaning. The higher the entropy of a message, the more information it contains. The entropy of a symbol is defined as the negative logarithm of its probability. To determine the information content of a message in bits, we express the entropy using the base 2 logarithm:

Number of bits = - Log base 2 (probability)

The entropy of an entire message is simply the sum of the entropy of all individual symbols. Entropy fits with data compression in its determination of how many bits of information are actually present in a message. If the probability of the character 'e' appearing in this manuscript is 1/16, for example, the information content of the character is four bits. So the character string "eeeee" has a total content of 20 bits. If we are using standard 8-bit ASCII characters to encode this message, we are actually using 40 bits.

The difference between the 20 bits of entropy and the 40 bits used to encode the message is where the potential for data compression arises. One important fact to note about entropy is that, unlike the thermodynamic measure of entropy, we can use no absolute number for the information content of a given message. The problem is that when we calculate entropy, we use a number that gives us the probability of a given symbol.

1.3 Modeling and Coding

In general, data compression consists of taking a stream of symbols and transforming them into codes. If the compression is effective, the resulting stream of codes will be smaller than the original symbols. The decision to output a certain code for a certain symbol or set of symbols is based on a model. The model is simply a collection of data and rules used to process input symbols and determine which code(s) to output. A program uses the model to accurately define the probabilities for each symbol and the coder to produce an appropriate code based on those probabilities. Modeling and coding are two distinctly different things. People frequently use the term coding to refer to the entire data-compression process instead of just a single component of that process. You will hear the phrases "Huffman coding" or "Run-Length Encoding," for example, to describe a data-compression technique, when in fact they are just coding methods used in conjunction with a model to compress data. Using the example of Huffman coding, a breakdown of the compression process looks something like this:

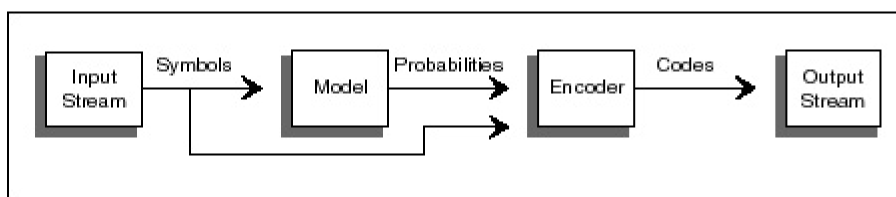


Figure 1.1 A Statistical Model with a Huffman Encoder.

In the case of Huffman coding, the actual output of the encoder is determined by a set of probabilities. When using this type of coding, a symbol that has a very high probability of occurrence generates a code with very few bits. A symbol with a low probability generates a code with a larger number of bits. We think of the model and the program's coding process as different because of the countless ways to model data, all of which can use the same coding process to produce their output. A simple program using Huffman coding, for example, would use a model that gave the raw probability of each symbol occurring anywhere in the input stream. A more sophisticated program might calculate the probability based on the last 10 symbols in the input stream. Even though both probably be radically different. So when the topic of coding methods comes up at your next cocktail party, be alert for statements like "Huffman coding in general doesn't produce very good compression ratios." This would be your perfect opportunity to respond with "That's like saying Converse sneakers don't go very fast. I always thought the leg power of the runner had a lot to do with it." If the conversation has already dropped to the point where you are discussing data compression, this might even go over as a real demonstration of wit

One important fact to note about entropy is that, unlike the thermodynamic measure of entropy, we can use no absolute number for the information content of a given message. The problem is that when we calculate entropy, we use a number that gives us the probability of a given symbol. The probability figure we use is actually the probability for a given model, not an absolute number. If we change the model, the probability will change with it. How probabilities change can be seen clearly when using different orders with a statistical model. A statistical model tracks the probability of a symbol based on what symbols appeared previously in the input stream. The order of the model determines how many previous symbols are taken into account. An order-0 model, for example, won't look at previous characters. An order-1 model looks at the one previous character, and so on. The different order models can yield drastically different probabilities for a character. The letter 'u' under an order-0 model, for example, may have only a 1 percent probability of occurrence.

But under an order-1 model, if the previous character was ‘q,’ the ‘u’ may have a 95 percent probability.

In order to compress data well, we need to select models that predict symbols with high probabilities. A symbol that has a high probability has a low information content and will need fewer bits to encode. Once the model is producing high probabilities, the next step is to encode the symbols using an appropriate number of bits.

Chapter 2: LOSSLESS DATA COMPRESSION

2.1 Statistical Modeling for Data Compression

Lossless data compression is generally implemented using one of two different types of modeling: statistical or dictionary-based. Statistical modeling reads in and encodes a single symbol at a time using the probability of that character's appearance. Dictionarybased modeling uses a single code to replace strings of symbols. In dictionary-based modeling, the coding problem is reduced in significance, leaving the model supremely important.

2.1.1

The simplest forms of statistical modeling use a static table of probabilities. In the earliest days of information theory, the CPU cost of analyzing data and building a Huffman tree was considered significant, so it wasn't frequently performed. Instead, representative blocks of data were analyzed once, giving a table of character-frequency counts. Huffman encoding/decoding trees were then built and stored. Compression programs had access to this static model and would compress data using it. But using a universal static model has limitations. If an input stream doesn't match well with the previously accumulated statistics, the compression ratio will be degraded— possibly to the point where the output stream becomes larger than the input stream. The next obvious enhancement is to build a statistics table for every unique input stream.

Building a static Huffman table for each file to be compressed has its advantages. The table is uniquely adapted to that particular file, so it should give better compression than a universal table. But there is additional overhead since the table (or the statistics used to build the table) has to be passed to the decoder ahead of the compressed code stream. For an order-0 compression table, the actual statistics used to create the table may take up as little as 256 bytes—not a very large amount of overhead. But trying to achieve better

compression through use of a higher order table will make the statistics that need to be passed to the decoder grow at an alarming rate. Just moving to an order 1 model can boost the statistics table from 256 to 65,536 bytes.

Though compression ratios will undoubtedly improve when moving to order-1, the overhead of passing the statistics table will probably wipe out any gains. For this reason, compression research in the last 10 years has concentrated on adaptive models. When using an adaptive model, data does not have to be scanned once before coding in order to generate statistics. Instead, the statistics are continually modified as new characters are read in and coded.

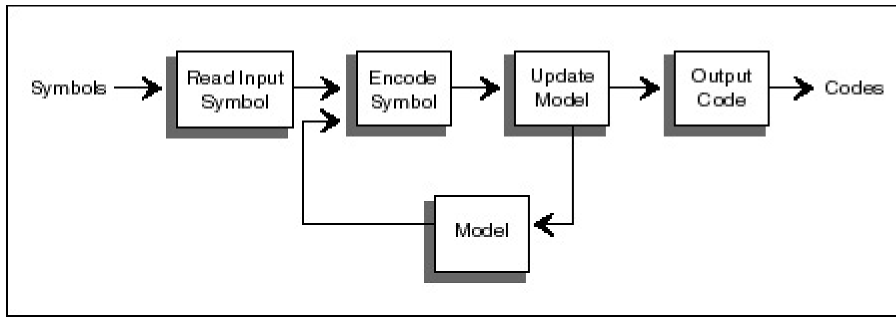


Fig 2.1 General Adaptive Compression

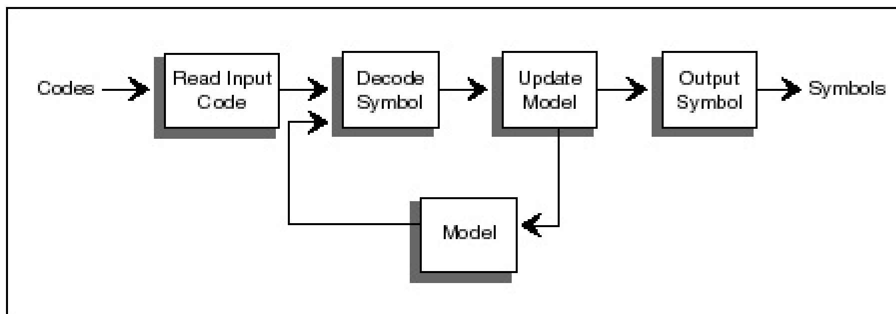


Fig 2.3 General Adaptive Decompression

The important point in making this system work is that the box labeled “Update Model” has to work exactly the same way for both the compression and decompression programs. After each character (or group of characters) is read in, it is encoded or decoded. Only after the encoding or decoding is complete can the model be updated to take into account the most recent symbol or group of symbols.

One problem with adaptive models is that they start knowing essentially nothing about the data. So when the program first starts, it doesn't do a very good job of compression. Most adaptive algorithms tend to adjust quickly to the data stream and will begin turning in respectable compression ratios after only a few thousand bytes. Likewise, it doesn't take long for the compression-ratio curve to flatten out so that reading in more data doesn't improve the compression ratio.

One advantage that adaptive models have over static models is the ability to adapt to local conditions. When compressing executable files, for example, the character of the input data may change drastically as the program file changes from binary program code to binary data. A well-written adaptive program will weight the most recent data higher than old data, so it will modify its statistics to better suit changed data.

2.2 Dictionary Schemes

Statistical models generally encode a single symbol at a time— reading it in, calculating a probability, then outputting a single code. A dictionary-based compression scheme uses a different concept. It reads in input data and looks for groups of symbols that appear in a dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression ratio. This method of encoding changes the focus of dictionary compression. Simple coding methods are generally used, and the focus of the program is on the modeling. In LZW compression, for example, simple codes of uniform width are used for all substitutions. A static dictionary is used like the list of references in an academic paper. The dictionary is static because it is built up and transmitted with the text of work—the reader does not have to build it on the fly. The first time I see a number in the text like this—[2]—I know it points to the static dictionary.

The problem with a static dictionary is identical to the problem the user of a statistical model faces: The dictionary needs to be transmitted along with the text, resulting in a

certain amount of overhead added to the compressed text. An adaptive dictionary scheme helps avoid this problem. Mentally, we are used to a type of adaptive dictionary when performing acronym replacements in technical literature. The standard way to use this adaptive dictionary is to spell out the acronym, then put its abbreviated substitution in parentheses. So the first time I mention the Massachusetts Institute of Technology (MIT), I define both the dictionary string and its substitution. From then on, referring to MIT in the text should automatically invoke a mental substitution.

2.3 Ziv and Lempel

Until 1980, most general-compression schemes used statistical modeling. But in 1977 and 1978, Jacob Ziv and Abraham Lempel described a pair of compression methods using an adaptive dictionary. These two algorithms sparked a flood of new techniques that used dictionary-based methods to achieve impressive new compression ratios.

3.1 LZ77

The first compression algorithm described by Ziv and Lempel is commonly referred to as LZ77. It is relatively simple. The dictionary consists of all the strings in a window into the previously read input stream. A file-compression program, for example, could use a 4K-byte window as a dictionary. While new groups of symbols are being read in, the algorithm looks for matches with strings found in the previous 4K bytes of data already read in. Any matches are encoded as pointers sent to the output stream. LZ77 and its variants make attractive compression algorithms. Maintaining the model is simple; encoding the output is simple; and programs that work very quickly can be written using LZ77. Popular programs such as PKZIP and LHarc use variants of the LZ77 algorithm, and they have proven very popular.

LZ78

The LZ78 program takes a different approach to building and maintaining the dictionary. Instead of having a limited-size window into the preceding text, LZ78 builds its dictionary out of all of the previously seen symbols in the input text. But instead of having carte blanche access to all the symbol strings in the preceding text, a dictionary of strings is built a single character at a time. The first time the string "Mark" is seen, for example, the string "Ma" is added to the dictionary. The next time, "Mar" is added. If "Mark" is seen again, it is added to the dictionary.

This incremental procedure works very well at isolating frequently used strings and adding them to the table. Unlike LZ77 methods, strings in LZ78 can be extremely long, which allows for high-compression ratios. LZ78 was the first of the two Ziv-Lempel algorithms to achieve popular success, due to the LZW adaptation by Terry Welch, which

2.4 Other Lossless Compression

2.4.1 Burrows Wheeler

The Burrows Wheeler algorithm is a relatively recent algorithm. An implementation of the algorithm called `bzip`, is currently one of the best overall compression algorithms for text. It gets compression ratios that are within 10% of the best algorithms such as PPM, but runs significantly faster.

Rather than describing the algorithm immediately, lets try to go through a thought process that

1. leads to the algorithm. Recall that the basic idea of PPM was to try to find as long a context as

(a)			(b)		(c)	
	a	ccbaccacba	$ccbaccacba_4$	a_1	$cbaccacbaa_1$	c_1
a	c	cbaccacba	$cbaccacbaa_1$	c_1	$ccbaccacba_4$	a_1
ac	c	baccacba	$baccacbaac_1$	c_2	$cacbaaccba_2$	c_3
acc	b	accacba	$accacbaacc_2$	b_1	$baaccbacca_3$	c_5
accb	a	ccacba	$ccacbaaccb_1$	a_2	$accbaccacb_2$	a_4
accba	c	cacba	$cacbaaccba_2$	c_3	$ccacbaaccb_1$	a_2
accbac	c	acba	$acbaaccbac_3$	c_4	$baccacbaac_1$	c_2
accbacc	a	cba	$cbaaccbacc_4$	a_3	$acbaaccbac_3$	c_4
accbacca	c	ba	$baaccbacca_3$	c_5	$aaccbaccac_5$	b_2
accbaccac	b	a	$aaccbaccac_5$	b_2	$accacbaacc_2$	b_1
accbaccacb	a		$accbaccacb_2$	a_4	$cbaaccbacc_4$	a_3

Fig 2.4

possible that matched the current context and use that to effectively predict the next character. A problem with PPM is in selecting k. If we set k too large we will usually not find matches and end up sending too many escape characters. On the other hand if we set it too low, we would not be taking advantage of enough context. We could have the system automatically select k based on which does the best encoding, but this is expensive. Also within a single text there might be some very long contexts that could help predict, while most helpful contexts are short. Using a fixed k we would probably end up ignoring the long contexts. Lets see if we can come up with a way to take advantage of the context that somehow automatically adapts. Ideally we would like the method also to be a bit faster. Consider taking the string we want to compress and looking at the full context for each character—*i.e.*, all previous characters from the start of the string up to the character. In fact, to make the contexts the same length, which will be convenient later, we add to the head of each context the part of the string following the character making each context $n - 1$ characters. Examples of the context for each character of the string accbaccacba are given in Figure 6.1. Now lets sort these contexts based on reverse lexical order, such that the last character of the context is the most significant (see Figure).

Note that now characters with the similar contexts (preceeding characters) are near each

other. In fact, the longer the match (the more preceeding characters that match identically) the closer they will be to each other. This is similar to PPM in that it prefers longer matches when "grouping", but will group things with shorter matches when the longer match does not exist. The difference is that there is no fixed limit k on the length of a match—a match of length 100 has priority over a match of 99.

\In practice the sorting based on the context is executed in blocks, rather than for the full message sequence. This is because the full message sequence and additional data structures required for sorting it, might not fit in memory. The process of sorting the characters by their context 38 is often refered to as a *block-sorting transform*. In the dicussion below we will refer to the sequence of characters generated by a block-sorting transform as the *context-sorted sequence* (*e.g.*,\ c1a1c3c5a4a2c2c4b2b1a3 in Figure 6.1). Given the correlation between nearyby characters in a context-sorted sequence, we should be able to code them quite efficiently by using, for example, a move-to-front coder . For long strings with somewhat larger character sets this technique should compress the string significantly since the same character is likely to appear in similar contexts. Experimentally, in fact, the technique compresses about as well as PPM even though it has no magic number k or magic way to select the escape probabilities.
The problem remains, however, of how to reconstruct the original sequence from contextsorted sequence. The way to do this is the ingenious contribution made by Burrows and Wheeler.

You might try to recreate it before reading on. The order of the most-significant characters in the sorted contexts plays an important role in decoding. In the example of in Figure , these are a1a4a2a3b2b1c1c3c5c2c4. The characters are sorted, but equal valued characters do not necessarily appear in the same order as in the input sequence. The following lemma is critical in the algorithm for efficiently reconstruct the sequence.

Chapter 3: DATA ENCODING TECHNIQUES

3.1 Probability Coding

As mentioned in the introduction, coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. How the probabilities are generated is part of the model component of the algorithm.

In practice we typically use probabilities for parts of a larger message rather than for the complete message, *e.g.*, each character or word in a text. We will consider each of these components a message on its own, and we will use the term *message sequence* for the larger message made up of these components. In general each little message can be of a different type and come from its own probability distribution. For example, when sending an image we might send a message specifying a color followed by messages specifying a frequency component of that color. Even the messages specifying the color might come from different probability distributions since the probability of particular colors might depend on the context.

We distinguish between algorithms that assign a unique code (bit-string) for each message, and ones that "blend" the codes together from more than one message in a row. In the first class we will consider Huffman codes, which are a type of prefix code. In the later category we consider arithmetic codes. The arithmetic codes can achieve better compression, but can require the encoder to delay sending messages since the messages need to be combined before they can be sent.

3.1.1 Prefix Codes

Once Information Theory had advanced to where the number of bits of information in a symbol could be determined, the next step was to develop new methods for encoding information. To compress data, we need to encode symbols with exactly the number of

bits of information the symbol contains. If the character 'e' only gives us four bits of information, then it should be coded with exactly four bits. If 'x' contains twelve bits, it should be coded with twelve bits. By encoding characters using EBCDIC or ASCII, we clearly aren't going to be very close to an optimum method. Since every character is encoded using the same number of bits, we introduce lots of error in both directions, with most of the codes in a message being too long and some being too short. Solving this coding problem in a reasonable manner was one of the first problems tackled by practitioners of Information Theory. Two approaches that worked well were Shannon-Fano coding and Huffman coding—two different ways of generating variable-length codes when given a probability table for a given set of symbols. Huffman coding, named for its inventor D.A. Huffman, achieves the minimum amount of redundancy possible in a fixed set of variable-length codes. This doesn't mean that Huffman coding is an optimal coding method. It means that it provides the best approximation for coding symbols when using fixed-width codes. The problem with Huffman or Shannon-Fano coding is that they use an integral number of bits in each code. If the entropy of a given character is 2.5 bits, the Huffman code for that character must be either 2 or 3 bits, not 2.5. Because of this, Huffman coding can't be considered an optimal coding method, but it is the best approximation that uses fixed codes with an integral number of bits. Here is a sample of Huffman codes

Symbol	Huffman Code
E	100
T	101
A	1100
I	11010
- - - -	
X	01101111
Q	01101110001
Z	01101110000

Table 1

Though Huffman coding is inefficient due to using an integral number of bits per code, it is relatively easy to implement and very economical for both coding and decoding.

Huffman first published his paper on coding in 1952, and it instantly became the most cited paper in Information Theory. It probably still is. Huffman's original work spawned numerous minor variations, and it dominated the coding world till the early 1980s. As the cost of CPU cycles went down, new possibilities for more efficient coding techniques emerged. One in particular, arithmetic coding, is a viable successor to Huffman coding.

Arithmetic coding is somewhat more complicated in both concept and implementation than standard variable-width codes. It does not produce a single code for each symbol. Instead, it produces a code for an entire message. Each symbol added to the message incrementally modifies the output code. This is an improvement because the net effect of each input symbol on the output code can be a fractional number of bits instead of an integral number. So if the entropy for character 'e' is 2.5 bits, it is possible to add exactly 2.5 bits to the output code. An example of why this can be more effective is shown in the following table, the analysis of an imaginary message. In it, Huffman coding would yield a total message length of 89 bits, but arithmetic coding would approach the true

information content of the message, or 83.56 bits. The difference in the two messages works out to approximately 6 percent.

Here are some sample message probabilities:

Symbol	No of Occurences	Information Content	Huffman Code Count	Total Bits Huffman Coding	Total Bits Arithmetic Coding
E	20	1.26	1	20	25.2
A	20	1.26	2	40	25.2
X	3	4.00	3	9	12.0
Y	3	4.00	4	12	12.0
Z	2	4.58	4	8	9.16

Table 2

The problem with Huffman coding in the above message is that it can't create codes with the exact information content required. In most cases it is a little above or a little below, leading to deviations from the optimum. But arithmetic coding gets to within a fraction of a percent of the actual information content, resulting in more accurate coding. Arithmetic coding requires more CPU power than was available until recently. Even now it will generally suffer from a significant speed disadvantage when compared to older coding methods. But the gains from switching to this method are significant enough to ensure that arithmetic coding will be the coding method of choice when the cost of storing or sending information is high enough.

3.1.2 Arithmetic Coding

Arithmetic coding is a technique for coding that allows the information from the messages in a message sequence to be combined to share the same bits. The technique allows the total number of bits sent to asymptotically approach the sum of the self information of the individual messages (recall that the self information of a message is defined as log2 pi).

To see the significance of this, consider sending a thousand messages each having probability .999. Using a Huffman code, each message has to take at least 1 bit, requiring 1000 bits to be sent. On the other hand the self information of each message is log21pi= .00144 bits, so the sum of this self-information over 1000 messages is only 1.4 bits. It turns out that arithmetic coding will send all the messages using only 3 bits, a factor of hundreds fewer than a Huffman coder. Of course this is an extreme case, and when all the probabilities are small, the gain will be less significant.

Arithmetic coders are therefore most useful when there are large probabilities in the probability distribution The main idea of arithmetic coding is to represent each possible sequence of n messages by a separate interval on the number line between 0 and 1, *e.g.* the interval from .2 to .5. For a sequence of messages with probabilities p1, . . . , pn, the algorithm will assign the sequence to an interval of size Qn i=1 pi, by starting with an interval of size 1 (from 0 to 1) and narrowing the interval by a factor of pi on each message i. We can bound the number of bits required to uniquely identify an interval of size s, and use this to relate the length of the representation to the self information of the messages. In the following discussion we assume the decoder knows when a message sequence is complete either by knowing the length of the message sequence or by including a special end-of-file message. This was also implicitly assumed when sending a sequence of messages with Huffman codes since the decoder still needs to know when a message sequence is over.

3.2 Applications of Probability Coding

3.2.1 Run Length Encoding

Probably the simplest coding scheme that takes advantage of the context is run-length coding. Although there are many variants, the basic idea is to identify strings of adjacent messages of equal value and replace them with a single occurrence along with a count.

For example, the message sequence acccbbaaabb could be transformed to (a,1), (c,3), (b,2), (a,3), (b,2). Once transformed, a probability coder (*e.g.*, Huffman coder) can be used to code both the message values and the counts. It is typically important to probability code the run-lengths since short lengths (*e.g.*, 1 and 2) are likely to be much more common than long lengths (*e.g.*, 1356). An example of a real-world use of run-length coding is for the ITU-T T4 (Group 3) standard for Facsimile (fax) machines1. At the time of writing (1999), this was the standard for all home and business fax machines used over regular phone lines.

Fax machines transmit black-and-white images. Each pixel is called a *pel* and the horizontal resolution is fixed at 8.05 pels/mm. The vertical resolution varies depending on the mode. The T4 standard uses run-length encoding to code each sequence of black and white pixels. Since there are only two message values black and white, only the run-lengths need to be transmitted. The T4 standard specifies the start color by placing a dummy white pixel at the front of each row so that the first run is always assumed to be a white run. For example, the sequence bbbbwwbbbbb would be transmitted as 1,4,2,5. The

run-length	white codeword	black codeword
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
..		
20	0001000	00001101000
..		
64+	11011	0000001111
128+	10010	000011001000

Table 3: ITU-T T4 Group 3 Run-length Huffman codes.

Fig. 3.1 Group 3 Run Length Huffman Codes

T4 standard uses static Huffman codes to encode the run-lengths, and uses a separate codes for the black and white pixels. To account for runs of more than 64, it has separate codes to specify multiples of 64. For example, a length of 150, would consist of the code for 128 followed by the code for 22. These Huffman codes are based on the probability of each run-length measured over a large number of documents. The full T4 standard also allows for coding based on the previous line.

3.2.2 Context Coding

Another simple coding scheme that takes advantage of the context is move-to-front coding. This is used as a sub-step in several other algorithms including the Burrows-Wheeler algorithm discussed later. The idea of move-to-front coding is to preprocess the message sequence by converting it into a sequence of integers, which hopefully is biased

toward integers with low values. The algorithm then uses some form of probability coding to code these values. In practice the conversion and coding are interleaved, but we will describe them as separate passes. The algorithm assumes that each message comes from the same alphabet, and starts with a total order on the alphabet (*e.g.*, [a, b, c, d, . ..]). For each message, the first pass of the algorithm outputs the position of the character in the current order of the alphabet, and then updates the order so that the character is at the head. For example, coding the character c with an order [a, b, c, d, . . .] would output a 3 and change the order to [c, a, b, d, . . .]. This is repeated for the full message sequence. The second pass converts the sequence of integers into a bit sequence using Huffman or Arithmetic coding.

The hope is that equal characters often appear close to each other in the message sequence so that the integers will be biased to have low values. This will give a skewed probability distribution and good compression.

3.2.3 Context Coding PPM

The main idea of PPM (Prediction by Partial Matching) is to take advantage of the previous K characters to generate a conditional probability of the current character. The simplest way to do this would be to keep a dictionary for every possible string s of k characters, and for each string have counts for every character x that follows s. The conditional probability of x in the context s is then C(x|s)/C(s), where C(x|s) is the number of times x follows s and C(s) is the number of times s appears. The probability distributions can then be used by a Huffman or Arithmetic coder to generate a bit sequence. For example, we might have a dictionary with qu appearing 100 times and e appearing 45 times after qu. The conditional probability of the e is then .45 and the coder should use about 1 bit to encode it. Note that the probability distribution will change from character to character since each context has its own distribution. In terms of decoding, as long as the context precedes the character being coded, the decoder will know the context and therefore know which probability distribution to use. Because the probabilities tend to be high, arithmetic codes work much better than Huffman codes for this approach.

Order 0		Order 1		Order 2	
Context	Counts	Context	Counts	Context	Counts
empty	a = 4	a	c = 3	ac	b = 1
	b = 2				c = 2
	c = 5	b	a = 2		
				ba	c = 1
		c	a = 1		
			b = 2	ca	a = 1
			c = 2		
				cb	a = 2
				cc	a = 1
					b = 1

Figure 10: An example of the PPM table for $k = 2$ on the string accbaccacba.

Figure 3.2

The PPM algorithm has a clever way to deal with the case when a context has not been seen before, and is based on the idea of partial matching. The algorithm builds the dictionary on the fly starting with an empty dictionary, and every time the algorithm comes across a string it has not seen before it tries to match a string of one shorter length. This is repeated for shorter and shorter lengths until a match is found. For each length 0, 1, . . . , k the algorithm keeps statistics of patterns it has seen before and counts of the following characters. In practice this can all be implemented in a single trie. In the case of the length-0 contexts the counts are just counts of each character seen assuming no context.

An example table is given in Figure 10 for a string accbaccacba. Now consider following this string with a c. Since the algorithm has the context ba followed by c in its dictionary, it can output the c based on its probability in this context. Although we might think the probability should be 1, since c is the only character that has ever followed ba, we need to give some probability of no match, which we will call the "escape" probability. We will get back to how this probability is set shortly. If instead of c the next character to code is an a, then the algorithm does not find a match for a length 2 context so it looks for a match of length 1, in this case the context is the previous a. Since a has never followed by another a, the algorithm still does not find a match, and looks for a match with a zero length context. In this case it finds the a and uses the appropriate probability for a (4/11). What if the algorithm needs to code a d? In this case the algorithm does not even find the character in the zero-length context, so it assigns the character a probability assuming all

Order 0		Order 1		Order 2	
Context	Counts	Context	Counts	Context	Counts
empty	a = 4	a	c = 3	ac	b = 1
	b = 2		$ = 1		c = 2
	c = 5				$ = 2
	$ = 3	b	a = 2		
			$ = 1	ba	c = 1
					$ = 1
		c	a = 1		
			b = 2	ca	c = 1
			c = 2		$ = 1
			$ = 3		
				cb	a = 2
					$ = 1
				cc	a = 1
					b = 1
					$ = 2

Figure 11: An example of the PPMC table for $k = 2$ on the string accbaccacba. This assumes the "virtual" count of each escape symbol ($) is the number of different characters that have appeared in the context.

Fig .3.3

Shanon Fano coding

The Shannon-Fano technique has as an advantage its simplicity. The code is constructed as follows: the source messages a I and their probabilities p(a I) are listed in order of non- increasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the _rst group receives 0 as the _rst digit of its codeword; the messages in the second half have codewords beginning with. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one message. Clearly the Shannon-Fano algorithm yields a minimal pre_x code. a

a_1	1/2	0	$step1$
a_2	1/4	10	$step2$
a_3	1/8	110	$step3$
a_4	1/16	1110	$step4$
a_5	1/32	11110	$step5$
a_6	1/32	11111	

Figure 3.1 A Shannon-Fano Code.

Fig .3.4 Shannon Fano Code

shows the application of the method to a particularly simple probability distribution. The length of each codeword is equal to lg p(ai). This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some codewords may be of length lg p(ai)+ 1. The Shannon-Fano algorithm yields an average codeword length S which satis_es H _ S _ H + 1. In Figure 3.2, the Shannon-Fano code for ensemble EXAMPLE is given. As is often the case, the average codeword length is the same as that achieved by the Hu_man code (see Figure 1.3). That the Shannon-Fano algorithm is not guaranteed to produce an optimal code is demonstrated by the following set of probabilities: f:35; :17; :17; :16; :15; g.

The Shannon-Fano Algorithm

A Shannon-Fano tree is built according to a specification designed to define an effective code table. The actual algorithm is simple:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
2. Sort the lists of symbols according to frequency, with the most frequently occuring symbols at the top and the least common at the bottom.

3. Divide the list into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.

4. The upper half of the list is assigned the binary digit 0, and the lower half is assigned the digit 1. This means that the codes for the symbols in the first half will all start with 0, and the codes in the second half will all start with 1.

5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree

CHAPTER 4: HUFFMAN AND ARITHMETIC CODING

4.1 Huffman Algorithm

Huffman coding shares most characteristics of Shannon-Fano coding. It creates variable length codes that are an integral number of bits. Symbols with higher probabilities get shorter codes. Huffman codes have the unique prefix attribute, which means they can be correctly decoded despite being variable length. Decoding a stream of Huffman codes is generally done by following a binary decoder tree.

Building the Huffman decoding tree is done using a completely different algorithm from that of the Shannon-Fano method. The Shannon-Fano tree is built from the top down, starting by assigning the most significant bits to each code and working down the tree until finished. Huffman codes are built from the bottom up, starting with the leaves of the tree and working progressively closer to the root.

The procedure for building the tree is simple and elegant. The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight, which is simply the frequency or probability of the symbol's appearance.
The tree is then built with the following steps:

• The two free nodes with the lowest weights are located.

• A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.

• The parent node is added to the list of free nodes, and the two child nodes are removed from the list.

• One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.

• The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.

This algorithm can be applied to the symbols used in the previous example. The five symbols in our message are laid out, along with their frequencies, as shown:

15	7	6	6	5
A	B	C	D	E

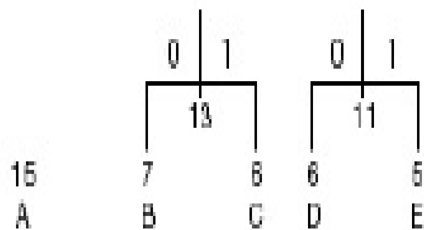
These five nodes are going to end up as the leaves of the decoding tree. When the process first starts, they make up the entire list of free nodes.

The first pass through the tree identifies the two free nodes with the lowest weights: D and E, with weights of 6 and 5. (The tie between C and D was broken arbitrarily. While the way that ties are broken affects the final value of the codes, it will not affect the compression ratio achieved.) These two nodes are joined to a parent node, which is assigned a weight of 11. Nodes D and E are then removed from the free list.

Once this step is complete, we know what the least significant bits in the codes for D and E are going to be. D is assigned to the 0 branch of the parent node, and E is assigned to the 1 branch. These two bits will be the LSBs of the resulting codes.

On the next pass through the list of free nodes, the B and C nodes are picked as the two with the lowest weight. These are then attached to a new parent node. The parent node is assigned a weight of 13, and B and C are removed from the free node list. At this point, the tree looks like that shown in Figure.

Fig 4.1 The Huffman Tree After Two Passes



On the next pass :

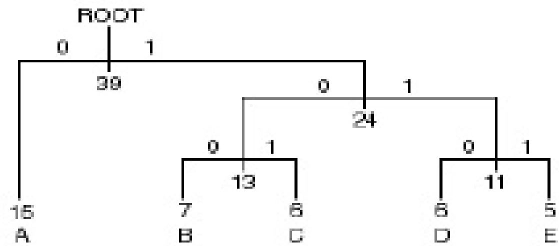


Fig 4.2

To determine the code for a given symbol, we have to walk from the leaf node to the root of the Huffman tree, accumulating new bits as we pass through each parent node. Unfortunately, the bits are returned to us in the reverse order that we want them, which means we have to push the bits onto a stack, then pop them off to generate the code. This strategy gives our message the code structure shown in the following table.

The Huffman Code Table

A	0
B	100
C	101
D	110
E	111

Fig .4.3

Since no code is a prefix to another code, Huffman codes can be unambiguously decoded as they arrive in a stream. The symbol with the highest probability, A, has been assigned the fewest bits, and the symbol with the lowest probability, E, has been assigned the most bits.

4.2 Arithmetic Coding For Data Compression

Arithmetic encoding is the most powerful compression techniques. This converts the entire input data into a single floating point number. A floating point number is similar to a number with a decimal point, like 4.5 instead of **41/2**. However, in arithmetic coding we are not dealing with decimal number so we call it a floating point instead of decimal point [4].

The idea behind arithmetic coding is to have a probability line, 0-1, and assign to every symbol a range in this line based on its probability, the higher the probability, the higher range which assigns to it. Once we have defined the ranges and the probability line, start to encode symbols, every symbol defines where the output floating point number lands.

Let say input string is
"Baca"

Symbol	Probability	Range
A	2	[0.0 , 0.5)
B	1	[0.5 , 0.75)
C	1	[0.7.5 , 1.0)

- Low = 0
- High = 1
- Loop. For all the symbols.
    - Range = high - low
    - High = low + range *high_range of the symbol being coded
    - Low = low + range *low_range of the symbol being coded

Where:

- Range, keeps track of where the next range should be.
- High and low, specify the output number.

Symbol	Range	Low value	High value
		0	1
B	1	0.5	0.75
A	0.25	0.5	0.625
C	0.125	0.59375	0.625
A	0.03125	0.59375	0.609375

The output number will be 0.59375. The way of decoding is first to see where the number lands, output the corresponding symbol, and then extract the range of this symbol from the floating point number. The algorithm for extracting the ranges is:

- Loop. For all the symbols.
  - Range = high_range of the symbol - low_range of the symbol
  - Number = number - low_range of the symbol
  - Number = number / range

Given this encoding scheme, it is relatively easy to see how the decoding process operates. Find the first symbol in the message by seeing which symbol owns the space our encoded message falls in. Since .2572167752(assuming this to be encoded form of

considered data) falls between .2 and .3, the first character must be B. Then remove B from the encoded number. Since we know the low and high ranges of B, remove their effects by reversing the process that put them in. First, subtract the low value of B, giving

.0572167752. Then divide by the width of the range of B, or .1. This gives a value of .572167752. Then calculate where that lands, which is in the range of the next letter, I. The algorithm for decoding the incoming number is shown
next:

```
number = input_code();
for ( ; ; ) {
symbol = find_symbol_straddling_this_range( number );
putc( symbol );
range = high_range( symbol ) - low_range( symbol );
number = number - low_range( symbol );
number = number / range;
```

Measuring Compression Performance

Performance measure is use to find which technique is good according to some criteria. Depending on the nature of application there are various criteria to measure the performance of compression algorithm. When measuring the performance the main thing to be considered is space efficiency [5]. and the time efficiency is another factor. Since the compression behavior depends on the redundancy of symbols in the source file, it is difficult to measure performance of compression algorithm in general. The performance of data compression depends on the type of data and structure of input source. The compression behavior depends on the category of the compression algorithm: lossy or lossless. Following are some measurements use to calculate the performances of lossless algorithms.

Compression ratio: compression ratio is the ratio between size of compressed file and the size of source file.

$$Compression\ ratio = \frac{Size\ after\ compression}{Size\ before\ compression}$$

Compression factor: compression factor is the inverse of compression ratio. That is the ratio between the size of source file and the size of the compressed file.

$$Compression\ factor = \frac{Size\ before\ compression}{Size\ after\ compression}$$

Saving Percentage:

$$saving\ percentage = \frac{size\ before\ compression - size\ after\ compression}{size\ before\ compression}\%$$

5. Context Based Modeling for Data Compression

5.1 Basics of Context based Modeling

Adaptive context modeling has emerged as one of the most promising new approaches to compressing text. A finite-context model is a probabilistic model that uses the context in which input symbols occur (generally a few preceding characters) to determine the number of bits used to code these symbols. We provide an introduction to context modeling and recent research results that incorporate the concept of context modeling into practical data compression algorithms.

5.1.1

A finite-context model uses the context provided by characters already seen to determine the encoding of the current character. The idea of a context consisting of a few previous characters is very reasonable when the data being compressed is natural language. We all know that the character following q in an English text is all but guaranteed to be u and that given the context now is the time for all good men to come to the aid of, the phrase their country is bound to follow. One would expect that using knowledge of this type would result in more accurate modeling of the information source. Although the technique of context modeling was developed and is clearly appropriate for compressing natural language, context models provide very good compression over a wide range of _le types.

We say that a context model predicts successive characters taking into account the context provided by characters already seen. What is meant by predict here is that the frequency values used in encoding the current character are determined by its context. The frequency distribution used to encode a character determines the number of bits it contributes to the compressed representation.

When character x occurs in context c and the model for context c does not include a frequency for x we say that context c fa predict x. A context model may use a mixed number of previous characters in its predictions or may be a blended model, incorporating predictions based on contexts of several lengths. A model that always uses i previous characters to predict the current character is a pure order-i context model. When i = 0, no context is used and the text is simply coded one character at a time. When i = 1, the previous character is used in encoding the current character; when i = 2, the previous two characters are used, and so on. A blended model may use the previous three characters, the previous two characters when the three- character context fails to predict, and one predecessor if both the order-3 and order-2 contexts fail. A blended model is composed of two or more sub models. An order-i context model consists of a frequency distribution for each i-character 3
sequence occurring in the input stream. In the order-1 case, this means that the frequency distribution for context q will give a very high value to u and very little weight to any other letter, while the distribution for context t will have high frequencies for a, e , i, o, u, and h among others and very little weight for letters like q, n and g.
A blended model is fully blended if it contains sub models for the maximum-length context and all lower-order contexts. That is, a fully-blended order-3 context model bases its predictions on models of orders 3, 2, 1, 0, and □1 (the model of order □1 consists of a frequency distribution that weights all characters equally). A partially-blended model uses some, but not all, of the lower-order contexts.

5.1.2

A context model is generally combined with arithmetic coding to form a data compression system. The model provides a frequency distribution for each context (each character in the order-1 case and each pair of characters in the order-2 case). Each frequency distribution forms the basis of an arithmetic code and these are used to map events into code bits. Hu_man coding is not appropriate for use with adaptive context models for the reasons given above.

5.2 Methods Of Blending

Blending is desirable and essentially unavoidable in an adaptive setting where the model is built from scratch as encoding proceeds. When the first character of a _le is read, the model has no history on which to base predictions. Larger contexts become more meaningful as compression proceeds. The general mechanism of blending, weighted blending, assigns a probability to a character by weighting probabilities (or, more accurately, frequencies) provided by the various sub models and computing the weighted sum of these probabilities.

This method of blending is too slow to be practical and has the additional disadvantage that there is no theoretical basis for assigning weights to the models of various orders. In a simpler and more practical blended order-I model, the number of bits used to code character c is dictated by the preceding i characters if c has occurred in this particular context before. In this case, only the order-i frequency distribution is used. Otherwise, models of lower orders are consulted until one of them supplies a prediction. When the context of order I fails to predict the current character, the encoder emits an escape code, a signal to the decoder that the model of lower order is being consulted. Some lowest-order model must be guaranteed to supply a prediction for every character in the input alphabet.

The frequencies used by the arithmetic coder may be computed in a number of ways. One of the more straightforward methods is to assign to character x in context c the frequency f where f is the number of times that context c has been used to predict character x. Alternatively, f may represent the number of times that x has occurred in context c. An implementation may also require x to occur in context c some minimal number of times before it allocates frequency to the event.

5.2.2 Escape Strategy

In order for the encoder to transmit the escape code, each frequency distribution in the blended model must have some frequency allocated to escape.

A simple strategy is to treat the escape event as if it were an additional symbol in the input alphabet. Like any other character, the frequency of the escape event is the number of times it occurs. Other strategies involve relating the frequency of the escape code to the total frequency of the context and the number of different characters occurring in the context. On one hand, as the number of different characters increases, the probability of prediction increases and the use of the escape code becomes less likely. On the other hand, if a context has occurred frequently and predicted the same character (or small number of characters) every time, the appearance of a new character (and the need to escape) would seem unlikely. There is no theoretical basis for selecting one of these escape strategies over another. Fortunately, empirical experiments indicate that compression performance is largely insensitive to the selection of escape strategy.

5.2.3 Exclusion Principle

The blending strategy described above has the effect of excluding lower-order predictions when a character occurs in a higher-order model. However, it does not exclude as much lower-order information as it might. For example, when character x occurs in context abc for the first time the order-2context bc is consulted. If character y has occurred in context abc it can be excluded from the order-2 prediction. That is, the fact that we escape from the order-3 context abc informs the decoder that the character being encoded is not y. Thus the bc model need not assign any frequency to y in making this prediction. By excluding y from the order-2 prediction x may be predicted more accurately. Excluding characters predicted by higher-order models can double execution time. The gain in compression performance is on the order of 5%, which hardly justifies the increased execution time [BCW90]. Another type of exclusion that is much simpler and has the effect of decreasing execution time is update exclusion.

Update exclusion means updating only those models that contribute to the current prediction. Thus if, in the above example, context bc 6 predicts x, only the order-3 model for abc and the order-2 model for bc will be updated. The models of lower order remain unchanged.

5.2.4 Memory Limitations

We call an order-i context model complete if for every character x occur- ring in context c, the model includes a frequency distribution for c that contains a count for x. That is, the model retains all that it has learned. Complete context models of even order 3 are rare since the space required to store all of the context information gleaned from a large _le is prohibitive. There are two obvious ways to impose a memory limit on a finite context model. The first is to monitor its size and freeze the model when the size reaches some maximum.

When the model is frozen, it can no longer represent characters occurring in novel contexts, but we can continue to update the frequency values already stored in the model. The second approach is to rebuild the model rather than freeze it. The model can be rebuilt from scratch or from a buffer representing recent history. The use of the buffer may lessen the degradation in compression performance due to rebuilding. On the other hand, the memory set aside for the buffer causes rebuilding to occur earlier. A third approach, which is not strictly a solution to the problem of limited memory, is to monitor compression performance as well as the size of the data structure. Rebuilding when compression begins to degrade may be more opportune than waiting until it becomes necessary. We will say more about the data structures used to represent context models in later sections. The representation of the model clearly impacts the amount of information it can contain and the ease with which it can be consulted and updated.

5.3 Context Modeling with order 2

The algorithm we describe in this section employs a blended order-2 context model. It can be implemented so as to provide compression performance that is better than that provided by compress and much better than other existing algorithms, using far less space than either of these systems (10 percent as much memory as compress). In Section 5.3 we describe the method of blending we employ. In Section 5.5 we describe the frequency distributions maintained by our algorithm, and Section 5.6 presents our escape strategy. Consider the use of dynamic memory to improve the memory requirement. Later on we show that hashing is a much more effective means to this end. We present some experimental data on the performance of our order-2-and-0 methods.

5.3.1 Blending Strategy

One of the ways in which we conserve on both memory and execution time is by blending only models of orders 2 and 0, rather than orders 2, 1, 0, and 1. Thus we refer to our model as an order-2-and-0 context model. We have experimented with order-2-and-1 and order-2-1-and-0 models. The order-2-and-1 model did not provide satisfactory compression performance and the order-2-1-and-0 model produces compression results that are very close to those of our order-2-and-0 algorithm. The order-2-and-0 model allows faster encoding and decoding since it consults at most two contexts per character. We provide more details on the models of orders 2 and 0

5.3.2 Self Organizing List

In our order-2-and-0 model, we maintain a self-organizing list of size s for each two-character context (s is a parameter of the algorithm). We encode z when it occurs in context xy by event k if z is in position k of list xy. When z does not appear on list xy we encode z itself using the order-0 model. Encoding entails mapping the event (k or z) to a frequency and employing an arithmetic coder. To complete the description of the algorithm, we need to specify a list organizing strategy and the method of maintaining frequencies. The frequency count list organizing strategy is inappropriate because

large number of counts required. We employ the transpose strategy because it provides faster update than move-to-front

When character z occurs in context xy and z appears on the context list for xy, the list is updated using the transpose strategy. If z does not appear on the xy list, it is added. If the size of list xy is less than s (size < s), the item currently in position size moves into position size + 1 and z is stored in position size. If the list is full when z is to be added, z will replace the last item. An obvious disadvantage to xing the size of the order-2 context lists is that the lists are likely to be too short for some contexts and too long for others. When an order-2 list (say, list xy) contains s items and a new character z occurs in context xy, we delete the bottom item (call it t) from the list and add z. Context xy no longer predicts t. This does not affects the correctness of our algorithm. When t occurs again in context xy it will be predicted by the order- 0 model. The fact that encoder and decoder maintain identical models ensures correctness. . In addition, the rationale behind the use of self-organizing lists is that we expect to have the s most common successors on the list at any point in time. As characteristics of the le change, successors that become common replace those that fall into disuse.

5.3.3 Frequency Distribution

In order to conserve memory we do not use a frequency distribution for each context. Instead, we maintain a frequency value for each feasible event. Since there are s + 1 values of k (the s list positions and the escape code) and n+ 1 values for z (the n characters of the alphabet and an end-of- le character), the number of feasible events is s+n+ 2. We can maintain the frequency values either as a single distribution or as two distributions, an order-2 distribution to which list positions are mapped and an order-0 distribution to which characters are mapped. Our experiments indicate that the two-distribution model is slightly superior. When z occurs in context xy we use the two frequency distributions in the following way: if list xy exists and z occupies position k, we encode k using the order-2 distribution. If list xy exists but does not contain z, we encode an escape code (using the order-2 distribution) as a signal to the decoder that an order-0 prediction (and the order-0 frequency distribution) is to be used, and then encode the character z.

When list xy has not been created yet, the decoder knows this and no escape code is necessary; we simply encode z using the order-0 distribution. Our limited use of frequency distributions is similar to that of algorithm ADSM.

5.3.4 Escape Strategy

We adopt the strategy of treating the escape event as if it were an additional list position. Given this decision, there are two reasonable choices for the value of escape. One choice is to use the value s + 1, as it will never represent a list position. The second choice is to use the value size + 1, where size is the current size of list xy (and ranges from 1 to s). In the rst case, the escape code is the same for every context and all of the counts for escape accrue to a single frequency value while in the second case, the value of escape depends on the context and generates counts that accrue to multiple frequency values. The two escape strategies produce similar compression results. The algorithm we describe here uses the rst alternative.

We apply update exclusion in dealing with both lists and frequency distributions. That is, a list or frequency distribution is updated when it is used. Thus, when list xy exists, both the list and the frequency distribution are updated after being used to encode either a list position or an escape. The order-0 distribution is used and updated each time context xy fails to predict.

5.4 Coding the Model

The models of order 3, 1, and 0 are used to form a prediction of the current character in much the same way as we used them in the order-2-and-0 algorithm. We encode character z occurring in context wxy by event k if z occurs in position k of the list for context wxy. If z does not appear on wxy's list, we code an escape and consult the list for the order-1 context y. An order- 3 frequency distribution is used to code either k or escape. When the order-1 model is consulted, an order-1 frequency distribution is used to code either j (if z occurs in position j of list y) or escape. When neither context wxy nor context y predicts z we follow the two escape codes with an order-0 prediction (i.e., we code the character itself). If the list for context wxy (likewise context y) is empty, the corresponding escape code is not necessary

The escape codes are represented as list positions s3 + 1 and s1 + 1, respectively. As in our order-2-and-0 algorithm we apply update exclusion so that lists and frequency distributions are updated only when they contribute to the prediction of the current character, z. If list wxy exists, we update it using the transpose heuristic. If no wxy list exists one will be created. If context wxy does not predict z, then the y list is updated using the transpose method. If list y is not used in the prediction, it is not updated. When list wxy exists, the wxy frequency distribution is updated after it is used to encode either a list position or an escape. When context wxy does not predict and list y exists, the y frequency distribution is updated. The order-0 frequency distribution is updated whenever the character itself is coded.

6. Prediction by Partial Matching

6.1 Introduction

The "Prediction by Partial Match" method, originally developed by Cleary and Witten, with extension and an implementation by A. Moffat, is capable of very good compression on a wide variety of source data. The adaptive nature of the scheme and the flexibility afforded by arithmetic coding mean that an effective compression model will be built for any input file that is reasonably homogeneous.

The method is based on an encoder that maintains a statistical model of the text. The encoder inputs the next symbol S, assigns it a probability P, and sends S to an arithmetic encoder, to be encoded with probability P. The statistical model counts the number of times each symbol has occurred in the past and assigns the symbol a probability based on that. In a context based statistical model, the idea is to assign a probability to symbol S depending not just on the frequency of the symbol but on the contexts in which it has occurred so far. A static context based modeler always uses the same probabilities and offers the advantage of being simple and producing good result on average. However it may lead to considerable expansion in the case when certain input stream is statistically very different from the data originally used to prepare the table and when it encounters zero probabilities. The arithmetic encoder requires all symbols to have non-zero probabilities. Another reason why a symbol must have non-zero probability is that its entropy depends on log2 P, which is undefined for P= 0.

An adaptive context-based modeler updates its probability table all the time as more data is being input, which adapts the probabilities to the particular data being compressed. Such a model is slower and more complex but produces better compression. An order N adaptive context based modeler reads the next symbol S from the input stream and considers the N symbols preceding S the current order N context C of S. The model then estimates the probability P that S appears in the input data following the particular context C. Theoretically, the larger the N, the better the probability estimate. However, a larger context is difficult to manage.

A very long context retains information about the nature of old data. Experience shows that large data files contain different distributions of symbols in different parts. Better compression can therefore be achieved if the model assigns less importance to information collected from old data and more weight to fresh, recent data. Such an effect is achieved by a short context.

The central idea of PPM is to use this knowledge. It uses an adaptive model based on a variable length context. At each coding step the longest previously encountered context is used to predict the next character. If the symbol is novel to the context, an escape code is transmitted and the context shortened by dropping one symbol. The Process continues until the symbol is successfully transmitted. If the current symbol is novel even to the zero order context then a final escape is transmitted, and the symbol will be transmitted as an 8 bit code. The adaptive model then adds the current symbol to all applicable contexts.

Based on the different methods of assigning probabilities to escape symbol following variants of PPM are presented.

- PPMA
- PPMB
- PPMC

In PPMA, a group of symbols has total frequencies n (excluding escape symbol).

The escape symbol is assigned a probability = 1/(n+1). This is equivalent to always assigning it a count of 1. Other members are still assigned their original probabilities (x/n).

In PPMB, a symbol S following context C is assigned a probability only after S has been seen twice in context C. This is done by subtracting 1 from the frequency counts. The subtracted 1's are added to the count of the Escape Symbol.

The way Escape probabilities are assigned in the three methods is based on intuition and experience, not on any underlying theory. Experience with the three variants of PPM shows that none is preferable. They produce compression ratios that normally differ by just a few percent. This shows that the basic PPM algorithm is robust, and does not depend on the precise way of assigning escape probabilities.

6.2 Prediction with Partial Matching Algorithm for Data Compression

The PPM data compression scheme has set the performance standard in lossless compression of text throughout the past decade. The original algorithm was first published in 1984 by Cleary and Witten, and a series of improvements was de- scribed by Moffat, culminating in a careful implementation, called PPMC, which has become the benchmark version. This still achieves results superior to virtually all other compression methods, despite many attempts to better it. Other methods
such as those based on Ziv-Lempel coding are more commonly used in practice, but their attractiveness lies in their relative speed rather than any superiority in compression indeed, their compression performance generally falls distinctly below that of PPM in practical benchmark tests.

Prediction by partial matching, or PPM, is a finite-context statistical modeling technique that can be viewed as blending together several mixed-order context models to predict the next character in the input sequence. Prediction probabilities for each context in the model are calculated from frequency counts which are updated adaptively; and the symbol that actually occurs is encoded relative to its predicted distribution using arithmetic coding. The maximum context length is a mixed constant, and it has been found that increasing it beyond about six or so does not generally improve compression.

The present paper describes a new algorithm, PPM*, which exploits contexts of unbounded length. It reliably achieves compression superior to PPMC, although our current implementation which we have not yet attempted to optimize uses considerably greater computational resources (both time and space). The next section describes the basic PPM compression scheme. Following that we motivate the use of contexts of unbounded length, introduce the new method, and show how it can be implemented using a trie data structure.

6.2.1 PPMC

The basic idea of PPM is to use the last few characters in the input stream to predict the upcoming one. Models that condition their predictions on a few immediately preceding symbols are called finite-context" models of order k, where k is the number of preceding symbols used. PPM employs a suite of mixed-order context models with different values of k, from 0 up to some pre-determined maximum, to predict upcoming characters.

For each model, a note is kept of all characters that have followed every length-k subsequence observed so far in the input, and the number of times that each has occurred. Prediction probabilities are calculated from these counts. The probabilities associated with each character that has followed the last k characters in the past are used to predict the upcoming character. Thus from each model, a separate predicted probability distribution is obtained.

These distributions are effectively combined into a single one, and arithmetic coding is used to encode the character that actually occurs, relative to that distribution.

The combination is achieved through the use of escape probabilities. Recall that each model has a different value of k. The model with the largest k is, by default, the one used for coding. However, if a novel character is encountered in this context, which means that the context cannot be used for encoding it, an \escape" symbol is transmitted to signal the decoder to switch to the model with the next smaller value of k. The process continues until a model is reached in which the character is not novel, at which point it is encoded with respect to the distribution predicted by that model. To ensure that the process terminates, a model is assumed to be present below the lowest level, containing all characters in the coding alphabet. This mechanism effectively blends the different order models together in a proportion that depends on the values actually used for escape probabilities.

Order $k=2$ Predictions	c	p	Order $k=1$ Predictions	c	p	Order $k=0$ Predictions	c	p	Order $k=-1$ Predictions	c	p
ab $\rightarrow$ r	2	$\frac{2}{3}$	a $\rightarrow$ b	2	$\frac{2}{7}$	$\rightarrow$ a	5	$\frac{5}{16}$	$\rightarrow$ A	1	$\frac{1}{\lvert A\rvert}$
$\rightarrow$ Esc	1	$\frac{1}{3}$	$\rightarrow$ c	1	$\frac{1}{7}$	$\rightarrow$ b	2	$\frac{2}{16}$			
			$\rightarrow$ d	1	$\frac{1}{7}$	$\rightarrow$ c	1	$\frac{1}{16}$			
ac $\rightarrow$ a	1	$\frac{1}{2}$	$\rightarrow$ Esc	3	$\frac{3}{7}$	$\rightarrow$ d	1	$\frac{1}{16}$			
$\rightarrow$ Esc	1	$\frac{1}{2}$				$\rightarrow$ r	2	$\frac{2}{16}$			
			b $\rightarrow$ r	2	$\frac{2}{3}$	$\rightarrow$ Esc	5	$\frac{5}{16}$			
ad $\rightarrow$ a	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{3}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
			c $\rightarrow$ a	1	$\frac{1}{2}$						
br $\rightarrow$ a	2	$\frac{2}{3}$	$\rightarrow$ Esc	1	$\frac{1}{2}$						
$\rightarrow$ Esc	1	$\frac{1}{3}$									
			d $\rightarrow$ a	1	$\frac{1}{2}$						
ca $\rightarrow$ d	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{2}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
			r $\rightarrow$ a	2	$\frac{1}{3}$						
da $\rightarrow$ b	1	$\frac{1}{2}$	$\rightarrow$ Esc	1	$\frac{1}{3}$						
$\rightarrow$ Esc	1	$\frac{1}{2}$									
ra $\rightarrow$ c	1	$\frac{1}{2}$									
$\rightarrow$ Esc	1	$\frac{1}{2}$									

Table 6.1 Table showing processing of string abracadabra up to order 2

As an illustration of the operation of PPM, Table shows the state of the four models with k = 2, 1, 0, and-1 after the input string abracadabra has been processed. For each model, all previously-occurring contexts are shown with their associated predictions, along with occurrence counts c and the probabilities p that are calculated from them. By convention, k = -1 designates the bottom-level model that predicts all characters equally; it gives them each probability jAj where A is the alphabet used.

Some policy must be adopted for choosing the probabilities to be associated with the escape events. There is no sound theoretical basis for any particular choice in the absence of some a priori assumption on the nature of the symbol source; some

Alternatives are evaluated in. The method used in the example, commonly called Method C gives a count to the escape event equal to the number of different symbols that have been

seen in the context so far [6]; thus, for example, in the order- 0 column of Table 1 the esc symbol receives a count of 5 because five different symbols have been seen in that context.

character	probabilities encoded (without exclusions)	probabilities encoded (with exclusions)	code space occupied
c	$\frac{1}{2}$	$\frac{1}{2}$	$-\log_2 \frac{1}{2} = 1$ bit
d	$\frac{1}{2}, \frac{1}{7}$	$\frac{1}{2}, \frac{1}{6}$	$-\log_2(\frac{1}{2} \cdot \frac{1}{6}) = 3.6$ bits
t	$\frac{1}{2}, \frac{3}{7}, \frac{5}{16}, \frac{1}{\lvert A \rvert}$	$\frac{1}{2}, \frac{3}{6}, \frac{5}{12}, \frac{1}{\lvert A \rvert - 5}$	$-\log_2(\frac{1}{2} \cdot \frac{3}{6} \cdot \frac{5}{12} \cdot \frac{1}{251}) = 11.2$ bits

Table 6.2 Encoding for three sample characters using the data from table 6.1

Sample encodings using these models are shown in Table 6.2. As noted above, prediction proceeds from the highest-order model (k = 2). If the context successfully predicts the next character in the input sequence, the associated probability p is used to encode it. For example, if c followed the string abracadabra, the prediction ra->c would be used to encode it with a probability of 1/2, that is, in one bit.

Suppose instead that the character following abracadabra were d. This is not predicted from the current k = 2 context ra. Consequently, an escape event occurs in context ra, which is coded with a probability of 1/2, and then the k = 1 context a is used. This does predict the desired symbol through the prediction a->d, with probability 1/7. In fact, a more accurate estimate of the prediction probability in this context is obtained by noting that the character c cannot possibly occur, since if it did

It would have been encoded at the k = 2 level. This mechanism, called exclusion corrects the probability to 1/6 as shown in the third column of Table 6.2. Finally, the total number of bits needed to encode the d can be calculated to be 3.6.

If the next character were one that had never been encountered before, say t, escaping would take place repeatedly right down to the base level k = -1.

Once this level is reached, all symbols are equi probable except that, through the exclusion device, there is no need to reserve probability space for symbols that already appear at higher levels. Assuming a 256-character alphabet, the t is coded with probability 1/251 at the base level, leading to a total requirement of 11.2 bits including those needed to specify the three escapes.

6.2.2 Performance Of PPMC

It may seem that PPM's performance should always improve when the maximum context length is increased, because the predictions are more specific. Figure 6.1 shows how the compression ratio varies when different maximum context lengths are used. The graph shows that the best compression is achieved when a maximum context length of five is chosen and that it deteriorates slightly when the context is increased beyond this.

This general behavior is quite typical. The reason is that while longer contexts do provide more specific predictions, they also stand a much greater chance of not giving rise to any prediction at all. This causes the escape mechanism to be used more frequently to reduce the context length down to the point where predictions start to appear. And each escape operation carries a small penalty in coding efficiency.

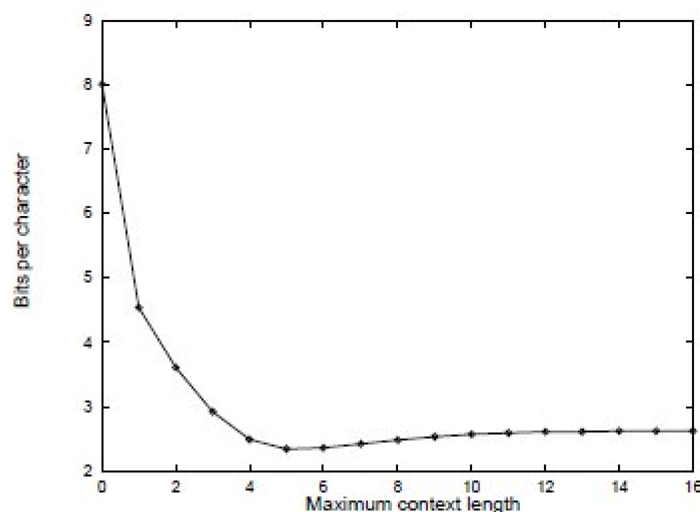


Fig 6.1 Figure Showing PPM compression ratio varies with the length of context.

6.3 Longer Contexts

An alternative to PPM's policy of imposing a universal _xed maximum upper bound on context length is to allow the context length to vary depending on the coding situation. It is possible to store the model in a way that gives rapid access to predictions based on any context, eliminating the need for an arbitrary bound to be imposed.

We call this approach, in which there is no a priori bound on context length, PPM*.
It bestows the freedom to choose any policy for determining the context to be used for prediction, subject only to the constraint that the decoder must be able to make the same choice despite the fact that it does not know the upcoming character.
How to choose which context is the best for prediction is an area of intense research. One attractive-sounding possibility is to keep a record, for each context, of how well it compressed in the past. The same record could be maintained independently by both encoder and decoder, and they could use the context with the best average compression. Curiously, this policy does not perform well in practice. This can be explained by considering its behavior under random input. Then some contexts will perform better than others purely by chance, and the best-performing ones will be selected for prediction. Of course, with random input good performance in the past is no guarantee of good performance in the future. The best policy is to use a zero-length context, and the worst thing one can do is to use a relatively \extreme" context, even if its historical performance does lie markedly above that of its competitors.
A simple but effective strategy is as follows. A context is defined to be deterministic when it gives only one prediction. We have found in experiments that for such contexts the observed frequency of the novel characters is much lower than expected based on a uniform prior distribution. This can be exploited by using such contexts for prediction. The strategy that we recommend is to choose the shortest deterministic context currently in the context list. If there is no deterministic context, then the longest context is chosen instead.

The main problem associated with the use of unbounded contexts is the amount of memory necessary to store them. It has often been noted that it is impractical to extend PPM to models with a substantially higher order because of the exponential growth of the memory that is required as k increases. For PPM*, the problem is even more daunting, as it demands the ability to access all possible contexts right back to the very first character.

6.3.1 Context Tries

A key insight in solving this problem is that the trie structure used to store PPM models can operate in conjunction with pointers back into the input string. In particular, a leaf node can point into the input string whenever a context is unique. Then, if the context needs to be extended, it is only necessary to move the input pointer forward by one position. To update the trie, a linked list of pointers to the currently active contexts can be maintained, with the longest context at the top. We call the resulting data structure a context trie.

Figure 6.2 illustrates the context trie for the string abracadabra. The root node of the trie (the null string _") is at the top. Contexts that have occurred before in the input string extend downward until they become unique, at which point a pointer, shown by a dashed line in the diagram, is stored back into the input string. For example, looking to the very left of the tree, none of a, ab, abr, abra are unique| they all appear two or more times in the input string, whereas abrac is unique.

Consequently it is at this level that a pointer into the input string is substituted for further refinement of the trie structure.

The context list is shown at the lower right. It relates to the current position in the input string, and contains pointers to the contexts that are currently active.

These are labeled 0 to 4 in the boxes on the left, and the corresponding nodes are marked with numbered arrows. The longest active context abra is placed at the top of the list, and each context below it is missing one further character. The number of elements in the

context list is the length of the longest context, plus one for the root
5 node. The list always contains at least one node the root.
As each character is processed, the context trie is updated by updating each node pointed at by the context list. There are four possibilities when updating a node, depending on the new symbol in the input string and the state of the node.

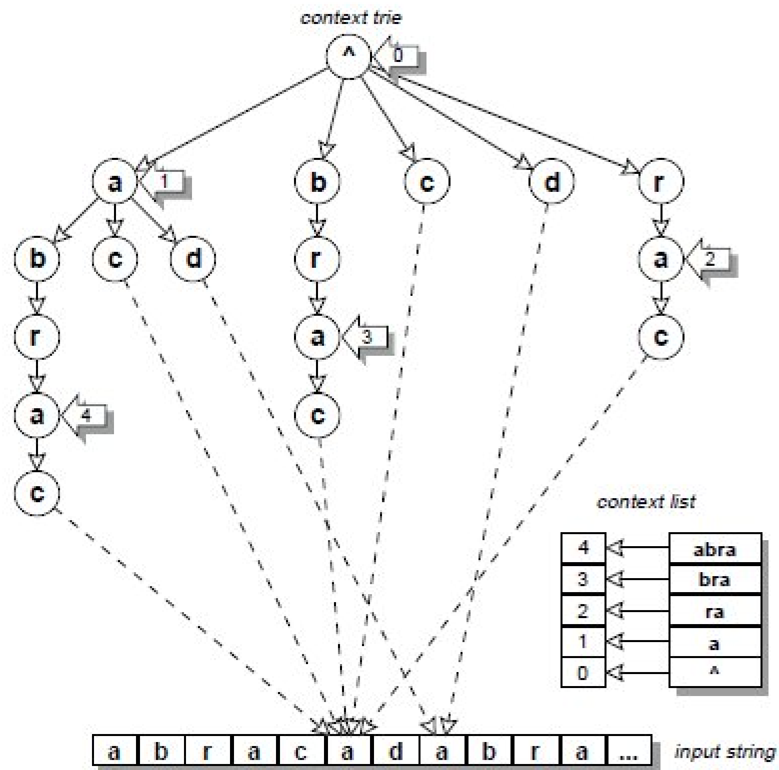


Fig 6.2 Context Trie for string abracadabra

Results

With the implementation Arithmetic coding algorithm for various strings, we found that for some of the strings after generating a floating point no the Decompression is giving valid output, but for few inputs there is an error of 1-2 characters/data.

For example for giving the input “BILL GATES”

The encode form that generated by the algorithm was 0.2572167752

And the decompressed string was comes out as “BILL GATEE”

While for few string like “JAYPEE”

the encoded form gave back the valid string

the discrepancy is due to the no of bits that are considered, for long floating point decimal values the MSB and the LSB has to be considered as well.

```
"C:\Users\Ankit\Desktop\arithmetic coding\Ac.exe"
**Implementation of Arithmetic Coding for Text Message***
the input data to be encoded
BILL GATES
input string is: BILL GATES

B->1
0.100000
I->1
0.100000
L->2
0.200000
 ->1
0.100000
G->1
0.100000
A->1
0.100000
T->1
0.100000
E->1
0.100000
S->1
0.100000

->1
0.100000

probability of occurence for characters:
 ->0.000000 to 0.100000
A->0.100000 to 0.200000
B->0.200000 to 0.300000
E->0.300000 to 0.400000
G->0.400000 to 0.500000
I->0.500000 to 0.600000
L->0.600000 to 0.800000
S->0.800000 to 0.900000
T->0.900000 to 1.000000

symbol bieng coded       range           low               high
        B                1.000000        0.200000002980    0.300000011921
        I                0.100000        0.250000000000    0.260000020266
        L                0.010000        0.256000012159    0.258000016212
        L                0.002000        0.257200002670    0.257600009441
                         0.000400        0.257200002670    0.257239997387
        G                0.000040        0.257216006517    0.257220000029
        A                0.000004        0.257216393948    0.257216811180
        T                0.000000        0.257216781378    0.257216811180
        E                0.000000        0.257216781378    0.257216781378
        S                0.000000        0.257216781378    0.257216781378
The encoded form of input after arithmetic coding is:

        0.257216781378
```

"C:\Users\Ankit\Desktop\arithmetic coding\Ac.exe"

```
The encoded form of input after arithmetic coding is:

        0.257216781378


        1.size of input string:15 bytes
2.size of encoded form:4 bytes

B
0.57216775
I
0.72167736
L
0.60838675
L
0.04193366

0.41933659
G
0.19336583
A
0.93365824
T
0.33658218
E
0.36582175
E
0.65821743

string after decompression is:

BILL GATEE
```

"C:\Users\Ankit\Desktop\arithmetic coding\Ac.exe"

```
JAYPEE
input string is: JAYPEE

J->1
0.166667
A->1
0.166667
Y->1
0.166667
P->1
0.166667
E->2
0.333333

->1
0.166667

probability of occurence for characters:
A->0.000000 to 0.166667
E->0.166667 to 0.500000
J->0.500000 to 0.666667
P->0.666667 to 0.833333
Y->0.833333 to 1.000000

symbol bieng coded      range           low                 high
         J              1.000000        0.500000000000      0.666666686535
         A              0.166667        0.500000000000      0.527777791023
         Y              0.027778        0.523148179054      0.527777791023
         P              0.004630        0.526234567165      0.527006208897
         E              0.000772        0.526363193989      0.526620388031
         E              0.000257        0.526406049728      0.526491761208
The encoded form of input after arithmetic coding is:

        0.526406049728


        1.size of input string:15 bytes
2.size of encoded form:4 bytes

J
0.15843628
A
0.95061767
Y
0.70370597
P
0.22223565
E
0.16670695
E
0.00012083

string after decompression is:

JAYPEE
```

A new method of text compression, PPM*, has been described that outperforms all others on test _les such as the Calgary corpus. The method revolves around the use of ever-growing contexts, and a data structure has been detailed that permits arbitrarily long contexts to be represented efficiently.

Also described is another, seemingly quite different, method of compression that has been introduced very recently. Surprisingly, this also appears to gain its power from its ability to utilize unbounded contexts.

Although there are a number of obvious areas in which further investigation will probably result in improvements to PPM*, it already provides a 5.6% performance increase over its predecessor, PPM. While this is not a large practical gain, we are clearly in an area where diminishing returns are to be expected. The most important contribution of PPM* is in pointing the way towards a general treatment of unbounded contexts.

REFERENCES

1. Mark Nelson, Jean Loop Gailly.Data Compression.

2. Arturo San Emeterio Campos(From www.arturocampos .com)
Arithmetic Coding.

3. Ian H. Willen, Radford M. Neal, and John G. Clearly,'Arithmetic Coding For Data Compression', June 1987.

4. Guy E. Blelloch, 'Introduction to Data Compression', Computer Science Department, Carnegie Mellon University, blellochcs.cmu.edu.

5.David Salomon,' Data compression The complete Reference', Springer Publication.

6. John.G.Cleary, W.J. Teahan and Witten, 'Unbounded Length Context for PPM', University of Waikato, New Zealand.

7. Alistair Moffat, 'Implementing PPM Data Compression Scheme', IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. **38,** NO. **11,** NOVEMBER **1990.**

8. Arturo San Emeterio Campos, 'Implementation of PPM using Hash Table and Trie'

9.Shubhangi Bhosle, 'Prediction By Partial Match', July 2013.

10. F. Choong, M. B. I. Reaz, T. C. Chin, F. Mohd-Yasin, 'Design and Implementation of a Data Compression Scheme: A Partial Matching Approach', Multimedia University, 63100 Cyberjaya, Selangor, Malaysia.

11. Daniel S. Hirschbergy and Debra A. Lelewerz,' Context Modeling for Text Compression'.