A COMPARISON OF METHODS FOR TEXT COMPRESSION

By

SUNNY CHOI

Bachelor of Science

Ewha Woman's University

Seoul, South Korea

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1989

A COMPARISON OF METHODS FOR

TEXT COMPRESSION

Thesis Approved:

[illegible]

Thesis Adviser

[illegible]

[illegible]

Norman N. Durham

Dean of the Graduate College

PREFACE

One of the purposes of this research was to introduce several well-known text compression methods and to test them in order to compare their performances, not only with each other, but also with results from various previous researches. Welch's implementation of the Ziv-Lempel method was found to outperform any other single method introduced in this thesis, or any combination of methods.

One other purpose of this research was to calculate the average distance from any one bit to the next synchronization point in static Huffman decoding, following a decoding error. The average distance in words decoded was predicted to be the average length of a codeword in bits, and tests on resynchronization showed that this was a good prediction.

Acknowledgements and thanks go to my thesis adviser Professor K. M. George for his help, guidance, and patience during the entire work. My thanks also go to Professor G. E. Hedrick and Professor M. Samadzadeh for their thorough checking and suggestions on the final draft. I wish to thank Mr. Mark Vasoll and Mr. Roland Stolfa for their help with UNIX.

I am very thankful for the love and encouragement of my family, and especially my husband John Chandler for his taking care of both his and my share of parenting of our son during my final month of research, as well as his valuable suggestions and feedback at various stages throughout the work.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Background on Coding and Data Compression . . . 2
Problem Statement 4

II. AN OVERVIEW OF TEXT COMPRESSION 6

Classification of Text Compression Methods . . . 6
Fixed-Length and Variable-Length Compression . . 7
Static and Adaptive Text Compression 10
Hybrid and Other Methods 12

III. SELECTED TEXT COMPRESSION METHODS 13

Huffman Coding 13
  Static Huffman Coding 13
  Adaptive Huffman Coding 17
The Method of Bentley, Sleator, Tarjan and Wei 28
Arithmetic Coding 33
The Ziv-Lempel Method 37
Iterating and Combining Methods 44

IV. AUTOMATIC RESYCHRONIZATION IN STATIC HUFFMAN CODES 45

V. PERFORMANCE ANALYSIS AND COMPARISONS 52

Choosing Test Data 52
Measure of Performance 52
Description of Methods 54
Comparisons 56
Iterating Methods 59
Automatic Resynchronization in Static Huffman Codes 63
Comparison to Previous Results 67

VI. SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE WORK 71

BIBLIOGRAPHY 73

LIST OF TABLES

Table Page

1. International Morse Code 2

2. A Probability Table for the Alphabet {a, e, i, o, u} 34

3. Sample Frequency Table 46

4. Compression Ratios (%) (% Reduction) 58

5. The Result from the BSTW Method (% Reduction) 60

6. The Result from Substituting Blanks Before Compression (% Reduction) 61

7. The Result from Iterating Static Huffman Method (% Reduction) 62

8. Execution Times on File PAS8 63

9. Comparison of Results with Other Tests 69

LIST OF FIGURES

Figure Page

1. A Block-Block Code for STRING 8

2. A Variable-Variable Code for STRING 8

3. A Sample Frequency Table 14

4. Constructing Huffman Codes from the Table in Figure 3 15

5. Initial Empty Huffman Tree and Codewords 18

6. Huffman Tree and Codewords After the First Input "C" 19

7. Huffman Tree and Codewords After the Second Input "B" 20

8. Constructing Different Huffman Codes from the Same Zero-Frequency Table 23

9. Initial Huffman Trees Corresponding to Tables in Figure 8 24

10. Numbering a Huffman Tree Constructed from the Table in Figure 3 26

11. Updating Procedure in Algorithm V 27

12. Static Arithmetic Codings for the message "eaii!" 35

13. LZW (Lempel-Ziv-Welch) Compression Algorithm . . . 38

14. Compressing a Source Message by the LZW Algorithm 40

15. Generating LZW String Table 41

16. Decompression by "Reversed Compression" 42

17. LZW Decompression Algorithm 43

18. Histogram of Code Lengths After Processing 5001 bits in TXT1 64

19. Histogram of No. of Wrong Characters in 5001 bits of file TXT1 65

20. Semilog Plot of Resynchronization Frequencies . . . 66

CHAPTER I

INTRODUCTION

Two areas of computational theory that deal with representation of information are *coding theory* and *information theory*. Coding theory deals with reliability in transmitting information, and information theory deals with efficiency in transmitting information.

Data compression is a topic that is involved in both areas and deals with both reliability and efficiency in transmitting information. In general, there are two categories in data compression, *lossy data compression* and *lossless data compression* [Storer 1988]. In lossy data compression, the physical size of the data is reduced while preserving only the relevant information in the data. Irrelevant data such as leading blanks or trailing zeros in a business file are deleted. Recovering all of the data is not expected in lossy data compression and so it is also called *irreversible data compression* [Lynch 1985]. Image compression and analog data compression are other well-known examples in this category.

On the other hand, in lossless data compression, preserving information so as to recover the data exactly in the original form is as important as reducing the physical size of the data. Lossless data compression is also called *reversible data compression*, *text compression*, or *database compression* [Rubin 1976]. This thesis concentrates on lossless data compression, which will be referred to hereafter as text compression or simply data compression.

Background on Coding and Data Compression

Lossless coding occurs in many contexts. One well-known case is the Morse code introduced for telegraphic communication by Samuel F. B. Morse in 1838 [Encyclopaedia Britannica 1988]. Morse code is a trinary number system that has symbols "•", "—", and a *space*. The International Morse code [Reingold and Hansen 1983] is shown in Table 1.

TABLE 1

INTERNATIONAL MORSE CODE

A	•—	N	—•
B	—•••	O	— — —
C	—•—•	P	•— —•
D	—••	Q	— —•—
E	•	R	•—•
F	••—•	S	•••
G	— —•	T	—
H	••••	U	••—
I	••	V	•••—
J	•— — —	W	•— —
K	—•—	X	—••—
L	•—••	Y	—•— —
M	— —	Z	— —••

In the international Morse code, the higher frequency letters are represented by the shorter and simpler sequences of symbols. For example, "e" and "t" are represented by a single "·" and a single "–", respectively. This shortens the transmission time. The "–" is read as "dah", and "." as "dit". When the code is transmitted by sound signals, the dahs are three times the length of the dit symbols, letters are separated by a *space* that is as long as a dit, and words are separated by a *space* that is as long as a dah.

The Hamming codes were developed by Hamming in 1950 [Hamming 1950] to detect and correct some level of errors. This is still a popular method for error detection and correction today. Many codes for error detection and error correction have been introduced since then [Ingels 1971].

Codes are also used for cryptographic purposes, where the object is to conceal information [Kahn 1967]. In this area, simple substitutions of code words for text symbols may be combined or permuted during the encoding, or the code alphabet is changed in a secret way after each symbol has been encoded.

In this thesis, we are interested in lossless coding for the purpose of text compression. Security will not be a consideration, althogh a text file could be encoded cryptographically for security either before or after it is compressed. Similarly, error detection and/or correction is not addressed in this thesis, but a compressed file could be encoded for error detection after compression.

Some text compression methods are more suitable for use on specific cases than other methods, but we will concentrate on text compression in general, where there is no known specific format in the text.

Problem Statement

Text compression is a process that *encodes* a certain volume of data into a smaller volume in such a way that the information is preserved and can be fully recovered in original form by *decoding* [Ingels 1971]. Application of text compression is particularly important in a network environment and in any system that processes large volumes of data. Transmitting compressed data will take less time than transmitting uncompressed data, and compressed data take less space to store in a memory. In this thesis, the best method is usually the method that produces the shortest compressed file, although speed of compression is sometimes a consideration.

The literature on text compression includes some very simple ad hoc methods. For example, a string of 37 consecutive blanks may be replaced by "#37", where "#" is some reserved or "illegal" symbol guaranteed not to occur in the source text [Ruth and Kreutzer 1972]. This method and similar ones can save much space in computer source language files by removing leading and trailing blanks. (Trailing blanks are not stored in most microcomputer files, but are stored, for example, in all FORTRAN source files in IBM mainframe computers.) Such methods do not fit into the modern literature on text compression, however, and will not be discussed further here, but the blank substitution method described above will be tested in combination with other methods in Chapter V.

In this thesis, we introduce some of the more well-known text compression methods and compare them, and try to select the best method in general. Background information on text compression and the classification of text compression methods are discussed in Chapter II. Existing text compression methods and algorithms, their implementation, and data structures used are discussed in Chapter III. In Chapter IV, the automatic self-synchronization in static Huffman decoding is discussed and the average distance from one point to the next resynchronization point is calculated. Chapter V contains results from tests of algorithms discussed in Chapter III and of combined methods. Chapter VI summarizes all of the results from tests and discusses future work in the area of text compression.

CHAPTER II

AN OVERVIEW OF TEXT COMPRESSION

Classification of Text Compression Methods

A *code* is a mapping of words from the source alphabet into sequences of the code alphabet. The words from the source alphabet comprise the *source messages,* and the sequences from the code alphabet are called *codewords*. A source message is *encoded* if it is mapped into a codeword, and the codeword is *decoded* when it is reverse-mapped into the original source message to recover the information. A sequence of messages is called a *message ensemble*. This notation is used by Lelewer and Hirschberg [1987] and some other authors, and is a very general notation because a "message" may be either a symbol or a string. Other authors would use "symbol" in place of "message" and "text" or "message" in place of "message ensemble". We will sometimes refer to "symbols" in the case of methods that are not string-oriented.

The source ensemble STRING in Example 1 below has source alphabet {a, b, c, d, e, f, *space*}. We will use binary numbers for encoded messages throughout the thesis; thus the code alphabet is {0, 1}.

Example 1:

STRING = "aa bbb cccc ddddd eeeeee fffffff" (1)

STRING will be used later in this chapter to demonstrate different classifications.

It is not always clear how to classify codes, but in general, codes can be categorized by the lengths of the source messages and codewords, or by the duration of the codewords (i.e., by whether a codeword for a message changes) [Lelewer and Hirschberg 1987]. The classification by the lengths of messages or codewords divides codings into fixed-length and variable-length coding, and the classification by duration of codewords divides codings into static and adaptive coding [Lelewer and Hirschberg 1987]. Since our purpose of coding in this thesis is to decrease the physical volume of the message, or to compress the message the two words *coding* and *compression (method)* will be used interchangeably.

Fixed-Length and Variable-Length Compression

Either a source message or a codeword, or both, may be of fixed length. On the other hand, both a source message and a codeword may be of variable length. Compression methods can be categorized as block-block, block-variable, variable-block, or variable-variable methods [Lelewer and Hirschberg 1987], where a block-block method is a mapping from a fixed-length message into fixed-length codewords, and a variable-block method is a mapping from a variable-length message into fixed-length codewords, etc. Figure 1 is an example of block-block coding for the message ensemble STRING in (1) and Figure 2 is an example of variable-variable coding.

Source Message	Codeword
a	000
b	001
c	010
d	011
e	100
f	101
space	110

Figure 1. A Block-Block Code for STRING

Source Message	Codeword
aa	00
bbb	010
cccc	11
ddddd	100
eeeeee	1010
fffffff	10111
space	10110

Figure 2. A Variable-Variable Code for STRING

Encoding a source message of fixed length requires a simple parsing method, and usually one symbol is processed at a time. A source message of variable length requires a more complicated parsing method to separate messages out of the message ensemble. Also, decoding a codeword of fixed length does not require special parsing techniques, but decoding a codeword of variable length usually does. For example, to encode a source ensemble STRING using the table in Figure 1 as a code table, one symbol should be parsed at a time to produce the encoded message

"000,000,111,001,001,001,111,010,...,101"

("," is not part of the message, but is used for readability.) Decoding the message is just as simple: Since the length of each codeword is 3, take 3 bits at a time and map them into the source alphabet. If the table in Figure 2 is to be used as a code table for encoding and decoding the message ensemble STRING in (1), the situation is very different. The encoder and the decoder have to "know" where each word and each codeword ends, respectively.

In general, if other factors are the same, block-block codes are the simplest to implement and variable-variable codes are the most complicated to implement in terms of parsing, and parsing makes up a big part of encode/decode algorithm complexity. At the same time, the variable length codes generally are more efficient than fixed length codes [Lelewer et al. 1987]. When STRING in (1) is encoded, the size of the compressed message using the table in Figure 1 is 96 (= 32 words of 3 bits each) bits, and the size when using the table in Figure 2 is 44 bits. If each symbol in STRING is 8 bits as is usually the case with EBCDIC or ASCII symbols, the size of STRING is 256 (32 words of 8 bits each) bits. The compression ratio (which will be defined precisely in Chapter V) achieved using the table in Figure 1 is

(256 - 96) / 256 x 100% = 62.5 %,

and the compression ratio achieved using the table in Figure 2 is

(256 - 44) / 256 x 100% = 84.1 %.

In general, the better compression method yields the higher compression ratio.

Static and Adaptive Text Compression

In order to encode a message ensemble, the encoder has to know the codeword into which each message is mapped. Codewords can be decided at once, perhaps after the whole message ensemble is seen at the beginning of the encoding stage, and remain the same through the encoding process. This is called a static method. Usually, static methods take two passes over a message ensemble; once to get the probabilities of the messages in order to determine a codeword for each message, and a second time to replace each message with the corresponding codeword [Huffman 1952]. (The alternative, which is sometimes used, is to use a fixed set of codewords chosen ahead of time, based on assumed frequencies of the messages [Witten et al. 1987].) The shorter codewords are assigned to the words with higher probabilities. Since the codeword should be known to the decoder, usually the codetable is sent to the decoder. Static Huffman coding is a good example of this method.

In an adaptive method, only one-pass parsing is required in general, and codewords are "learned" while encoding

[Lelewer and Hirschberg 1987]. This is very useful especially in a network environment where the end of a message ensemble is not predictable. A codeword for each message changes from one point to another in the encoding stage. Usually an adaptive coding involves at least one data structure that changes continually to accomodate the changes in the source ensemble. For example, in dynamic Huffman coding [Knuth 1985], a Huffman tree (a binary tree that is built from the probabilities of symbols) is updated continually, and in the BSTW method [Bentley et at. 1986], an auxiliary list that contains the words used recently, with the most recently used word at the front of the list, is updated continually. Since the codewords are changed by the source message, the codeword table need not be sent to the decoder. Instead, the same changing data structure that was constructed by the encoder is constructed while decoding [Bentley et al. 1986; Gallager 1978].

In general, a static compression method is simpler and easier to implement but not as efficient. It could even be impractical because of the two-pass nature of the parsing. A dynamic compression method is more complicated to implement, but takes only one pass and usually is more efficient. Static methods pursue the best interests for the message ensemble as a whole, and therefore the local characteristics of the text are usually ignored. Adaptive compression methods respond naturally to local variations in the text, and as a result may actually be more efficient than the corresponding static methods.

Hybrid and Other Methods

Between different methods within a classification, it may not always be easy to tell to which category a compression method belongs. For example, a compression method might block a message ensemble and compress each block using a static method. If this method constructs a new probability table for each block, it is an adaptive method by block, but it is also a static method within each block. A case like this is hard to categorize. Also, if after one method compresses a message ensemble, another method is used to compress the text further, this combined method is not suitable to be classified as belonging either to the first compression method used for compression or the second. Hybrid and other methods include cases that are hard to categorize as just one method [Lelewer and Hirschberg 1987].

CHAPTER III

SELECTED TEXT COMPRESSION METHODS

Huffman Coding

Static Huffman Coding

The static Huffman method [Huffman 1952] is a block-block method that constructs minimum redundancy codes by assigning shorter codewords to the more frequent symbols in the text. The encoding process involves two parts: building a probability table and assigning a codeword to each letter, and the actual encoding.

First, the source text is scanned to get the frequencies of the symbols. The symbols are sorted by their probabilities into non-increasing order. From the sorted list, the two smallest probabilities are taken to build a binary tree in such a way that the two elements are children of a new node whose weight is the sum of the weights of its children. This new weight is inserted into the ordered list and the two smallest elements that have just been combined are deleted. This process is repeated until there is only one element left in the list. At that point there is one merged binary tree and it contains all of the symbols at the leaf level.

For any internal node of the completed binary tree, the left child is assigned a bit '0' and the right child a bit '1'. A codeword for any particular symbol is a concatenation of code bits along the path from the root to the leaf node that contains the symbol [Standish 1980].

Figures 3 and 4 show this procedure with an example. In Figure 1, symbols a1, a2, a3, a4 and a5 occurred 62, 52, 42, 24 and 20 times respectively in the text. The probabilities 0.31, 0.26, 0.21, 0.12, and 0.10 are the relative frequencies 62/200, 52/200, 42/200, 24/200, and 20/200, respectively. The two smallest probabilities, 0.10 and 0.12, are combined to form 0.22. The probabilities 0.10 and 0.12 are deleted from the list while the probabilities 0.22 is added. The list of probabilities is maintained as an ordered list. The two smallest probabilities, 0.21 and 0.22 in the new list, are now combined. This process is continued until there is only one element (1.0) in the list.

<u>symbol</u>	sorted <u>freq.</u>	<u>prob.</u>
a1	62	0.31
a2	52	0.26
a3	42	0.21
a4	24	0.12
a5	20	0.10
total	200	1.00

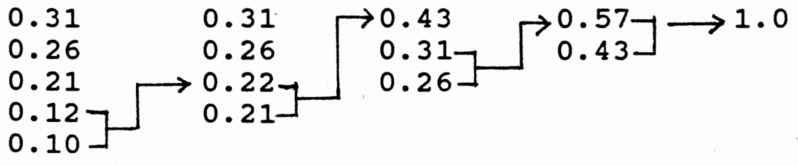
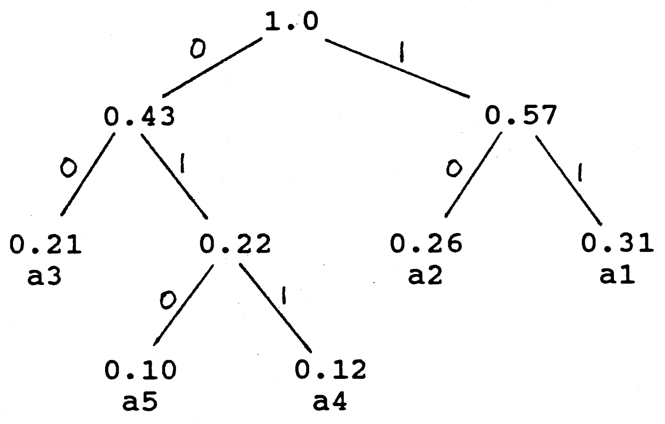


Figure 3. A Sample Frequency Table

Figure 4 is a Huffman tree constructed from the table in Figure 3. The second time the source text is scanned, each symbol is simply replaced by its codeword.



codewords: a1 ≡ 11, a2 ≡ 10, a3 ≡ 00,
a4 ≡ 011, a5 ≡ 010

Figure 4. Constructing Huffman Codes from the Table in Figure 3

In order to decode a message, the tree is constructed from the probability table in exactly the same manner as the encoder constructed it. The table can be passed on to a decoder externally or with the encoded message.

The static Huffman method guarantees to build minimum redundancy prefix codes [Huffman 1952; Lelewer and Hirschberg 1987]. (When the code for one symbol cannot be a proper prefix of the code for another symbol, the codes are called prefix codes.) However, this does not guarantee that a Huffman code will produce an encoded message of minimum length. The Huffman method assumes that the probability of each letter is constant and independent of every other letter, and does not depend on the previous letter or sequence of letters. In English text, if a letter "q" is seen, the letter "u" is expected next, and after a letter "j", it is very unlikely that a consonant will occur. These correlations imply that the Huffman assumption is usually unrealistic. Also, the Huffman method separates the encoded bits for each symbol, a limitation that is overcome by the arithmetic coding method, as will be seen later. But the Huffman scheme is simple and easy to implement and it gives considerable compression in many cases as will be seen in our experiments in Chapter V.

One other very strong argument for choosing static Huffman coding could be the well-known automatic resynchronization in case of error [Lelewer and Hirschberg 1987]. That is, if for some reason such as a noisy channel, a bit in an encoded message gets altered or lost and therefore the decoder gets out of synchronization, it usually does not take too long to become resynchronized. There has been some work in this area, mostly using deterministic analysis [Gilbert and Moore 1959; Rudner 1971] rather than stochastic models. The average length from any one bit to the next resynchronization point is discussed later (Chapter IV), and experiments on some test cases are shown in Chapter V.

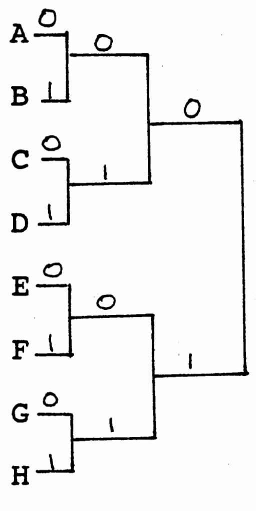
Adaptive Huffman Coding

Static Huffman coding is very simple to implement and offers optimum prefix codes. However, it takes two passes to compress a message. In a situation such as a network environment, making two passes might be impossible. Adaptive Huffman coding enables a user to start encoding without waiting for the end of the message to come in order to get the probabilities of the symbols. Another motivation for this method is when the message is known to have blocks of different characteristics, so that the overall probabilities are not very meaningful for the individual sections of the message. A source program with big blocks of comments is a good example to show this point. The overall probabilities of the source message will be quite different from the probabilities of the comment parts or the probabilities of the program parts. An adaptive Huffman code "learns" the changes in the probabilities while encoding or decoding, and may remain nearly optimal for the current estimates.

One way to accomplish some adaptivity is to break the source into fixed-size data blocks and apply the static Huffman method to each block; after blocking, the symbols are counted and sorted to build a Huffman tree, and the second time the block is scanned, symbols are encoded [Welch 1984]. Details such as multiple buffers to hold continuously incoming data have to be worked out carefully. This "batched static

Huffman" approach is acceptable if a high transmission rate is not too important and if the data blocks are very large relative to the size of the translation table [Welch 1984]. But this is not what is usually meant by an adaptive Huffman code.

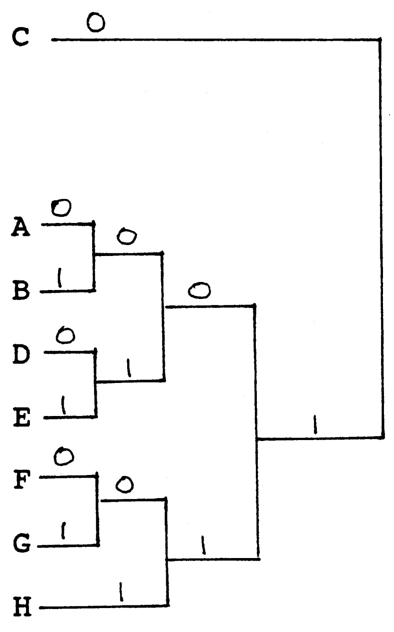
The basic idea in fully adaptive Huffman coding is to encode the *k*-th message using the Huffman tree constructed from the frequencies found in messages 1, 2, ..., *k*-1. (When *k* = 1, we start from a balanced, minimum height tree.) Figures 5, 6 and 7 show the basic idea of constructing adaptive Huffman codes for a sample message ensemble "CBA", where alphabet is {A, B, C, D, E, F, G, H}.



codewords	
A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

Figure 5. Initial empty Huffman tree and codewords

Every codeword for an 8-bit symbol set is of length 8-bits in the initial empty (zero-frequency) tree in Figure 5. The message "C" is encoded using one of these codewords. Then the frequency for the message "C" becomes 1, and the Huffman tree becomes the tree in Figure 6. So the codeword for the first symbol in the message ensemble "C" becomes "0", a one-bit codeword. As successive messages are found and encoded, their codewords likewise become short (and the codeword for "C" becomes longer!). Figure 7 shows the change in the tree after the second input symbol "B" is entered.

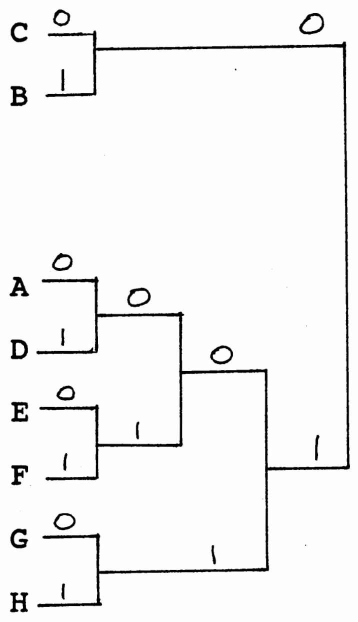


codeword for C 0

codewords for the rest

A	1000
B	1001
D	1010
E	1011
F	1100
G	1101
H	111

Figure 6. Huffman Tree and Codewords After the first input "C" and codewords



codeword for C 00

codeword for B 01

codewords for the rest

A	1000
D	1001
E	1010
F	1011
G	110
H	111

Figure 7. Huffman Tree and Codewords After the Second Input "B"

If this were done without modification, however, the method would become less and less rapid to adapt as *k* become very large, because the last 50 messages, for example, would be a smaller and smaller fraction of the total. Hence the goal is modified as follows: use the frequencies from some "recent" subset of messages. This could be done using a queue, but this would be very slow. So the adaptive algorithms instead multiply each cumulative frequency by a fixed positive constant c, $c < 1$, after every N-th message [Gallager 1978]. Except for this multiplication, a frequency is incremented by 1 each time the corresponding message occurs. The multiplication by c causes the algorithm to "forget" ancient history; it is very similar to an exponential smoothing filter [Brown 1963]. The value of $M = N / (1 - c)$ provides a measure of how well the system adapts to changes in the message. In general, as the value of M increases, the "memory" will go back further. If the value of M is too small, the codes will change too fast and the algorithm only "remembers" a very few recent symbols [Gallager 1978]. On the other hand, if the value of M is too large, the change is too slow and the system does not adapt to changes in the message rapidly enough. For easy multiplication in programming, $c = 2^{-k}$ (usually, $c = 0.5$) should be a good choice, since a simple shift operation can replace the multiplication.

Just as in the static Huffman coding, it is assumed that the probability of each letter is independent of every other, and does not depend on the previous letter or sequence of letters. Even though there is no concern for correlations between symbols, this method responds to locality of the source message, because the codes are changing while the encoder or the decoder is "learning" the environment. After each input symbol, the cumulative frequencies up to the current point are used to encode or decode the next symbol [Gallager 1978].

A simple implementation of this "learning" concept can be effected by building a Huffman tree after each input, based on the cumulative frequencies up to the current point. The initial frequency table contains all frequencies equal to zero before any symbol is entered from a message ensemble, and the initial Huffman tree is built based on this table. Since the initial table contains all zeros for frequencies, sorting is not necessary. Furthermore, the new position to insert the sum of the two smallest frequencies can be anywhere in the list, since the sum of the two frequencies is also zero. As an example consider the following message:

"a b b b ... b c" (containing n copies of b)

where a < b < c. There are n+1 places where a new copy of b could be inserted while preserving the order. The shape of the initial tree depends on how the new position is determined, and the shape affects the height of the tree. Trees in Figure 8 and 9 show the two different possible initial trees based on the same zero-frequency table for alphabet the {A, B, C, D} in order. Tree (a) in Figure 9 is built by inserting the new frequencies at the top of the (sorted) list as shown in table (a) in Figure 8, and tree (b) in Figure 9 is built by inserting them at the end of the list as shown in table (b) in Figure 8.

The maximum length of a codeword in tree (a) in Figure 9 is 2, and the maximum length in tree (b) is 3 when the size of alphabet is 4. We can imagine what kind of difference it would make in constructing a Huffman tree for the ASCII symbol set (expressed in eight-bit bytes), where the size of the alphabet is 256.

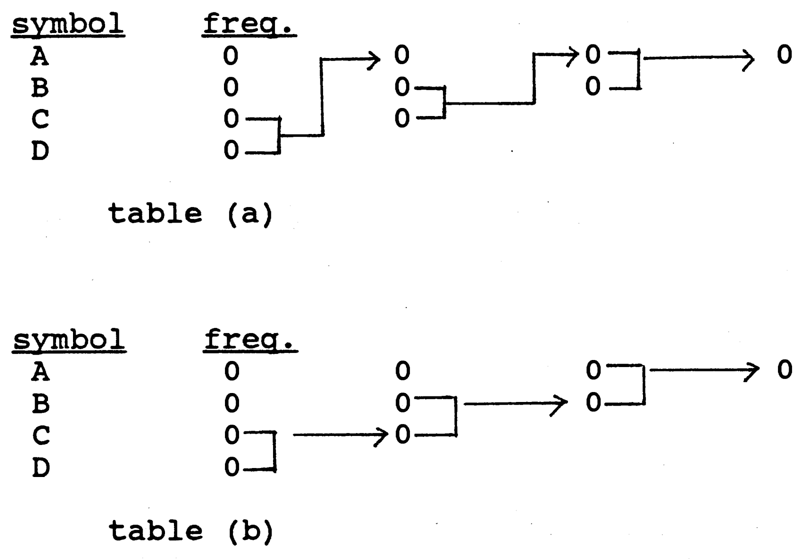


Figure 8. Constructing Different Huffman Codes From the Same Zero-Frequency Table

When the size of the alphabet is 256, a tree (a)-like construction will produce a perfectly balanced binary tree with the maximum length of a codeword equal to $\log_2 256 = 8$, and a tree (b)- like construction will produce a maximum length codeword of length 256.

One simple way of keeping the height of the tree as short as possible is to start with frequencies that are ones instead of zeros. Then the Huffman tree built from this table automatically will be a balanced binary tree that contains all of the ASCII symbols with weight 1 at the leaf level.

After the initial tree is built in this implementation, the table is updated after each message, and the tree is rebuilt based on the current table. If one chooses to bias the frequency by 1, it can be easily seen that the increment for the frequency corresponding to the current symbol should be at least as large as the size of the alphabet (256, for example, in an eight-bit ASCII symbol set) instead of 1, in order to keep the output of the algorithm the same as the output from the algorithm based on the true frequencies of the symbols. This brute force method of repeatedly rebuilding the entire tree is very expensive and not very practical, but it leads us to the next implementation.

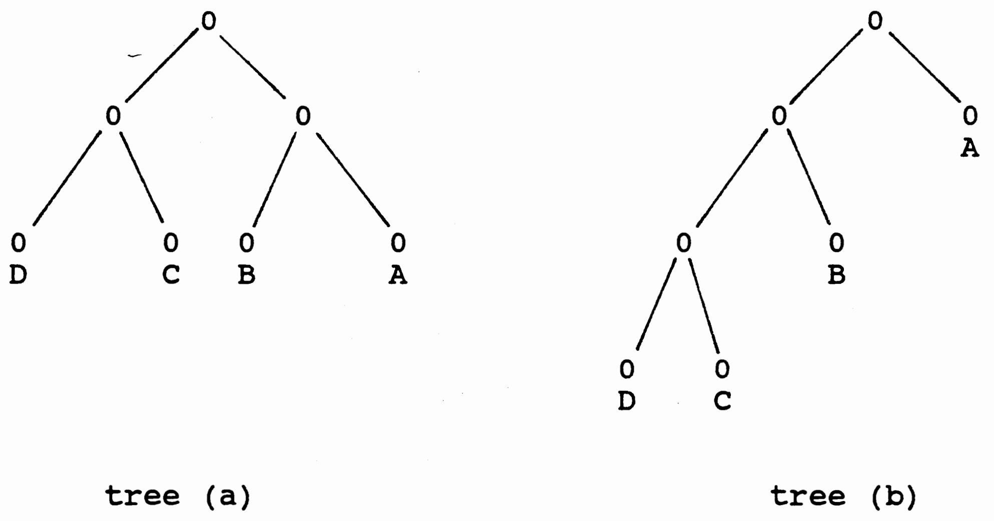


Figure 9. Initial Huffman Trees Corresponding to Tables in Figure 8

A method of implementing adaptive Huffman coding efficiently was proposed by Faller [1973] and Gallager [1978] independently, and improved by Knuth [1985], and recently by

Vitter [1987; 1989]. The adaptive version of Huffman's algorithm implemented by Faller, Gallager, and Knuth is called the FGK algorithm for short. In the FGK algorithm, the tree is continually updated (not rebuilt from scratch as in the brute force method) using only the minimum possible number of pointer exchanges. Unlike the static Huffman or the batched static Huffman implementation, the probability table need not be transmitted, because the encoder and the decoder are synchronized every step of the way.

The FGK algorithm involves two parts: updating the probability table and updating the Huffman tree. Updating the table is very simple. Updating a tree in the FGK algorithm is simplified and improved by Vitter in Algorithm V [Vitter 1987], by reducing the number of pointer exchanges and by minimizing the height of the tree, even though the complexity of the algorithm is the same (the time required for each encoding and decoding operation on one symbol in both implementations is $O(l)$, where l is the current length of the codeword for the symbol).

The data structure used in Algorithm V is called a *floating tree*, a tree whose pointers are not maintained explicitly. Instead of keeping an explicit tree, this algorithm uses an explicit numbering that corresponds to the physical storage locations used to keep information about the nodes, in order to maintain the Huffman tree without reconstructing the whole tree. (A similar method of storing a tree is used in the well-known Heapsort algorithm [Reingold and Hansen 1983].) If the nodes in Figure 4 are numbered explicitly by their physical storage locations as Algorithm V would do, they will be labeled as in Figure 10.

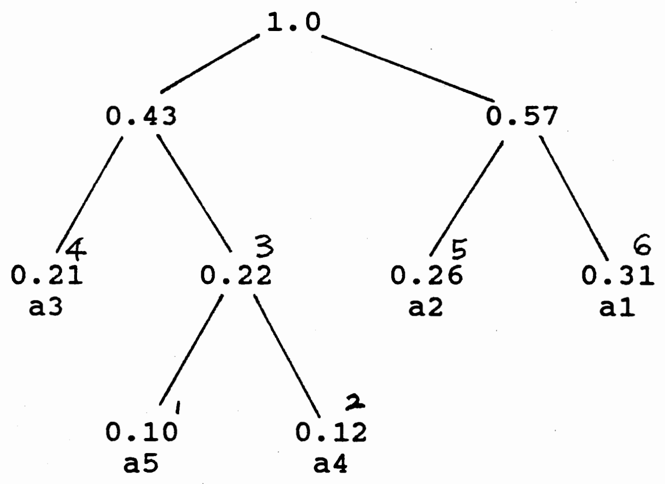


Figure 10. Numbering a Huffman Tree Constructed from the Table in Figure 3

Figure 11 is the updating subroutine of Algorithm V [1987], when the (t+1)st symbol in the message is processed. The full implementation is found in the more recent paper [Vitter 1989]. There is a special node called the 0-node in this algorithm, which represents all of the symbols whose weight is zero, or in other words the symbols that have never occurred so far. Initially the tree contains only one 0-node. The line starting with "####" is not in the simplified version of Algorithm V [Vitter 1987] but needs to be added in order to update the weights of the nodes involved along the path.

```
procedure *Update*;

{The alphabet contains a(1), a(2), ..., and a(n).
 The input being encoded is a(i) and it is (t+1)st a(i)
 entered so far.
 There are n-k unused symbols in the 0-node.
}

begin
  q := leaf node corresponding to $a(i)_{t+1}$;
  if (q is the 0-node) and (k < n - 1) then
     begin
       Replace q by a parent 0-node with two leaf 0-
          node children, numbered in the order left
          child, right child, parent;
       q := right child just created;
     end;

  if q is the sibling of a 0-node then
     begin
       Interchange q with the highest numbered leaf of
          the same weight;
       Increment q's weight by 1;
       q := parent of q;
     end;

  while q is not the root of the Huffman tree do
     begin {main loop}
       Interchange q with the highest numbered node
          of the same weight;
       {q is now the heighest numbered node of the same
          weight}
       Increment q's weight by 1;
       q := parent of q;
     end;

#### Increment q's weight by 1;

end;
```

Figure 11. Updating procedure in Algorithm V [Vitter 1987]

Vitter proved that the length of the message encoded by Algorithm V will exceed the length of the output from the static Huffman method by at most one bit per codeword [Vitter 1987]. Often, the length will actually be less, due to the adaptation.

The Bentley, Sleator, Tarjan and Wei (BSTW) Method

This is an adaptive method in which locality, or the correlations between symbols, is taken into account. In this algorithm, a word is processed as an entity rather than processing characters individually, and a codeword is an integer denoting the position of the word in a self-organizing list. Frequently-used words are near the front of the sequential list, in order to have shorter integer encodings [Bentley et al. 1986; Hester and Hirschberg 1985]. As in all adaptive methods, the encoder and the decoder construct lists that are the same at corresponding points in the message.

Initially, the list (of size N) is empty (N = 0). The encoder looks for the words in the list before transmitting each word. If a word exists at position P in the list, the value of the integer P is transmitted, and the word moves to the front of the list. If the word does not exist in the list, the integer N+1 is transmitted followed by the word itself, and the word is inserted at the front of the list. If the list is already full when an insertion needs to take place, the most aged element, which is located at the rear of the list, should be deleted first.

The decoder receives an integer P and replaces it with the word in the P-th position in the list, if it exists. Again, the word moves to the front of the list. If there is no word in the P-th position, the decoder accepts a new word and inserts it at the front of the list. Just as in encoding, the least recently used element at the rear of the list is deleted before the insertion, if the list is already full.

The source text in Example 3 is taken from Bentley, Sleator, Tarjan and Wei's original article [Bentley et al. 1986]. This example shows how the list is updated while encoding proceeds.

Example 3:

```
  source ensemble : "THE CAR ON THE LEFT HIT THE CAR I LEFT"

  msg  : 1THE
  list : THE

  msg  : 1THE 2CAR
  list : CAR THE

  msg  : 1THE 2CAR 3ON
  list : ON CAR THE

  msg  : 1THE 2CAR 3ON 3
  list : THE ON CAR

  msg  : 1THE 2CAR 3ON 3 4LEFT
  list : LEFT THE ON CAR

  msg  : 1THE 2CAR 3ON 3 4LEFT 5HIT
  list : HIT LEFT THE ON CAR

  msg  : 1THE 2CAR 3ON 3 4LEFT 5HIT 3
  list : THE HIT LEFT ON CAR

  msg  : 1THE 2CAR 3ON 3 4LEFT 5HIT 3 5
  list : CAR THE HIT LEFT ON

  msg  : 1THE 2CAR 3ON 3 4LEFT 5HIT 3 5 6I
  list : I CAR THE HIT LEFT ON
```

```
msg  : 1THE 2CAR 3ON 3 4LEFT 5HIT 3 5 6I 5
list : LEFT I CAR THE HIT ON
```

The decoder should build the list in exactly the same way, in order to decode the message.

Since this algorithm works with one word at a time, the message ensemble has to be grouped into words so that a processor knows where each word ends. The character domain is grouped into two disjoint sets first, letters and non-letters. A message is an alternating sequence of words that consist only of letters and words that consist only of non-letters.

It is up to the user to determine what are letters and non-letters, and they should be chosen using criteria appropriate to the application. In English text, it may be a good choice to set all of the ordinary letters and numbers as "letters" and all other characters as non-letters. But if a message contains many special characters combined with alphabetic characters to form certain names, and if they are used repeatedly, for example, care should be taken in order not to break up one name into several pieces. Breaking a name not only fills up the list unnecessarily, but also increase the length of the encoded message.

For example, if a message contains some often-repeated hyphenated words, it might be best to declare the hyphen to be a "letter", so that the hyphenated words, rather than just their individual components, are considered as "words" in the encoding. Similarly, in encoding a PASCAL program containing many references to array elements LIST[J], square brackets should perhaps be declared to be letters. Having to know the characteristics of the source ensemble for the best result may be the weakest part of this algorithm.

A self-organizing list is used in the BSTW method. It involves three basic operations: insertion, deletion, and searching. A move-to-front operation which moves the most recently used element to the front of the list is just a deletion followed by an insertion. A linked list was used in this work to simplify the implementation, whereas Bentley et al. discussed using an interlinked binary tree and a binary trie [Knuth 1973]. Neither Bentley et al., nor anyone else, however, has produced a production code for this algorithm using these data structures, according to the published literature [Bentley et al. 1986].

Insertions and deletions will not differ between the two implementations, but the searching would be a lot faster with a trie structure, at the cost of complicating the list updating [Bentley et al. 1986]. Either way, each node in the list must be large enough to hold the longest word. For a defense against a possible overflow (a word longer than the maximum word length allowed a priori), it would be a good idea to have encoding start all over from the beginning with an empty list from the overflow point, if overflow ever does occur.

Encoded messages consist of numerical position numbers, letter strings, and non-letter strings. Since a decoder has to know where each string ends, a one-byte character was used to denote end-of-word in a test program. But if the algorithm were to be used only on the seven-bit ASCII character domain, the highest bit of each eight-bit byte could be set as a flag to indicate either an end-of-letter (or non-letter) string or a decimal position number, since 7 bits are enough to represent all ASCII characters.

Because the output from the BSTW encoding is still in byte-oriented character form, it is possible to use another compression technique that takes byte form as input, such as Huffman coding, after the BSTW method has been applied, in order to compress the message further. In published tests [Bentley et al. 1986], the BSTW method has not been implemented in the basic form described above. Instead, the output from the BSTW method has been further compressed. For example, Bentley et al. themselves wrote [Bentley et al. 1986], "For ease of implementing the prototype, we encoded the position in the list by a Huffman code, which implies that an implementation would have to make two passes over the data." Similarly, Fiala and Greene wrote, "Since the empirical results in [BSTW] do not actually give an encoding for the positions of words in the list or for the characters in new words that are output, we have taken the liberty of using the V compressor (Vitter's adaptive Huffman algorithm) as a subroutine to generate these encodings adaptively."

In this thesis, we have chosen to implement the BSTW idea in a simple block-to-block scheme, in order to see how much compression the word-list idea achieves by itself. This is what we refer to from here on as BSTW. Separately, we implemented BSTW-followed-by-Huffman; it is the results of this compound algorithm that should be compared to published BSTW results [Bentley et al. 1986].

One question that remains unanswered is whether BSTW encoded characters into new words using Huffman compression, and if not, why not. The quote above implies that they did not, but this would seem to be a strange thing to do, because the compression ratio achieved by the BSTW alone will be very poor, as will be seen in Chapter V.

Arithmetic Coding

This method was invented by Elias and implemented by Rissanen [1976], Rubin [1979], Cleary and Witten [1984a], and Witten et al. [1987]. This method is rather similar to Huffman coding: symbols are encoded independently and the correlations between symbols are not considered. In arithmetic coding, a message is represented by an interval of real numbers between 0.0 and 1.0. Each symbol in a source message specifies a subinterval and the subinterval maps into the interval [0.0, 1.0) again.

Initially, a real interval between 0.0 and 1.0 is divided into subintervals corresponding to the probability of each symbol. The more frequent symbols take larger subintervals within the interval between 0.0 and 1.0. When the first symbol a_1 in the source message is entered, the subinterval that belongs to the symbol a_1 is expanded and mapped into the interval [0.0, 1.0). All of the subintervals get new ranges. Figure 12 shows an example of this narrowing, expanding, and remapping process as each input symbol in the source message "eaii!" is processed using arithmetic encoding, based on the sample probability table in Table 2.

Every time a subinterval expands to map into the interval [0.0, 1.0), the new range of each symbol can be defined by the following formulae [Lelewer and Hirschberg 1987].

$$\text{new\_left} = \text{prev\_left} + \text{msgleft} \times \text{prev\_size} \quad (2)$$
$$\text{new\_size} = \text{prev\_size} \times \text{msgsize} \quad (3)$$

TABLE 2

A PROBABILITY TABLE FOR THE ALPHABET {a, e, i, o, u, !}

Symbol	Probability	Range
a	.2	[0.0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

For example, after the first character "e" is seen, the new range of the symbol becomes [0.2, 0.5) by plugging numbers into the formulae (2) and (3) above;

$$\text{new\_left} = 0.0 + 0.2 \times (1.0 - 0.0) = 0.2,$$
$$\text{new\_size} = (1.0 - 0.0) \times (0.5 - 0.2) = 0.3,$$
$$(\text{new\_right} = \text{new\_left} + \text{new\_size} = 0.5).$$

And after the character "a" is seen,

```
     new_left = 0.2 + 0.0 x 0.3   = 0.2,
     new_size = 0.3 x (0.2 - 0.0) = 0.06,
```

and therefore the new range becomes [0.2, 0.26).
At the end of encoding, the two real numbers that define a final range contain all of the information needed to be decoded. In fact any one number within the last range will serve the purpose sufficiently. The example in Figure 12, "!" has the final range [0.23354, 0.2336), and a number within the range such as 0.23355 can be the encoded message for the string "eaii!".

	interval
	[0, 1)
e	[0.2, 0.5)
a	[0.2, 0.26)
i	[0.23, 0.236)
i	[0.233, 0.2336)
!	[0.23354, 0.2336)

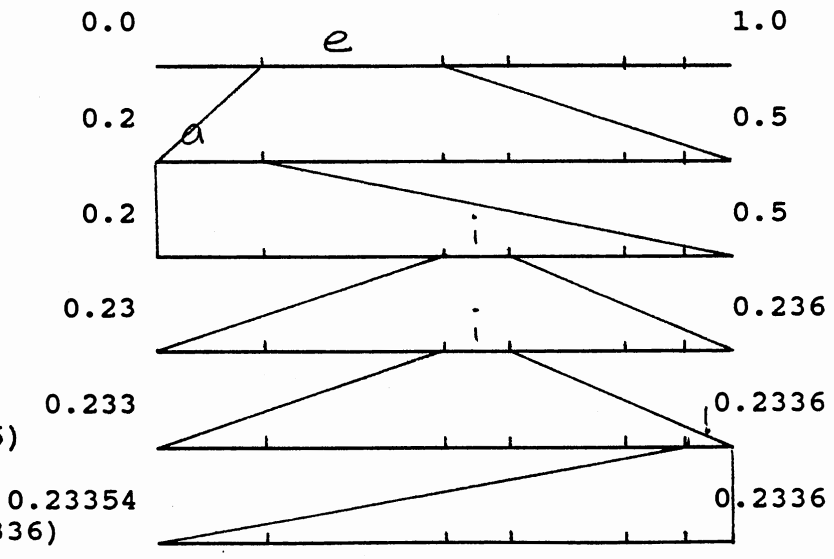


Figure 12. Static Arithmetic Codings for the Message "eaii!"

The decoding process is almost identical with the encoding process. The encoded message, which is a number in the final range in the interval [0.0, 1.0), shows which part of the original interval the encoded number came from. For example, the encoded message of Figure 12, "0.23355", shows that the number belongs to the range of the symbol "e", [0.2, 0.5) in the original segment [0.0, 1.0). The symbol "e" is sent to the output and the range [0.2, 0.5) is mapped into the interval [0.0, 1.0) just as in encoding. The number "0.23355" is found this time in the range of the symbol "a", which is [0.2, 0.26). So the symbol "a" is sent to the output and the substring of range 0.2 and 0.26 expands again. This could go on forever since the decoder would not know when to stop expanding. For example, the number 0.0 could mean "a", "aa", "aaa", "aaaa", etc. Obviously a decoder needs to know when to stop. The symbol "!" in the example in Table 2 and Figure 12 is being used as an end-of-message symbol.

Witten et al. [1987] show the complete algorithm in C language. They also show that the encoded message doesn't need to be held until the final range is calculated. Whenever the leftmost digits of the endpoints of the current interval do not show any further change, those digits can be sent to the output. In the example in Figure 12, instead of waiting until the whole message is encoded, digit 2 could be sent out after the symbol "e", because that digit is not going to change any more, and digit 3 could be sent after the first "i" is seen, and so on.

This algorithm can be implemented either as a static algorithm or as a dynamic algorithm. In static arithmetic coding, the probabilities of the symbols are given ahead of time, and in a dynamic coding, the estimated probabilities of symbols entered so far are used. Witten et al. show the algorithm written in C [Witten et al. 1987], where generally known probabilities in English text are used in the static model instead of exact probabilities of the symbols in the text to be compressed.

Because arithmetic coding does not produce one codeword for each symbol, it can produce fewer bits of output for a given message than does the static Huffman method.

The Ziv-Lempel Method

The Ziv-Lempel method is a class of compression methods rather than a single compression algorithm. The Ziv-Lempel style of parsing and textual substitution in adaptive text compression was suggested by Ziv and Lempel [Lempel and Ziv 1976; Ziv and Lempel 1977; Ziv and Lempel 1978], but different implementations by many others [Storer and Szymanski 1982; Rissanen 1983; Welch 1984] improved the algorithm in many ways. Basically, the encoding process parses the message repeatedly to find the longest recognized string that exists in the string table [Ziv and Lempel 1977; Lelewer and Hirschberg 1987].

Welch's implementation was chosen here to show how the algorithm works. Initially, the string table contains single-character symbols which are assigned unique code values [Welch 1984]. This saves some expansion during initial encoding, in comparison to using the table without single-character symbols at the beginning [Lelewer and Hirschberg 1987]. Next, source text is parsed just until the parsed part of the string does not exist in the table. In other words, the parsing stops when the longest recognized string plus one more symbol have been parsed. This new string is added to the table with a unique code value assigned, and the code value of the longest recognized string (without the new symbol) becomes output. The final symbol that made the string different from the longest recognized string in the table is the beginning of the next string and the beginning of the next parsing. This step is repeated until the source text is exhausted. Figure 13 shows Welch's version of the Ziv-Lempel compression algorithm [Welch 1984].

```
Initialize table to contain single-character strings.
Read the first input character -> prefix string w

Step: Read next input character K
     If no such K exists (end of input):
          code (w) -> output; EXIT

     If wK exists in string table:
          wK -> w;  repeat Step.

     Else wK not in string table:
          code (w) -> output; wK -> string table;
          K -> w; repeat Step.
```

Figure 13. LZW (Lempel-Ziv-Welch) Compression Algorithm

Figure 14 shows an example for Welch's Ziv-Lempel encoding process and the string table that is created. All of the symbols a, b, and c are first stored in the table with unique codes 1, 2, and 3 assigned before actual parsing takes place. The source message is parsed up to the first "b", since the longest recognized string at this point was "a". The codeword for the recognized string "a", namely 1, is sent to the output and the new string "ab" in the alternate form "1b", in actual practice is added to the table and assign a new codeword "4". The extension character "b" becomes the start of the next string, parsing stops at the second "a", and the codeword for the longest recognized string "b", namely 2, is sent to the output and the new string "ba" in the form "2a" is added to the table with a new codeword 5. The extension character "a" is the beginning of the next parsing and the parsing stops after "abc", since "ab" is the longest recognized string. The codeword for "ab", namely 4, is sent to the output and the new string "abc" in the shorter alternative form "4c" is added with a new codeword. This process continues until the end of the source message is reached. It is easy to see that the encoded message for the example source message is "1 2 4 3 5 8 1 10 11". When storing the string table in Welch's implementation, since each table entry is a prefix string plus a single character, it can be stored in a fixed length of storage. Each table entry is encoded as (i, c), where i is the codeword for the prefix string and c is the extension character [Welch 1984; Lelewer and Hirschberg 1987]. The alternative table in Figure 15 makes storing the string table easier to understand.

source message = "a b a b c b a b a b a a a a a a a"

position	table	output
1	a	
2	b	
3	c	
		1 (= a)
4	ab = 1b	
		2 (= b)
5	ba = 2a	
		4 (= ab)
6	abc = 4c	
		3 (= c)
7	cb = 3b	
		5 (= ba)
8	bab = 5b	
		8 (= bab)
9	baba = 8a	
		1 (= a)
10	aa = 1a	
		10 (= aa)
11	aaa = 10a	
		11 (= aaa)
12	aaaa = 11a	

Figure 14. Compressing a Source Message by the LZW Algorithm

Rodeh et al. [1981] point out that a straightforward implementation of the Ziv-Lempel algorithm takes $O(n^2)$ time to process a string of length n. Hashing was proposed by Welch [1984] to achieve O(n) processing time, and the fixed-size entries in string table are well suited for it. The public domain UNIX utility *compress* implements LZW, Welch's version of the Ziv-Lempel algorithm. In *compress*, once the string table is full, the table construction starts over from scratch (in effect the current table is destroyed and the algorithm starts over using only the single-character symbols).

string table		alternative table	
a	1	a	1
b	2	b	2
c	3	c	3
ab	4	1b	4
ba	5	2a	5
abc	6	4c	6
cb	7	3b	7
bab	8	5b	8
baba	9	8a	9
aa	10	1a	10
aaa	11	10a	11
aaaa	12	11a	12

stored table = {(0,a),(0,b),(0,c),(1,b),(2,a),(4,c),
(3,b),(5,b), (8,a),(1,a),(10,a),(11,a)}

Figure 15. Generating LZW String Table

The decompression process constructs the table in the same way, as the message is translated. "Compression-in-reverse" (such as in the BSTW decompression) can be a simple-minded implementation that shows the concept of the decompression algorithm in a simple way. In this implementation, it is assumed that the string table stores variable-length whole strings rather than shorter, fixed alternative forms, in order to provide a way to search a string in the table. Decompression is a recursive operation in which the codeword produces the last (extension) character and the codeword of the prefix string in reverse order. Figure 16 shows simplified steps in this implementation, and also illustrates a certain problem that occurs. Since the last character in a string is peeled off one at a time, the output string comes out in reverse order, even though Figure 16 shows strings in the right order for the sake of comprehensibility.

message to decode = "1 2 4 3 5 8 1 10 11"

position	table	output
1	a	
2	b	
3	c	
		a (= 1)
		b (= 2)
4	ab	
		ab (= 4)
5	ba	
		c (= 3)
6	abc	
		ba (= 5)
7	cb	
		? (= 8)

Figure 16. Decompression by "Reverse Compression"

These steps are almost like the steps in Figure 14, except that after the string "ba" is produced, the decoder cannot go on any more, since it has not learned what is in the table at position 8. Only substrings of the form K*w*K*w*K will cause this problem while decompressing, according to Welch [1984]. A complete decompression algorithm that solves this problem [Welch 1984] is shown in Figure 17. (The line that starts with "###" is present in Welch's decompression algorithm but it should be ignored, because it is not necessary and does not make sense.) Hashing is not necessary in decompression since the string table is accessed directly by codewords, and therefore decompression is faster than the compression process.

```
Decompression: First input code -> CODE -> OLDcode;
               with CODE = code(K), K -> output;
                                    K -> FINchar;
Next Code:     Next input code -> CODE -> INcode;
               If no new code:
                    EXIT;
               If CODE not defined:
                    FINchar -> output;
                    OLDcode -> CODE;
               ####code(OLDcode, FINchar) -> INcode;

Next Symbol:   If CODE = code(wK):
                    K -> stack;
                    code(w) -> CODE;
                    Go to Next Symbol;

               If CODE = code(K):
                    K -> output;
                    K -> FINchar;

               Do while stack is not empty:
                    stack top -> output;
                    POP stack;

               OLDcode, K -> string table;
               INcode -> OLDcode;
               Go to Next Code;
```

Figure 17. LZW Decompression Algorithm

The Ziv-Lempel parsing method is often called a greedy parsing method because the longest recognized string is sought while parsing rather than attempting to achieve global optimality. As an example, if the source message in Figure 14, "ababcbababaaaaaaa", can somehow be broken into "ab", "cb", and "aaaa" instead of into the 12 pieces produced by the Ziv-Lempel method, we would have needed only 4 codewords and would have accomplished a greater compression.

Iteration and Combining Methods

Some text compression algorithms such as the BSTW and Ziv-Lempel methods take byte-oriented source messages and produce byte-oriented encoded messages, while some others such as both static and dynamic Huffman codings produce bit-oriented output rather than bytes of constant length. This leads us to think about the possibility of running one method followed by another. If one method compressed a source text and produced byte-oriented output, surely it is possible to see more compression on that output by running the output through another method that produces bit-oriented output, since there is a possibility of wasting bits in byte-oriented output. This approach is investigated in Chapter V of this thesis.

CHAPTER IV

AUTOMATIC RESYNCHRONIZATION IN STATIC HUFFMAN CODES

The static Huffman coding method has an advantage not shared by any other method we have described. If an error occurs in transmission, such as the corruption of a few bits by noise, static Huffman decoding may produce a few errorneous symbols because it is not starting the decoding of each symbol on a correct "first" bit, but will almost certainly resynchronize itself automatically and decode the rest of the message correctly. In this chapter the average distance from any random bit to the next synchronization point, in decoding a message encoded with the static Huffman method, is calculated, assuming a simple probabilistic model.

Imagine a complete encoded message that contains codewords for letters a_1, a_2, ..., a_m whose frequencies are n_1, n_2, ..., n_m, with the length of codewords l_1, l_2, ..., l_m respectively (see Table 3). Let

$$N \equiv n_1 + n_2 + \dots + n_m = \sum_{i=1}^{m} n_i$$

denote the number of characters in the original message. Let $M = \sum_{i=1}^{m} [n_i \cdot l_i]$ denotes the number of bits in the encoded message, and let $p(a_j)$ be the probability of a_j in the Huffman method. Then

$$p(a_j) = n_j \,/\, N,$$

which can be rewritten as

$$n_j = p(a_j) \cdot N \qquad (4).$$

Let $p_1(i)$ be the probability of any bit being the first bit of any occurrence of the letter a_i in the encoded message. From the definition of a probability as a relative frequency,

$$p_1(i) = (\#\text{ of occurences of the codeword } a_i) \,/\, (\text{total bits})$$

$$= n_i \,/\, M$$

$$= n_i \,/\, \sum_{k=1}^{m} [n_k \cdot l_k] \qquad (\text{from } (4)).$$

TABLE 3

SAMPLE FREQUENCY TABLE

symbol	frequency	probability	code-length
a_1	n_1	n_1 / N	l_1
a_2	n_2	n_2 / N	l_2
.	.	.	.
.	.	.	.
.	.	.	.
a_m	n_m	n_m / N	l_m
totals	N	1	

Now, let s be the probability of any random bit being the first bit of any character in the encoded message. Then

$$s = \sum_{i=1}^{m} p_1(i)$$

$$= \sum_{i=1}^{m} [n_i \,/\, \sum_{k=1}^{m} (n_k \cdot l_k)]$$

$$= [1 / \sum_{k=1}^{m} (n_k \cdot l_k)] \cdot \sum_{i=1}^{m}(p(a_i) \cdot N) \qquad \text{(by (4))}$$

$$= [1 / \sum_{k=1}^{m} (n_k \cdot l_k)] \cdot 1 \cdot N \qquad \text{(since } \sum_{i=1}^{m} p(a_i) = 1\text{)}$$

$$= 1 / [N^{-1} \sum_{k=1}^{m} (n_k \cdot l_k)]$$

$$= 1 / [\sum_{k=1}^{m} (p(a_k) \cdot l_k)] \qquad \text{(by (4))}$$

$$= 1 / d \qquad (5),$$

where d is the average length (in bits) of a codeword. With this preparation completed, we will state and prove the following theorem.

Theorem 1 : If the decoder is out of synchronization at the current bit, and if it has a constant probability of resynchronizing after the next letter is decoded, regardless of how many incorrect letters have been decoded up to that point, the average distance from the current bit to the next resynchronization point is $D = d$, and the average number of incorrect symbols produced before resynchronization occurs is $(d - 1)$.

Proof : The probability of the next bit being the first bit of any letter is

$$p = 1 / d \qquad \text{(by (5))}.$$

and the probability of the next bit not being the first bit of any letter is

$$q \equiv 1 - p = 1 - 1 / d \qquad (6).$$

The distance from the current bit to the synchronization point will be

1, if the next bit is the first bit of any encoded letter,

2, if the decoder misses the next first bit and gets resynchronized on the following first bit,

.
.
.

i, if the decoder misses the next (i - 1) first bits and gets resynchronized on the i-th first bit, (7),

.
.

Therefore,

$$D = p\cdot 1 + (1-p)\cdot p\cdot 2 + (1-p)^2\cdot p\cdot 3 + \ldots + (1-p)^i\cdot p\cdot (i+1) + \ldots$$

$$= \sum_{i=1}^{\infty} [(1-p)^{i-1}\cdot p\cdot i]$$

$$= p \sum_{i=1}^{\infty} [i \cdot q^{(i-1)}] \qquad \text{(by (6))}$$

$$= p \sum_{i=1}^{\infty} [d/dq\ q^i]$$

$$= p\ d/dq \sum_{i=1}^{\infty} q^i$$

$$= p\ d/dq\ [1/(1-q)]$$

$$= p\ [1\ /\ (1-q)^2]$$

$$= p\cdot[1/p^{2]}$$

$$= 1\ /\ p$$

$$= d \qquad \text{Q.E.D}$$

Theorem 1 shows that the average number of characters decoded from a random starting point to the resynchronization point is equal to the average length in bits of an encoded codeword in the Huffman scheme. The last decoded symbol in this sequence is the first correct symbol, so that on the average (*d* - 1) incorrect symbols will have been produced. The model used here is the common discrete exponential stochastic distribution [Feller 1957].

Theorem 1 provides only an "average" distance, and a case that violates our assumption can be constructed. Imagine a perfectly balanced Huffman tree with all of the leaf nodes on the same level of the tree, and therefore all of the codewords being the same length. In that case, resynchronization will never take place [Gilbert and Moore 1959]: If the decoder once assumes that what is actually the second bit of a codeword is the first bit, then it will produce an incorrect decoded symbol from the incorrect codeword and, because the length of all codewords is constant, will again arrive at the incorrect second bit of the next codeword, and will repeat this same process indefinitely. However, Huffman code trees are almost never well-balanced in practice (a perfectly-balanced tree implies that no compression before packing into binary bits is possible). As a result this case is of little practical interest and there is reason to hope that the hypothesis of Theorem 1 may be approximately true in practice. In Chapter V we will find that Theorem 1 holds rather accurately in some test cases.

Example 4 shows resynchronization while decoding, where the encoded message is wrong in two places. One of the Huffman code sets that the source text "ABRACADEBRA" produces can be 0, 110, 100, 1011, 1010 and 111 for symbols A, B, C, D, E and R respectively. After the second bit of the codeword in the first "B" is lost, the decoder produces wrong symbols "E" and "A" before resynchronization occurs. After the first bit of the codeword for the symbol "E" is altered, the wrong symbols "A", "A", "D" and "A" are produced before resynchronization occurs. The average length of the codewords is calculated by

$$[\sum_{j=1}^{6} \text{(code length for symbol } a_j) \times \text{(frequency)}]$$
$$/\text{(total frequencies)}$$
$$= (4\cdot1 + 2\cdot3 + 1\cdot3 + 1\cdot4 + 1\cdot4 + 2\cdot3) \;/\; 11$$
$$= 27 \;/\; 11 \approx 2.5,$$

and, the expected number of wrong symbols before the resynchronization is

$$d - 1 = 2.5 - 1 = 1.5.$$

Example 4:

Suppose that Huffman codewords for alphabet A, B, C, D, E and R are 0, 110, 100, 1011, 1010, and 111 respectively, and that a message "ABRACADEBRA" needs to be transmitted. The underlines in the encoded message denote missing or altered bits, and the up arrows (↑), the resynchronization points.

source message	:	A	B	R	A	C	A	D	E	B	R	A
correct encoded message	:	0	$1\underline{1}0$	111	0	100	0	1011	$\underline{1}010$	110	111	0
incorrect encoded message	:	0	10	111	0	100	0	1011	0010	110	111	0
							↑				↑	
decoded message	:	A	D	E	A		A	D	AAD	A	R	A

Most natural language and computer language texts can be encoded using an average of only three or four bits per character, according to our experiments. Thus, on the average only two to three incorrect symbols will be produced from static Huffman decoding of these texts, after a transmission error occurs, before automatic resynchronization takes place. Test cases and their analysis are considered in Chapter V.

CHAPTER V

PERFORMANCE ANALYSIS AND COMPARISONS

Choosing Test Data

We tested twenty message ensembles: eight PASCAL program files, eleven C program files, and one English text file. The sizes of the files vary from 92 bytes to about 59K bytes (1K = 1024). Files of small sizes are chosen to demonstrate that some of the methods will not perform well in the initial stage of compression.

We used one English text file to test resynchronization on static Huffman coding and one PASCAL program source code, which is the largest in volume, to test the speed of the various basic methods.

Measure of Performance

The primary measure of performance is the compression ratio, which is calculated by

$$C1 = [(\text{size of input} - \text{size of output}) / \text{size of input}] \times 100\%, \qquad (8)$$

and it shows the percentage by which the file size has been reduced. The larger the compression ratio is, the better the compression. Some other authors such as Fiala and Greene [1989] define the compression ratio as

$$C2 = [\text{size of output} / \text{size of input}] \times 100\%, \qquad (9)$$

which is essentially a "lack-of-compression ratio" rather than a compression ratio. The relationship between two compression ratios defined by (8) and (9) is

$$100\% - C1 = C2.$$

According to our definition, 0% compression ratio means that the size of the text file is not changed by compression, and 100% means that the size of output becomes zero (even though this is not realistic).

The compression ratios yielded by each method is shown in the tables (Tables 4, 5, 6, and 7) in later section. The tables contain the names of the files, the sizes of the files before compression in bytes, the compression ratios, and the average compression ratio of each method.

The prefixes in the file names of "PAS", "C", and "TXT" represent PASCAL program files, C program files, and English files, respectively. The file C7 uses mostly tab characters to achieve indentation, while all other program files use blanks for the same purpose.

Whenever the actual size of a compressed file is known, we try to use actual numbers for the compression ratio, but when the software used for the tests did not produce an output file (when the size of file does not decrease), we used "N/A". So if "N/A" is shown in place of the size of the compressed file, it means that the size of the file has not been decreased but has actually been increased by the compression. "AVRG-RATIO" is the average ratio of each method performed on the seventeen files that are large enough (larger than 600 bytes) to show positive compression ratios for every method.

All of the test cases were run on UNIX, and some of methods were run on an MS-DOS system too to compare the results. In every case the length of each compressed file was exactly the same when run on these two systems, provided that "carriage return" characters were interpreted the same way in both systems to make the input files identical. (Normally, MS-DOS adds a "line feed" character after each "carriage return".)

The secondary measure of performance is execution time of each method on the UNIX system. Table 8 shows the results from the tests of execution time. Absolute execution times are hard to measure because the system load is changing constantly. For execution time, we tested seven methods once one after another using the UNIX utility *time*, and repeated this four more times, and averaged the execution times over five tests.

Description of Methods

We programmed static Huffman coding straightforwardly as described in Chapter III. For adaptive Huffman coding, the UNIX utility *pack* was used.

We programmed the BSTW method using a linked-list as an auxiliary data structure, which can store a word (of letters or non-letters) of up to 100 characters in each node in the list. We tested two sizes of lists that contain 255 and 127 nodes, respectively. The number 255 is the maximum size of list that allows one ASCII character to be used as an end-of-word character, which has ASCII value 255. When the size of the list is 127, only 7 bits are needed to represent a position in the list. Instead of using an end-of-word character, the first bit can be used as a flag to denote whether the next character string is a word or a position number. Table 5 shows the results from the tests where the size of the lists are 255 and 127, and when the flag is used when the size is 127.

For arithmetic coding, both fixed and adaptive implementations, we used the program by Witten et al. [Witten et al. 1987], and for the Ziv-Lempel method, we used the UNIX utility *compress*, which is an LZW implementation.

Two other techniques which combine more than one method described above were also tested. Since the output from the BSTW method is byte-oriented, we also applied another method, static Huffman coding, that takes byte-oriented output from the BSTW method as input and compressed it further.

In program source files, indentation for readability such as in nested loops takes up a lot of space. Since most of the test files we used are program source files we also tried a method that substitutes for consecutive blanks an illegal character followed by the number of blanks, as described in Chapter I, before a compression method was applied.

In testing the resynchronization model, in order to calculate the number of incorrect characters produced before the resynchronization from any random bit position, the encoded message was decoded starting from each bit to the next synchronization point. The synchronization points are the beginning bit positions of successive symbols in the encoded message, and were prestored by the compression routine in an external file. For instance, while the message "ABRACADEBRA" in example 6 is compressed, the starting bit positions 0, 1, 4, 7, 8, 11, 12, 16, 20, 23, 26 are stored externally so that the decoder can use them as a synchronization index. While decoding, the decoder keeps on checking whether or not the current first bit position is in the index list, while proceeding one bit at a time. If the current first bit is found in the list, that is a resynchronization point.

Comparisons

Results from tests of the compression methods mentioned above are shown in Tables 4, 5, 6, and 7. The first column in the table shows file names and the SIZE column gives the input file sizes in bytes. All other columns show the compression ratios when each method on the top row was applied to the files in the leftmost column. On the top row of the table, "S.H" denotes static Huffman coding, "A.H", adaptive Huffman coding, "BSTW (255)", the BSTW method with list size

255, "F.A", fixed arithmetic coding, "A.A", adaptive arithmetic coding and "Z.L", the Ziv-Lempel method. "B.H (255)" means that the BSTW method with list size 255 was applied and then to the result of it, static Huffman coding was applied.

The code table that needs to be transferred from the encoder to the decoder in static Huffman coding is saved by the encoder externally, and therefore the size of the code table is not counted into the compression ratio results. The size of the code table is a constant, and when the file is big enough, its size is negligible compared to the file itself.

As shown in Table 4, the Ziv-Lempel method shows a better compression ratio than any other single or composite method tested. As a single method, static Huffman coding performed the second best, but even the combination of the BSTW method and static Huffman did not do any better than the Ziv-Lempel method (and even with substituting for blanks, as we will see later in this chapter). The fact that the BSTW method [Bentley et al. 1986] was developed after the LZW implementation [Welch, 1984] was, and that Ziv-Lempel still outperforms BSTW, is rather surprising and justifies the rather contentious claims of Horspool [Horspool 1986]. The BSTW method suggests an interesting way of reflecting locality in data. And the fact that the encoded message is byte-oriented makes it somewhat easier to program than some other methods that result in bit-oriented encoded messages. But this method alone does not yield satisfactory results.

TABLE 4

COMPRESSION RATIOS(%)
(% REDUCTION)

FILE NAME	SIZE (BYTES)	S.H	A.H	BSTW (255)	F.A	A.A	Z.L	B.H (255)
PAS1	92	42.4%	N/A	-40.2%	22.8%	10.9%	2.2%	10.9%
PAS2	184	44.0%	N/A	-17.4%	25.5%	19.6%	18.5%	21.7%
PAS3	640	45.2%	34.8%	4.8%	28.9%	32.5%	35.2%	33.4%
PAS4	666	44.6%	35.1%	15.5%	26.0%	32.1%	38.6%	41.1%
PAS5	794	47.5%	39.9%	18.1%	32.3%	36.5%	45.3%	42.3%
PAS6	1052	44.0%	37.4%	20.5%	26.5%	34.8%	41.4%	42.2%
PAS7	34457	52.2%	51.9%	58.0%	38.8%	51.8%	71.8%	69.5%
PAS8	59003	54.7%	54.5%	66.5%	39.8%	54.6%	75.2%	75.1%
C1	361	43.2%	N/A	-10.0%	32.1%	26.0%	29.6%	26.6%
C2	3507	49.2%	46.8%	27.7%	36.1%	45.8%	59.7%	52.5%
C3	4589	39.0%	36.7%	15.1%	26.3%	36.3%	46.5%	38.5%
C4	7123	48.7%	47.4%	39.0%	26.0%	46.7%	62.8%	54.8%
C5	12214	44.1%	43.2%	32.0%	34.3%	43.3%	57.3%	50.3%
C6	12578	42.8%	41.9%	27.3%	31.7%	41.9%	59.2%	49.2%
C7	13238	43.6%	42.9%	50.7%	3.1%	42.5%	64.4%	61.2%
C8	13348	42.2%	41.4%	42.3%	31.8%	41.1%	61.4%	55.7%
C9	13784	39.0%	38.2%	41.8%	21.6%	38.0%	60.1%	55.1%
C10	33470	48.7%	48.4%	44.1%	36.9%	48.5%	67.2%	59.1%
C11	47650	39.7%	39.4%	31.9%	29.0%	39.7%	60.7%	50.0%
TXT1	13020	45.0%	44.3%	11.3%	44.2%	43.9%	48.8%	39.4%
AVRG-RATIO		45.3%	42.6%	32.2%	30.2%	41.8%	56.2%	51.1%

Witten et al. [1987] claims that the Ziv-Lempel method does not have great potential for compression unless raw speed is the main concern, but nothing is further from the truth. Arithmetic coding performs about the same as static Huffman coding does, at best. When the fixed implementation of arithmetic coding was used, most of the files show compression ratio between 25% and 40%, but file C7 shows extremely poor performance. The fact that file C7 contains many tab characters for indentation hurts its compression ratio because the probabilities used in this fixed model are based on the probabilities of English text files, which normally don't use many tab characters in proportion to the whole text. In the test of fixed arithmetic coding, the English text file TXT1 showed the best performance, as would be expected.

Table 5 shows the result from the BSTW method with two different sizes of buffers, 255 and 127, with and without a flag bit when the size of buffer is 127. The result shows that the larger buffer yields a better compression ratio unless the overhead is avoided by using a flag bit.

Table 6 shows the result of substituting for consecutive blanks by a special character followed by the number of blanks, before applying each compression method. Slightly better compression ratios are achieved for most of the methods, comprared to the result from the tests without substituting for the blanks before the compression. This combining of methods did not produce enough improvement to make the extra time spent in compression worthwhile, and hence it is not recommended.

Iterating Methods

Static Huffman coding was iterated repeatedly for each file to see how much improvement it makes. When static Huffman coding was applied twice in a row, the compression ratio increased about 3% on the average compared to when the file is compressed only once by static Huffman coding. When applied four times, approximately 4% of compression ratio increase was seen on the average. More than four times of iteration did not improve the compression ratio. Iteration of static Huffman coding is judged not to be effective, and we do not recommend it.

TABLE 5

THE RESULTS FROM THE BSTW METHOD
(% REDUCTION)

FILE	SIZE (BYTES)	255	127 no-flag	127 w/flag
PAS1	92	-40.2%	-40.2%	-19.6%
PAS2	184	-17.4%	-17.4%	0.5%
PAS3	640	4.8%	4.8%	17.3%
PAS4	666	15.5%	15.5%	25.4%
PAS5	794	18.1%	18.1%	27.2%
PAS6	1052	20.5%	20.7%	29.9%
PAS7	34457	58.0%	49.7%	53.7%
PAS8	59003	66.5%	60.2%	62.1%
C1	361	-10.0%	-10.0%	3.1%
C2	3507	27.7%	27.7%	33.1%
C3	4589	15.1%	10.7%	19.5%
C4	7123	39.0%	30.9%	37.2%
C5	12214	32.0%	26.7%	33.7%
C6	12578	27.3%	21.4%	28.0%
C7	13238	50.7%	46.0%	50.3%
C8	13348	42.3%	34.9%	40.3%
C9	13784	41.8%	36.2%	41.2%
C10	33470	44.1%	36.1%	41.7%
C11	47650	31.9%	21.7%	28.8%
TXT1	13020	11.3%	1.3%	11.1%
AVRG-RATIO		32.2%	27.2%	34.1%

TABLE 6

THE RESULT FROM SUBSTITUTING BLANKS
BEFORE COMPRESSION
(% Reduction)

FILE	SIZE (BYTES)	S.H	A.H	BSTW (255)	F.A	A.A	Z.L	B.H (255)
PAS1	92	40.2%	N/A	-40.2%	14.1%	9.8%	1.1%	8.7%
PAS2	184	39.7%	N/A	-16.3%	10.9%	17.4%	16.9%	20.7%
PAS3	640	43.4%	32.5%	8.6%	24.7%	31.9%	35.8%	34.4%
PAS4	666	42.5%	32.4%	16.7%	21.2%	30.8%	39.0%	41.3%
PAS5	794	46.4%	38.2%	23.4%	27.0%	36.0%	46.0%	44.1%
PAS6	1052	42.7%	N/A	22.4%	20.2%	34.0%	42.5%	42.5%
PAS7	34457	54.8%	54.4%	63.9%	42.5%	54.5%	72.0%	71.5%
PAS8	59003	57.8%	57.6%	71.2%	45.4%	57.9%	76.4%	76.8%
C1	361	44.9%	N/A	2.22%	32.7%	29.9%	31.0%	31.6%
C2	3507	53.6%	50.8%	44.7%	40.6%	50.3%	60.5%	57.8%
C3	4589	38.3%	35.7%	18.6%	31.5%	35.7%	46.5%	39.0%
C4	7123	53.1%	51.7%	44.9%	30.4%	51.3%	63.6%	57.4%
C5	12214	47.0%	46.0%	39.3%	35.7%	45.9%	58.0%	53.1%
C6	12578	44.6%	43.5%	40.4%	31.5%	43.7%	59.1%	53.7%
C7	13238	43.2%	42.4%	51.2%	1.7%	42.1%	64.5%	61.5%
C8	13348	42.0%	41.1%	46.5%	27.4%	41.1%	61.3%	57.5%
C9	13784	37.7%	36.8%	43.1%	17.5%	36.6%	60.1%	55.8%
C10	33470	52.9%	52.5%	50.9%	42.1%	52.7%	68.0%	61.8%
C11	47650	39.9%	39.6%	37.3%	27.6%	39.9%	60.7%	52.3%
TXT1	13020	44.3%	43.6%	11.4%	42.9%	43.1%	48.8%	39.4%
AVRG-RATIO		46.1%	43.7%*	37.3%	30.0%	42.8%	56.6%	52.9%

NOTE : "*" mark in AVRG-RATIO for the adaptive Huffman means that not all seventeen files were involved in the calculation. The file C1 was not counted because the compression ratio was not available.

TABLE 7

THE RESULT FROM ITERATING STATIC HUFFMAN METHOD
(% REDUCTION)

FILE	SIZE (BYTES)	S.H ONCE	S.H TWICE	...	S.H 4 TIMES
PAS1	92	42.4%			
PAS2	184	44.0%			
PAS3	640	45.2%	50.5%	...	58.3%
PAS4	666	44.6%	50.0%	...	57.2%
PAS5	794	47.5%	52.2%	...	57.9%
PAS6	1052	44.0%	47.2%	...	51.4%
PAS7	34457	52.2%	54.5%	...	54.5%
PAS8	59003	54.7%	56.9%	...	57.1%
C1	361	43.2%			
C2	3507	49.2%	52.8%	...	53.9%
C3	4589	39.0%	40.9%	...	41.7%
C4	7123	48.7%	52.9%	...	53.7%
C5	12214	44.1%	46.0%	...	46.2%
C6	12578	42.8%	45.3%	...	45.6%
C7	13238	43.6%	46.2%	...	46.6%
C8	13348	42.2%	44.0%	...	44.1%
C9	13784	39.0%	40.5%	...	40.6%
C10	33470	48.7%	51.0%	...	51.1%
C11	47650	39.7%	41.2%	...	41.4%
TXT1	13020	45.0%	45.9%	...	46.0%
AVRG-RATIO		45.3%	48.1%		49.8%

Table 8 shows average execution time of each method when applied on the file PAS8 five times. A UNIX utility *time* was used for each set of seven methods to get the average execution time. *compress* and *pack* are the fastest methods and static Huffman coding is next to them. The BSTW method and arithmetic coding are both eight times or more slower than the fastest methods. One should bear in mind that this implementation of the BSTW method is not a production quality code (no one, including Bentley et al., has yet claimed to have produced such a production code), and consequently the times for the BSTW method may be susceptible to great improvement. Similar excuses, however, cannot be made for the arithmetic compression routines. Arithmetic coding, at least as implemented by Witten et al., appears to be inherently slow.

TABLE 8

EXECUTION TIMES ON FILE PAS8

Methods	Execution Times
A.H	4.4 Sec.
Z.L	4.9 Sec.
S.H	12.2 Sec.
BSTW (127)	35.5 Sec.
F.A	36.9 Sec.
A.A	42.0 Sec.
BSTW (255)	46.0 Sec.

Automatic Resynchronization in Static Huffman Codes

One English text file (TXT1) was used to examine the automatic resynchronization in static Huffman decoding. The decoding process was iterated 5001 times (arbitrarily chosen large number) each such that bits 1, 2, ..., and 5001 were the starting positions for decoding, respectively. Figures

18 and 19 show the distribution of length of codewords and number of wrong characters produced, respectively. The histogram (frequency table) in Figure 19 is also plotted on a log scale in Figure 20. The value 0 in the "# of wrong chars" column in Figure 19 means that the starting position happened to be a resynchronization position, and therefore no wrong characters were produced in these cases.

code-length (in bits)	frequency
1	0
2	0
3	3513
4	5185
5	1668
6	1952
7	606
8	52
9	83
10	95
11	63
12	44
13	10
14	6
15	0

average code-length = 4.45 bits

Figure 18. Histogram of code lengths after processing 5001 bits in TXT1

For the file TXT1, the predicted average distance is (average code-length - 1) = 4.45 - 1.0 = 3.45, and the actual average distance is 3.12, and this is about 10% better than the prediction. The model obviously does not fit the situation exactly, but the agreement is good.

All of the results from tests above will be summarized in Chapter VI.

# of wrong characters (in bits)	occurences
0	1082
1	948
2	696
3	579
4	429
5	347
6	241
7	163
8	131
9	105
10	79
11	55
12	39
13	31
14	27
15	14
16	9
17	7
18	8
19	5
20	3
21	2
22	1
23	0

Figure 19. Histogram of No. of wrong chars in 5001 bits of file TXT1

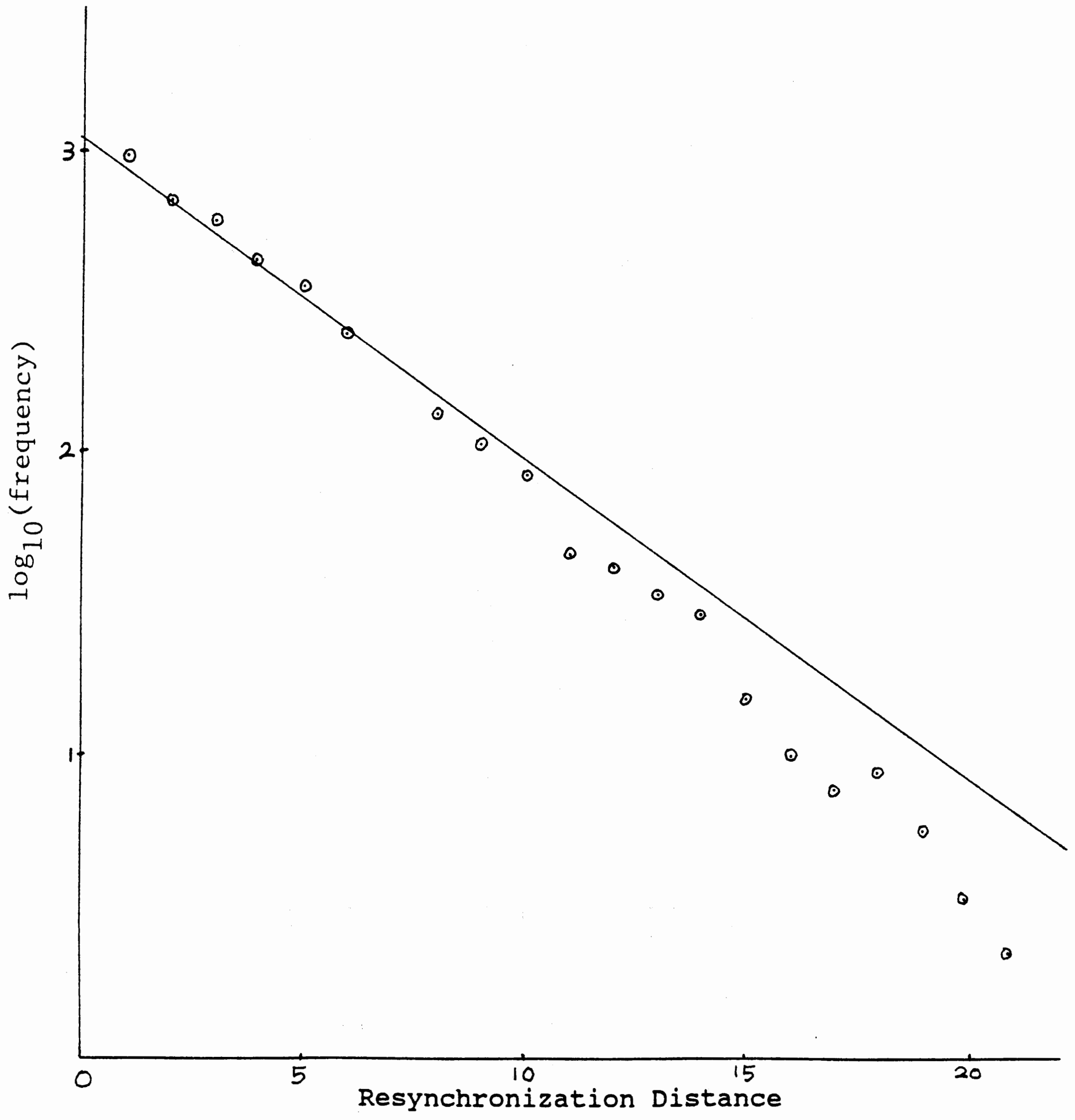


Figure 20. Semilog Plot of Resynchronization Frequencies

Comparison to Previous Results

We should compare some of our results to earlier published results. In order to do this, we will first summarize the results from the two published articles containing substantial compression results.

BSTW [Bentley et al. 1986] compressed seven C language source files having sizes from 16282 bytes to 23225 bytes, five PASCAL files of sizes 8535 bytes to 31930 bytes, one terminal session of 142762 bytes, and eight book sections of sizes 15104 to 22360 bytes. They found that compression improved as the list length N increased, flattening out around N = 32 to 256, depending on the file. For N = 256, they achieved compressions (using the definition used in this thesis) of about 55% for the terminal session, 57% to 74% for the source code file, 52% to 61% for the book sections. There was no consistent trend of greater compression for larger files or vice versa. An ordinary byte-oriented Huffman encoding achieved compression of about 33% to 43% for all files; a word-oriented Huffman scheme achieved about the same compression as did the BSTW (plus Huffman) method.

Fiala and Greene [1989] compressed 1185 source files, 134 technical memoranda files, five news service files, and a selection of object code, boot, font, and image files, using fourteen variants of several methods. They present some average compression values. Static Huffman compressed the source code files about 27% and the English language files about 40%. Dynamic Huffman performed slightly poorer. A basic Ziv-Lempel method achieved 49% and 54% on source and English, respectively. *compress* achieved 48% and 54%, BSTW-with-Huffman achieved 57% and 55%, a slow third-order Markov method by Cleary and Witten [1984b] achieved 63% and 66%, and a new Ziv-Lempel "windowing" variant by Fiala and Greene [1989] achieved 64% and 62%.

Fiala and Greene [1989] also present graphs showing the dependence of compression on file length. Each method is shown as a smooth and usually monotonic curve. Byte Huffman gives approximately a flat curve and two-byte Huffman gives a falling curve (less compression for longer files). All other methods give dramatically rising compression for larger files. These curves are inconsistent with the results of BSTW [Bentley et al. 1986]. We suspect that the curves have been heavily smoothed. The authors give no explanation of their methods of plotting and/or smoothing the curves, of the great improvement of most compression on long files, or of the poorer performance of two-byte Huffman on long files, nor do they compare their results to those of BSTW [Bentley et al. 1986].

The results from the tests in Bentley et al. [1986] show a little better compression ratio than the results from our test of BSTW-with-Huffman for similar sizes of files. Our results do not agree with the results from the tests of Fiala and Greene [1989] in many ways. It is interesting to see that the results of the same UNIX utility *compress* in the two tests are rather different. Our results for *compress* on source code files on the average show about 9% higher compression than theirs. And also their BSTW-with-Huffman performs better than *compress* with about 10% higher compression ratio, while our test shows that *compress* is a better method than BSTW-with-Huffman. Table 9 shows comparisons of results from our test, BSTW's, and Fiala and Greene's.

TABLE 9

COMPARISONS OF RESULTS WITH OTHER TESTS
(% REDUCTION)

METHODS	Static Huffman			BSTW-Huffman			*compress*	
TESTED BY	CHOI	BSTW	F&G	CHOI	BSTW	F&G	CHOI	F&G
Sources	45.3%	35.4%	26.8%	51.9%	66.7%	57.4%	56.7%	47.9%
English	45.0%	38.8%	41.0%	39.4%	55.1%	53.5%	48.8%	55.8%

These results are not the same but are not definitely incompatible either. The difference could be due to different file content and/or due to the differences in lengths of the files used for the tests. Our tests show definite improvement of compression ratios for PASCAL source files as the size of files grow, when the BSTW-Huffman, or Ziv-Lempel was used, but there is less trend, if any, for the C files. This is not completely contradictory with Fiala and Greene but the smoothness of their curve seems to have been greatly over-simplified.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE WORK

The measures of performance used in this thesis were the compression ratio and execution time on the UNIX system. The Ziv-Lempel method outperforms any other single or composite method tested. The BSTW method followed by the static Huffman method performs the second best.

Arithmetic coding does not show much strength and did not outperform static Huffman in the way claimed for this method [Witten et al. 1987]. As a single method, static Huffman showed the second best performance.

The BSTW method yields better performance with the larger buffer, but if the buffer size has to be smaller than 127, setting one bit as a flag bit is advised.

Substituting for consecutive blanks by a special character followed by the number of blanks before applying other compression method improved the compression ratios slightly, but probably not enough to make the effort worthwhile.

Iterating static Huffman coding twice increased the compression ratios by about 3% on the average, and iterating four times showed a 4% increase compared to doing static Huffman coding just once, and after that iteration made almost no difference. This iteration is not recommended for practical use.

The test of automatic resynchronization shows that the average distance from one point to the next resynchronization point in our model (D = d, where d is the average length of a codeword in bits) is a good prediction.

For most practical text compression applications, we recommend the LZW method as implemented in the public domain UNIX utility *compress*. It is fast and, on the average, outperforms all other methods tested here. (The documentation of *compress* could use considerable improvement, however.)

The recent work on text compression using Markov state models appears promising, and strong claims are being made for some of these methods [Bell and Moffat 1989]. When efficient algorithms for these methods are developed, they should be tested and compared to the methods we have tested here.

SELECTED BIBLIOGRAPHY

Aaron, J. Data Compression - A Comparison of Methods (National Bureau of Standards Special Publication 500-12), U.S. Goverment Printing Office, Washington (1977).

Abramson, N. Information Theory and Coding, McGraw-Hill, New York (1963).

Bell, T., and Moffat, A. "A Note on the DMC Data Compression Scheme", *Computer Journal 32*,1 (1989), 16-20.

Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. "A Locally Adaptive Data Compression Scheme", *Commun. ACM 29*,4 (Apr, 1986), 320-330.

Brown, R.G. Smothing, Forecasting, and Prediction of Discrete Time Series, Prentice-Hall, Englewood Cliffs, N.J. (1963).

Cleary, J.G., and Whitten, I.H. "A Comparison of Enumerative and Adaptive Codes", *IEEE Trans. on Info. Theory 30*,2 (Mar, 1984a), 306-315.

Cleary, J.G., and Witten, I.H. "Data Compression Using Adaptive Coding and Partial String Matching", *IEEE Trans. on Commun. 32*,4 (Apr, 1984b), 396-402.

Encyclopaedia Britannica, Encyclopaedea Britannica Inc., Chicago, IL (1988).

Faller, N. "An Adaptive System for Data Compression", In *Record of the 7th Asilomar Conference on Circuits, Systems and Computers* (Pacific Grove, CA), Naval Postgraduate School, Monterey, CA (Nov, 1973), 593-597.

Feller, W. AN Introduction to Probability Theory and Its Applications Vol. 1, John Wiley & Sons, New York (1957).

Fiala, E.R., and Greene, D.H. "Data Compression with Finite Windows", *Commun. ACM 32*,4 (Apr, 1989) 490-505.

Gallager, R.G. Information Theory and Reliable Communication, John Wiley & Sons, New York (1968).

Gallager, R.G. "Variations on a Theme by Huffman", *IEEE Trans. on Info. Theory 24*,6 (Nov, 1978), 668-674.

Gilbert, E.N., and Moore, E.F. "Variable Length Binary Encodings", *Bell Syst. Tech. J. 38*,4 (July, 1959), 933-967.

Hamming, R.W. "Error Detecting and Error Correcting Codes", *Bell Syst. Tech. J. 29*,2 (Apr, 1950), 147-160.

Held, G. Data Compression, John Wiley & Sons, New York (1987).

Hester, J.H., and Hirschberg, D.S. "Self-Organizing Linear Search", *Computing Surveys 17*,3 (Sept, 1985), 295-311.

Horspool, R.N. *Computing Reviews* (Oct, 1986), 518.

Huffman, D.A. "A Method for the Construction of Minimum-Redundancy Codes", *Proc. IRE 40*, 9 (Sept, 1952), 1098-1101.

Ingels, F.M. Information Theory and Coding Theory, Intext, Scranton, PA (1971).

Kahn, D. The Codebreakers, MacMillan, New York (1967).

Knuth, D.E. "Dynamic Huffman coding", *J. Algorithms 6*,2 (June, 1985), 163-180.

Knuth, D.E. The Art of Computer Programming Vol. 1, Addison-Wesley, Reading, MA (1973).

Lelewer, D.A., and Hirschberg, D.S. "Data compression", *ACM Computing Surveys 19*,3 (Sept, 1987), 261-296.

Lempel, A., and Ziv, J. "On the Complexity of Finite Sequences", *IEEE Trans. on Info. Theory 22*,1 (Jan, 1976), 75-81.

Llewellyn, J.A. "Data Compression for a Source with Markov Characteristics", *Computer Journal 30*,2 (1987), 149-156.

Lynch, Thomas J., Data Compression, Van Nostrand Reinhold, New York, NY (1985).

McIntyre, D.R., and Pechura, M.A. "Data Compression Using Static Huffman Code-Decode Tables", *Commun. ACM 28*,6 (June, 1985), 612-616.

Parker, D.S. "Conditions for Optimality of the Huffman Algorithm", *SIAM J. Comput. 9*,3 (Aug, 1980), 470-489.

Pechura, M.A. "File Archival Techniques Using Data Compression", *Commun. ACM 25*,9 (Sept, 1982), 605-609.

Reghbati, H.K. "An Overview of Data Compression Techniques", *Computer 14*,4 (Apr, 1981), 71-75.

Reingold, E. and Hansen, W., Data Structures. Little Brown Computer System Series (1983).

Rissanen, J. "Generalized Kraft Inequality and Arithmetic Coding", *IBM J. Res. Dev 20* (May, 1976), 198-203.

Rissanen, J. "A Universal Data Compression System", *IEEE Trans. on Info. Theory 29*,5 (Sept, 1983), 655-664.

Rodeh, M., Pratt, V.R., and Even, S. "Linear Algorithm for Data Compression Via String Matching", *J. ACM 28*,1 (Jan, 1981), 16-24.

Rubin, F. "Experiments in Text File Compression", *Commun. ACM 28*,1 (Jan, 1981), 617-623.

Rubin, F. "Arithmetic Stream Coding Using Fixed Precision Registers", *IEEE Trans. on Info. Theory 29*,5 (Nov, 1979), 672-675.

Rudner, B. "Construction of Minimum-Redundancy Codes with an Optimum Synchronizing Property", *IEEE Trans. on Info. Theory 17*,4 (July,1971), 478-487.

Ruth, S.S., and Kreutzer, P.J. "Data Compression for Large Business Files", *Datamation* (Sept, 1972), 62-66.

Ryabko, B.Y. "A Locally Adaptive Data Compression Scheme", *Commun. ACM 16*,2 (Sept, 1987), 792.

Standish, T.A. Data Structure Techniques, Addison-Wesley, Reading, Mass. (1980).

Storer, J.A. Data Compression. Computer Science Press, Rockville, Md (1988).

Storer, J.A. and Szymanski, T.G. "Data Compression Via Textual Substitution", *J. ACM 29*,4 (Oct, 1982), 928-951.

Vitter, J.S. "Algorithm 673 Dynamic Huffman Coding", *ACM Trans. on Math. Software 15*,2 (June, 1989), 158-167.

Vitter, J.S. "Design and Analysis of Dynamic Huffman Codes", *J. ACM 34*,4 (Oct, 1987), 825-845.

Welch, T.A. "A Technique for High-Performance Data Compression", *Computer 17*, 6 (June, 1984), 8-19.

Whitten, I.H., Neal, R.M., and Cleary, J.G. "Arithmetic Coding for Data Compression", *Commun. ACM 30*,6 (June, 1987), 520-540.

Ziv, J., and Lempel, A. "A Universal Algorithm for Sequential Data Compression", *IEEE Trans. on Info. Theory 23*,3 (May, 1977), 337-343.

Ziv, J., and Lempel, A. "Compression of Individual Sequences Via Variable-Rate Coding", *IEEE Trans. on Info. Theory 24*,5 (Sept, 1978), 530-536.

2

VITA

Sunny Choi

Candidate for the Degree of

Master of Science

Thesis: A COMPARISON OF METHODS FOR TEXT COMPRESSION

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Inchon, South Korea, September 29, 1956, the daughter of Mr. and Mrs. JangSun Choi;

Education: Graduated from In-Il Girls' High School, Inchon, South Korea, in January, 1974; received Bachelor of Science degree in Mathematics from Ewha Woman's University, Seoul, South Korea in January, 1979; completed requirements for Master of Science at Oklahoma State University in December, 1989.

Professional Experience: Systems Analyst, TMS Inc., Stillwater Oklahoma, 1987-1988.