Implementing high-order context models for statistical data compression
by Robert L Wall

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
As multimedia applications gain popularity and the public availability and connectivity of computers increase, more and more digital data is being stored and transmitted. This growing volume of data is creating an increasing demand for efficient means of reducing the size of the data while retaining all or most of its information content; this process is known as data compression. Data compression techniques can be subdivided into a number of categories; this discussion will concentrate solely on the subset of lossless text compression methods which are based on statistical modelling techniques.

As background, some basic principles of information theory as they apply to data compression are reviewed, then three different statistical compression techniques are presented — specifically, Shannon-Fano, Huffman, and arithmetic coding. The basic algorithms are described and actual implementations using the C programming language are given. Next, enhancements which allow the Huffman and arithmetic coders to adapt to changing data characteristics are examined; again, C implementations are presented. Various methods of improving the compression efficiency of these adaptive algorithms by deriving more accurate models of the data being compressed are then discussed, along with the problems which their implementations pose. All of this information is essentially a survey of the field of statistical data compression methods.

Once the background information has been presented, some of the problems involved with implementing higher-order data models are addressed. Specifically, implementations of the PPM (Prediction by Partial Match) modelling technique are presented which can run in a constrained memory space, specifically that available on an Intel processor-based personal computer running the DOS operating system. The development and implementation of PPM modelers for both Huffman and arithmetic coders is described in detail. Once the basic programs are described, additional enhancements designed to improve their compression efficiency are described and evaluated.

The compression performance of each of the algorithms developed is compared to the performance of some of the publicly available data compressors; in general, the PPM modelers with Huffman and arithmetic coders are shown to achieve nearly the performance of the best commercial compressors. Finally, further research into enhancements which would allow these programs to achieve even better performance is suggested.

IMPLEMENTING HIGH-ORDER CONTEXT MODELS FOR STATISTICAL DATA COMPRESSION

by

Robert L. Wall

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 1994

APPROVAL

of a thesis submitted by

Robert Lyle Wall

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

April 28, 1994
Date

Chairperson, Graduate Committee

Approved for the Major Department

4/28/94
Date

Head, Major Department

Approved for the College of Graduate Studies

5/10/94
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Robert L Wall

Date April 29, 1994

ACKNOWLEDGEMENTS

I would like to thank Dr. Gary Harkin for the guidance, assistance, and inspiration he has provided, not only in the completion of this document, but throughout my college career (lengthy as it has been). Thanks also to the other members of my graduate committee, Drs. Denbigh Starkey and Rockford Ross, from whom I learned a great deal. I would especially like to thank Jenn Pomnichowski for her support and tolerance while I labored on my thesis, and for her assistance in the actual production of this document; I know it wasn't easy. Thanks also to a number of people at VLC, especially Bill Procunier and Mike Magone, for financing a portion of my graduate studies and cutting me the slack I needed to get things finished up. Finally, I would like to thank the "compressorheads" who hang out on the comp.compression and comp.compression.research newsgroups on the Internet for providing a wealth of new ideas, a source of information and references, and a sounding board for new ideas.

TABLE OF CONTENTS

TABLE OF CONTENTS -- Continued

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As multimedia applications gain popularity and the public availability and connectivity of computers increase, more and more digital data is being stored and transmitted. This growing volume of data is creating an increasing demand for efficient means of reducing the size of the data while retaining all or most of its information content; this process is known as data compression. Data compression techniques can be subdivided into a number of categories; this discussion will concentrate solely on the subset of lossless text compression methods which are based on statistical modelling techniques.

As background, some basic principles of information theory as they apply to data compression are reviewed, then three different statistical compression techniques are presented -- specifically, Shannon-Fano, Huffman, and arithmetic coding. The basic algorithms are described and actual implementations using the C programming language are given. Next, enhancements which allow the Huffman and arithmetic coders to adapt to changing data characteristics are examined; again, C implementations are presented. Various methods of improving the compression efficiency of these adaptive algorithms by deriving more accurate models of the data being compressed are then discussed, along with the problems which their implementations pose. All of this information is essentially a survey of the field of statistical data compression methods.

Once the background information has been presented, some of the problems involved with implementing higher-order data models are addressed. Specifically, implementations of the PPM (Prediction by Partial Match) modelling technique are presented which can run in a constrained memory space, specifically that available on an Intel processor-based personal computer running the DOS operating system. The development and implementation of PPM modelers for both Huffman and arithmetic coders is described in detail. Once the basic programs are described, additional enhancements designed to improve their compression efficiency are described and evaluated.

The compression performance of each of the algorithms developed is compared to the performance of some of the publicly available data compressors; in general, the PPM modelers with Huffman and arithmetic coders are shown to achieve nearly the performance of the best commercial compressors. Finally, further research into enhancements which would allow these programs to achieve even better performance is suggested.

CHAPTER 1

INTRODUCTION

With the advent of the Information Superhighway, the evolution of multimedia applications, and the increasing availability of computing resources and connectivity between computing sites, the volume of data that is being stored and transmitted by digital computers is increasing dramatically. The amount of digitally stored text, electronic mail, image and sound data, and executable binary images being accessed, archived, and transferred over bandwidth-limited channels has continued to outpace technological improvements in data storage and communication capacity (or at least cost-effective and readily available improvements), demanding some means of reducing storage space and transmission time requirements. Methods are desired which can compact data before it is stored or transmitted and which can subsequently expand the data when it is accessed or received. Such data reduction or compaction is typically accomplished by some form of *data compression.*

Data Compression Definitions

Data compression is defined as "the process of encoding a body of data D into a smaller body of data $\Delta(D)$. It must be possible for $\Delta(D)$ to be decoded back to D or some acceptable approximation of D." ([STORER88], p. 1) The objective of compressing a message is to minimize the number of symbols (typically binary digits, or bits) required to represent it, while still allowing it to be reconstructed. Compression techniques exploit the redundancy in a message, representing redundant portions in a form that requires fewer bits than the original encoding. A reverse transformation, the decompression technique, is then applied to this reduced message to recover the original information or an approximation thereof.

Compression techniques are typically categorized into many different subdivisions. The division of the highest scope is between techniques which are applicable to digital data processing and signal encoding

techniques used in communications. Digital techniques include text compression and compression of digitally sampled analog data (although the two are not mutually exclusive; text compression algorithms are often successfully applied to two-dimensional image data). The primary difference between digital techniques, and text compression in particular, and compression of communications signals is that digital compressors typically do not have a well-defined statistical model of the data source to be compressed which can be tuned to optimize performance. It is thus necessary for the compression method to determine a model of the data source and compute probability distributions for the symbols in each data message. Communications signals, on the other hand, are typically produced by a source having a well-defined model, such as the model of the human vocal tract which produces voice signals, and can therefore be carefully tuned to perform well on that data.

Another division suggested above is between methods which are applied to text data, such as standard ASCII text files, source code, and binary object and executable image files, and those applied to digitally sampled analog data, such as image or sound data; a closely related categorization is between *lossless* and *lossy* methods. As their names suggest, lossless techniques allow the exact reconstruction of the original message from the compressed data, while lossy methods do not. Lossless methods are most appropriate, and usually essential, for text compression applications, where it is not acceptable to restore approximately the same data (for example, the readability of this thesis would deteriorate rapidly if one character in fifty were changed each time it was archived and extracted). Lossy methods are more typically used on digitally sampled analog data, where a good approximation is often sufficient. An example would be the compression of a digitized audio signal; due to imperfections of the human ear, the loss of a small amount of information would probably be unnoticeable. Another example is the new Joint Photographic Expert Group's (JPEG) proposed standard for compressing two dimensional image data. Since the restriction on exact recovery of the data has been relaxed, lossy techniques will usually achieve greater amounts of compression. For example, lossless text compression methods typically reduce English text to between 40% and 70% of its original size, with some schemes approaching 25%; the best reduction possible is estimated to be no less than about 12% ([BELL90], p. 13). On image data, some text compression schemes can approach a 50% reduction,

depending on the content of the image. JPEG, on the other hand, can achieve reduction to 2% or smaller without severely degrading the quality of the decompressed image [WALLACE]. This thesis will concentrate exclusively on lossless text compression methods, since they are applicable to a more general class of data storage and communication applications.

A common division of text compression methods is into *ad hoc*, *statistical*, and *dictionary-based* schemes. As the name implies, ad hoc methods are created to solve a very specific problem, and they are typically not well suited to a wide variety of situations. Both statistical and dictionary-based methods are generalized data compression algorithms which can be used on a wide variety of data types with suitable effectiveness; in the latter, a dictionary of phrases which might be seen in the input is used or generated, and occurrences of these phrases are replaced by references to their position in the dictionary. In the former, a set of variable length codes is constructed to represent the symbols in a message based on the probability distribution of those symbols. Both categories have been widely explored, and in fact the dictionary-based algorithms are most often used in today's popular compression programs; however, statistical methods are also well understood, and they may have the potential for greater improvement of compression efficiency. Also, it has been proven that any dictionary-based scheme has an equivalent statistical method which will achieve the same compression ([BELL90], chapter 9, and [KWAN]). This discussion therefore focuses on statistical compression techniques.

Another distinction frequently made is between *static*, *semi-adaptive*, and *adaptive* techniques; this categorization refers to the method used to determine the statistical characteristics of the data. The basis of all compression algorithms is essentially the determination of the probability of a symbol or string of symbols appearing in the source message, and the replacement of high probability symbols or strings with short code representations and low probability symbols or strings with longer code representations; this is discussed in more detail in the following section on information theory. Static (non-adaptive) algorithms assume an *a priori* probability distribution of the symbols in order to assign codes to them. These types of algorithms typically suffer badly in compression ratio if the statistical character of the input data is substantially different from the assumed statistical model. Adaptive algorithms make an initial assumption about the

symbol probabilities and adjust that assumption as the message is processed, while semi-adaptive algorithms make an initial pass over the data to extract accurate probability information for the compression of that data. This probability information is then fixed for the compression of the message. The latter methods suffer in execution speed, since two passes must be made through the data, and they are not applicable for stream-based environments, where it may not be possible to make two full scans of the data. Also, their performance on small files may be adversely affected by the amount of probability information which must be passed to the decoder. The former methods require only one pass through the data and will adjust to messages with probability distributions different than those originally assumed. A small percentage of the optimal compression efficiency may be sacrificed as the compressor adapts its statistics to match those of the data, but the adaptive techniques are much more robust than the other methods. They also do not need to prepend any statistical data on the compressed data, which may offset their initially poorer compression efficiency. Both semi-adaptive and adaptive methods are presented in this thesis.

Information Theory Fundamentals

Most research into data compression algorithms has its foundation in the field of information theory, which was pioneered in the late 1940s. Claude Elwood Shannon of Bell Labs published a paper, *The Mathematical Theory of Communications*, in the Bell System Technical Journal in 1948 ([SHANNON] includes a reprint of this paper); it presented the framework from which most information theory research sprang. The following subsections provide a brief overview of pertinent terms and associated standard symbols used in information theory and discuss some important theorems which have been proven.

Terminology and Definitions

Any message to be compressed is assumed to have been produced by a message source s with certain characteristics. The first of these is the source alphabet, S, which is the set of all symbols $\{S_i\}$ which can be produced by the source. The size of the alphabet is defined as $q = |S|$, the number of symbols in S. S^n is the set of all strings of n symbols which can be formed from S. A definition of the message source s is thus a subset of S^*, the Kleene closure of S, which is the set of all S^n for all integer values of n from zero to infinity.

The source alphabet S must be represented or encoded in some form in order to be stored on a computer. $C(S_i)$ refers to the code corresponding to symbol S_i in a particular representation of S, and $C(S)$ is the collection of codes representing the entire alphabet. These codes are constructed of strings created from a code alphabet X, the set of all code symbols $\{X_i\}$. The size of the code alphabet, $r = |X|$, is also referred to as the *radix* of the code alphabet. $l_i = |C(S_i)|$ is the length of the ith code word, or the number of symbols from X required to form $C(S_i)$, and L is the average length of all code words $C(S)$; this is defined as $\sum p_i l_i$, where p_i is the probability of symbol i being produced (as is explained in more detail in the next section). As an example, computer data is typically represented using the ASCII character set; since this data is stored on binary computers, $X = \{0, 1\}$ and $r = 2$. The ASCII set defines 128 fixed length codes of seven bits, so $q = 128$, $l_i = 7$ for i in $[1, q]$, and $L = 7$ bits, but most computers actually use an extended set with $q = 256$ and $L = 8$ bits. The goal of a compression algorithm is therefore to find an alternate coding for an ASCII message which will produce an average code length L which is less than eight bits.

Modelling of an Information Source

In general, a message source s is not fully characterized by S. As mentioned previously, s is a subset of S^*, and additional information is required to describe which strings are actually included in the subset. This typically requires that some assumptions be made about the nature of s; a common one is that s can be modelled as a *Markov process* of order m. An mth-order Markov source can best be depicted as a state diagram with q^m states, each of which can have up to q transitions to other states; each transition is labeled by a symbol S_i and the corresponding probability p that the transition will be taken (i.e., that symbol will be produced) while in that state. Figs. 1 and 2 show order-0 and order-1 examples of this, for $S = \{a, b, c\}$.

If s is assumed to be a Markov process, it can thus be characterized by S and the probability information describing the transitions. For an order-0, or memoryless, source, this information is just $P = \{p_i\}$, the set of probabilities that each symbol will be produced, where $\sum_{i=1-q} p_i = 1$. For a higher-order source, these probabilities become the conditional probabilities $p_i = \sum_{j=1-q} P_j\, p\,(i \mid j)$, where P_j is the probability of being in state j and $p\,(i \mid j)$ is the probability that symbol i will be produced following symbol j. This idea can be further extended to higher order models using similar conditional probabilities.

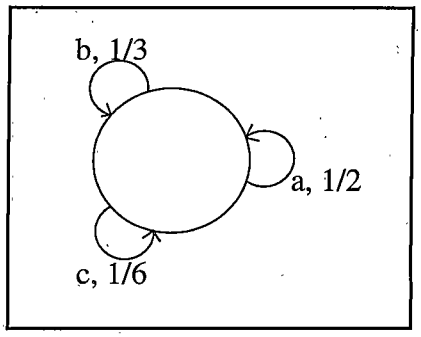

Figure 1: Example of Order-0 Markov Source

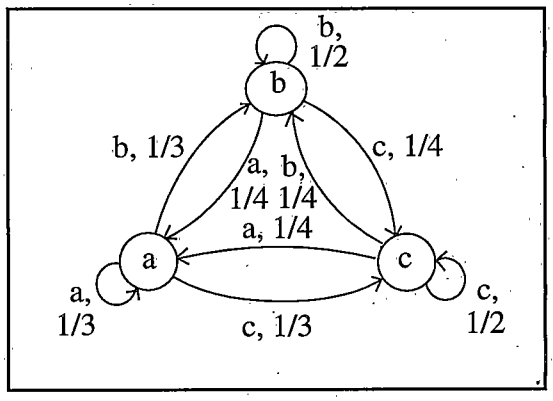

Figure 2: Example of Order-1 Markov Source

A further assumption usually made about a Markov source is that it is *ergodic*; that is, it is assumed that from any state, it is possible to get to any other state, and that in time the system settles down to a steady state with a probability distribution independent of the initial state ([HAMMING]). Another way of stating this is that it is possible to determine the complete probability information of an ergodic by examining any suitably long string produced by that source. The assumption of ergodicity is made because it significantly simplifies any computations based on the transition probabilities, since they can be assumed to be constant and independent of the initial state of the source.

Entropy

A concept that is fundamental to information theory is that of *entropy*, which is a measure of the information content in any message produced by *s*. This is directly related to the randomness of the source;

for example, if a machine produces messages that are always a single binary zero, there is very little information contained in each of these messages. This is because the source is not random, so it is known in advance what it will do.

The formal definition of entropy as it relates to information theory was presented by Shannon in his groundbreaking paper. It is stated as follows: given a message source characterized by S and P, the entropy of the source is defined as

$$H_r(S) = \sum p_i \log_r (1 / p_i) \qquad \text{(or equivalently as } H_r(S) = \sum - p_i \log_r (p_i)\text{)}$$

where r is the radix of the measurement. Typically, if r is not defined, it is assumed to be two; this indicates that the information content is measured in bits.

In the same paper, Shannon presented his *fundamental coding theorem for a noiseless channel*, the proof of which shows that the average length of any encoding of the symbols of S cannot be less than the entropy of S. That is, L can asymptotically approach $H(S)$ but cannot exceed it, or $H_r(S) \leq L$ for any $C(S)$. It must be noticed that in this presentation, the calculation of the entropy, and thus the lower bound on average code length, is dependent on the model used to describe the source. For example, the entropy of a given source will be different if calculated using an order-0 model than that calculated using an order-1 model. It might be possible for a compressor which uses an order-1 model to produce codes whose average length is less than the order-0 entropy of the source, but this average length cannot be less than the entropy calculated using the same order-1 probabilities.

For a more detailed discussion of the preceding information, see [ABRAMSON], [COVER91], [HAMMING] or other books on information theory.

Measuring Performance of Compression Programs

There are three primary means by which implementations of compression algorithms (and the associated decompressors) are evaluated. These include the speed at which they compress or decompress data, the memory which they use, and their compression efficiency. The speed of a compressor is typically measured by the number of input bytes consumed per second, while the speed of the decompressor is measured

by the number of output bytes produced per second. The measurement of memory usage is fairly straightforward, consisting of the maximum number of bytes required by the program while processing a message. This maximum is often a fixed value for a particular execution of a program, although it may be a parameter which can be adjusted.

Compression efficiency is a measure of the effectiveness of the compressor in removing the redundancy from a message. It can be expressed in a number of ways, each of them typically involving a ratio of the size of the output message to the size of the input message. Obvious measures are the size of the output divided by the size of the input, and the inverse ratio. This thesis uses a related compression ratio, defined as follows:

ratio = (1 - output_size / input_size) * 100

This expresses the compression efficiency as a percentage, with zero percent indicating no compression (the output size is the same as the input size) and 100 percent indicating perfect compression (the output size is zero). Note that it is possible to achieve a negative compression ratio, if the program actually expands the data (the output size is greater than the input size).

As with most digital algorithms, there are definite trade-offs between these three performance measures within a compression program. For instance, if an algorithm exhibits superior compression efficiency, it typically requires either extensive memory or increased execution time, and either one or the both of these must be increased to improve the algorithm's efficiency. Likewise, execution time for some algorithms may be reduced by modifying their data structures to make less efficient use of memory. A great deal of attention is typically given to compression algorithms' performance as measured by execution time and compression efficiency, with memory usage a secondary consideration; however, the algorithms presented in this thesis have instead been tuned to decrease memory usage and increase compression efficiency, where possible. If an algorithm's execution time can be reduced without adversely affecting one of the other factors, the algorithm will be tuned appropriately; however, the focus is on creating compressors which can produce a level of compression efficiency comparable to that of other popular compression programs, while executing in a reasonable amount of memory. The target for memory usage is to constrain each program to be able to exe-

cute in a limited memory space, specifically, that available on an Intel processor-based personal computer running the DOS operating system. This implies that the program can use no more than about 450 Kbytes of memory for data, and that each data structure or array of data elements must fit within a 64 Kbyte segment.

Benchmarking Compression Algorithms

It is usually important when comparing various compression algorithms to evaluate their performance on a variety of input types. A standard set of files has been assembled to demonstrate the strengths and weaknesses of various compression algorithms. This set of files, known as the Calgary compression corpus or the BWC corpus, was first presented in the book Text Compression ([BELL90], Appendix B). It is available via anonymous FTP from the Internet site *fsa.cpsc.ucalgary.ca*, and it has become the standard benchmark for measuring the efficiency of text compression programs. It is used in this thesis to compare the performance of the various programs, and to compare them to other publicly available compressors.

The corpus consists of the files listed in Table 1. It is used in a number of places throughout this thesis when the performance of the various compression algorithms are being evaluated.

File Name	Description
bib	A list of references for comp.sci.papers in UNIX 'refer' format
book1	"Far from the Madding Crowd" by Thomas Hardy, plain ASCII
book2	"Principles of Computer Speech" by Ian Witten, Unix troff format
geo	Geophysical data consisting of 32-bit numbers.
news	Usenet news
obj1	VAX executable
obj2	Mac executable
paper1	Technical paper, Unix troff format
paper2	Technical paper, Unix troff format
paper3	Technical paper, Unix troff format
paper4	Technical paper, Unix troff format
paper5	Technical paper, Unix troff format
paper6	Technical paper, Unix troff format
pic	CCITT fax test picture 5 (1728x2376 bitmap, 200 pixels per inch)
progc	C program
progl	Lisp program
progp	Pascal program
trans	Transcript of terminal session (text + ANS1I control codes)

Table 1: Files included in the Calgary Compression Corpus

Source Code for Compression Programs

The remainder of this thesis presents a number of lossless statistical text compression algorithms and discusses some of the details of their implementation in the C programming language. Anyone wishing to receive a copy of this code can send an e-mail request to *thesis-request@cs.montana.edu*; in response, a uuencoded archive file in the ZIP format will be returned. This archive contains the following files:

• common.h	Defines a number of useful data types and constants
• bit_io.h	Defines types and interface for library of bit-oriented I/O functions
• compress.h	Defines external functions and variables provided in computil.h and those required of the compression module
• encode.c	Contains main entry point for compressor programs - processes arguments, opens files, and calls compression routine
• decode.c	Contains main entry point for decompressor program - processes arguments, opens files, and calls decompression routine
• computil.c	Contains utility routines used by semi-adaptive programs
• bit_io.c	Contains bit-oriented I/O function library
• shanfano.c	Compression / decompression routines to perform semi-adaptive Shannon-Fano coding
• huffman.c	Compression / decompression routines to perform semi-adaptiveHuffman coding
• arith.c	Compression / decompression routines to perform semi-adaptive arithmetic coding
• adap_huf.c	Compression / decompression routines to perform adaptive Huffman coding
• adap_ari.c	Compression / decompression routines to perform adaptive arithmetic coding without escape symbols
• adap_are.c	Compression / decompression routines to perform adaptive arithmetic coding with escape symbols
• ahuff_n.h	Data types and structure definitions for ahuff_n.c
• ahuff_n.c	Compression / decompression routines to perform order-n adaptive Huffman coding
• ahn[1-6].c	Various data structures for use with ahuff_n.c
• aarith_n.h	Data types and structure definitions for aarith_n.c
• aarith_n.c	Compression / decompression routines to perform order-n adaptive arithmetic coding
• aan[1-5].c	Various data structures for use with aarith_n.c

CHAPTER 2

STATISTICAL COMPRESSION ALGORITHMS

As mentioned in the first chapter, this thesis concentrates solely on lossless statistical text compression methods. Statistical methods are those which use the probability information discussed in the section on the modeling of information sources in Chapter 1 to assign codes to the symbols of a source alphabet. This chapter contains sections describing the three most familiar forms of statistical coding -- Shannon-Fano, Huffman, and arithmetic codes. Each section discusses the algorithms required to generate the codes for a given source, to encode a source message using those codes, and to decode a stream of codes to reproduce the original source message. Details of the implementation of those algorithms using the C programming language are also discussed.

Each of the algorithms discussed in this chapter is a semi-adaptive method; that is, the encoder must make a first pass over the input data file to accumulate the required statistics, and it must then save those statistics as the first part of the compressed data file. The decoder must read these statistics from its input file in order to have the same statistics that the compressor used to build the coding model. Also, each of the algorithms is an order-0 model, so the only statistics required are the relative frequencies of occurrence of the individual symbols in the input file, independent of the context in which they appeared.

Shannon-Fano Coding

Part of Shannon's proof of his fundamental noiseless coding theorem was the presentation of a method of coding a source's messages which did approach the entropy of the source. He also noted that R. M. Fano of M.I.T. had independently found essentially the same method; for this reason, the coding technique is now known as Shannon-Fano coding. In addition to proving that the technique produced codes

whose average length L approached the entropy of the source, he also proved an upper bound on the average code length, given by the equation

$$H_r(S) \leq L < H_r(S) + 1$$

That is, he showed that while L only asymptotically approaches the source's entropy, it cannot vary from it by more than one (i.e., one bit, if the radix used is two).

Encoding Algorithm

The primary step required in the encoding of a message using Shannon-Fano codes is the assignment of a unique code to each of the source symbols which occur in the message; the encoding of the message is then simply a matter of translating from the ASCII code for each symbol to the new code and outputting it. The codes are of varying length; in order to achieve compression, this algorithm must obviously code some symbols with codes shorter than eight bits, and it is therefore necessary to code other symbols with codes longer than eight bits. Intuitively, in order for the output to be smaller than the input, the shorter codes should be assigned to symbols which occur more frequently, and conversely, the longer codes should be assigned to symbols which occur infrequently. The semi-adaptive Shannon-Fano algorithm to perform the required assignment is as follows:

1. Generate a list of symbols occurring in the input, along with the relative frequency of each (i.e., the count of the number of times each symbol occurred).
2. Sort the list by frequency (either ascending or descending).
3. Divide the list into two parts, with the total frequency of the symbols in each half being as close to equal as possible.
4. Assign bit 0 as the most significant bit (MSB) of the symbols in the upper half of the list, and bit 1 as the MSB of the symbols in the lower half.
5. Recursively apply steps 3 and 4 to each half of the list, adding the next most significant bit to each symbol's code until every list contains only one symbol.

Consider the following example, for a source alphabet $S = \{a, b, c, d\}$. Since $q = 4$, $\log_2 q = 2$ bits would normally be required to represent each code. However, application of the algorithm to the message "aaaaaaaaaaccccccbbbdd" would produce the coding shown in Fig. 3 (on the next page). It can be seen that this results in a total coded message length of 37 bits, rather than the 42 bits required by the standard encoding; this corresponds to a compression ratio of approximately 12%.

Step 1		Step 2		Steps 3 and 4 (repeated recursively)				
Symbol / Frequency		Symbol / Frequency		Symbol / Frequency				
a	10	a	10	a	10	0		
b	3	c	6	c	6	1	0	
c	6	b	3	b	3	1	1	0
d	2	d	2	d	2	1	1	1
Total	21							

Resulting codes

a	0
b	110
c	10
d	111

Figure 3: Example of Shannon-Fano Code Generation

This example also illustrates the reason why the average Shannon-Fano code length can exceed the source's entropy. If this message is representative of the source's probability characteristics, P is given by $\{p_a = 10/21, p_b = 3/21, p_c = 6/21, p_d = 2/21\}$. The entropy $H(S)$ is given by

$$H(S) = -\ 10/21 \log_2 10/21 - 3/21 \log_2 3/21 - 6/21 \log_2 6/21 - 2/21 \log_2 2/21 = 1.75 \text{ bits/symbol}$$

while the average code length L is given by

$$L = 10/21\ (1) + 3/21\ (3) + 6/21\ (2) + 2/21\ (3) = 1.76 \text{ bits/symbol}$$

Although the difference is small, the average length is greater than the entropy. The reason is that the necessity of assigning an integral number of bits to each code makes that code's length different than the optimal length. Equating the equations for entropy and average code length shows that the optimal l_i is equal to $-\log_2 p_i$; for this example, that gives $l_a = 1.07$, $l_b = 2.81$, $l_c = 1.81$, and $l_d = 3.39$. The differences between these optimal lengths and the integral lengths assigned by the algorithm account for the variation between the average code length and the entropy. It can be seen from this that if the probabilities in P are all integral powers of 1/2, L will be equal to $H(S)$ (since the optimal lengths will all be integers). This is in fact the only case in which Shannon-Fano coding, or any other coding scheme which assigns an integral number of bits to each code, performs optimally.

Decoding Algorithm

In order to retrieve the original message, the decoder must process the stream of bits output by the encoder, recover each code word, and perform the inverse mapping from code symbol to appropriate source symbol. Once the source symbol is recovered, its standard ASCII representation can be output. If the decoder is to be able to retrieve each code word from the compressed data, the code must be *uniquely decodable*; that is, it must be possible to uniquely determine which code word is next in the compressed data stream. In order for this to be true, the code must exhibit the *prefix property*, which requires that no code word be a prefix of any other code word. Due to the method used to create the Shannon-Fano codes, this is indeed the case. This is made more obvious by viewing the process of creating the codes as the creation of a binary tree; each time a list is divided into two parts, this is analogous to creating two child nodes of the current node representing the list and assigning all of the symbols in the first half of the list to the left child and all nodes in the last half of the list to the right child. The tree which matches the example of Fig. 3 is shown in Fig. 4.

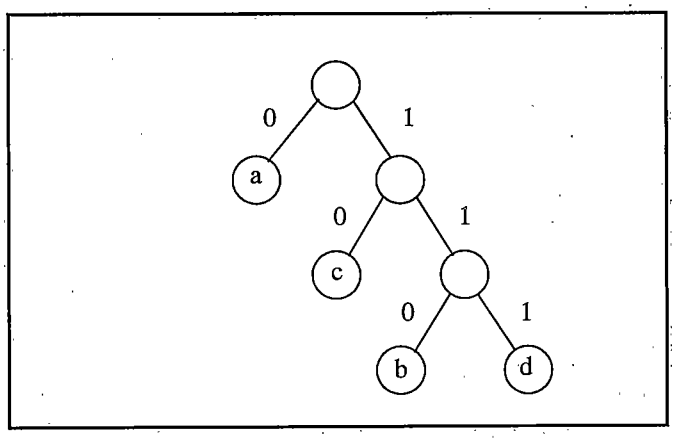

Figure 4: Tree Corresponding to Previous Shannon-Fano Example

This tree can be used to generate the Shannon-Fano codes; a recursive traversal of the tree is performed, accumulating bits as each child node is visited (a 0 for the left child and a 1 for the right child) until a node corresponding to a symbol is reached. The symbol in that node is then assigned the code accumulated to that point. It can be seen that the code thus generated will only exhibit the prefix property if all of the sym-

bol nodes are also leaf nodes. This is guaranteed in the Shannon-Fano algorithm by the fact that any list which contains only one symbol can no longer be subdivided, and is therefore represented by a leaf node in the tree. The codes generated by the Shannon-Fano algorithm are therefore uniquely decodable.

The process of decoding uses the tree representation of the codes as well. The decoder creates the same tree that the encoder used by processing the set of symbol frequencies prepended to the compressed data stream and using the identical algorithm to construct the tree. Once the tree has been built, the decompressor begins at the tree's root and begins processing input data one bit at a time. With each bit read, the decompressor will visit either the left or right child of the current node, depending on the value of the bit. When a leaf node is reached, the symbol corresponding to that node is output, and the current node is reset to the tree's root. This process continues until the compressed data is exhausted.

A final problem which must be addressed is that of actually determining the end of the compressed data. If the compressed data were actually stored as a string of bits, this would not be difficult; however, the smallest storage unit provided by most computers is an eight-bit byte, so if the length of the compressed data is not an integral multiple of eight bits, additional bits must be added on the end to pad out the last byte. This can potentially cause problems in decoding, since the decoder cannot differentiate between these pad bits and compressed data bits, so it might erroneously produce extra bytes at the end of the decoded message. One means of handling this is the inclusion of the length of the compressed data (in bits) at the start of the compressed file; this requires that the decoder keep track of the number of bits it has processed and stop when it has reached the specified number. Another method is to add an additional symbol to the source alphabet which can be used to represent the "end of stream" (EOS) condition. This symbol is assigned a frequency of one and is encoded along with all of the other symbols; the decoder can then stop processing data when it decodes an EOS symbol.

Implementation Details

An implementation of the Shannon-Fano encoding and decoding algorithms is included in the file *shanfano.c* (for information on obtaining this source code, see the last section of Chapter 1). This subsection reviews some of the key data structures and algorithms used by the programs.

The encoder calls the routines CountBytes, ScaleCounts, and OutputCounts provided in the file *computil.c* to retrieve the order-0 probabilities from the input file, to scale them so their values are all less than 256 (so each of the counts will fit within a single byte), and to write the scaled counts to the output file. The decoder calls the routine InputCounts after it has opened the compressed data file to retrieve these counts. Both the encoder and decoder dynamically allocate an array of 256 unsigned longs to hold these counts, one for each possible source symbol, even though the decoder uses only one byte of each long to hold the scaled count.

The encoder and decoder both construct a binary coding tree using these counts. The tree is constructed of *Node_T* structures; each of these includes the accumulated weight of all symbols in the subtree below the node, pointers to the node's children, and a pointer to the first of the leaf nodes which is included in its subtree. The entire tree is constructed within the *nodes* data structure, which is just an array of 513 *Node_T*s; each of the pointers is just the index of an element of the *nodes* array. 513 elements are sufficient to hold the largest possible coding tree; a binary tree with n leaf nodes always contains n - 1 internal nodes, and there are up to 257 leaf nodes (one for each of the possible input symbols and an EOS symbol). The *nodes* array is organized to form the tree as follows: the first 257 elements are reserved for the leaves, and in these elements, the node's *first_leaf* field is actually used to store the symbol's value. Some of these nodes may be unused if the input file does not contain all 256 possible symbols, but the first 257 elements are always reserved for leaves. The 258th element is the tree's root node, and the internal nodes are allocated from the higher numbered nodes; however, each subtree eventually points back to the leaves in the first 257 nodes.

The encoder and decoder both dynamically allocate space for the *nodes* array, and both call the routine *build_tree* and provide it the array of scaled counts and the array of nodes. *build_tree* first copies the counts into the leaves of the tree, then sorts the leaf nodes in descending order, so that the node with the highest count is in *nodes [0]*. The last node with a non-zero count is the EOS symbol, with a count of one; all symbols with a count of zero occupy unused nodes between the EOS node and the root node. Once the leaves have been ordered, the root node's *count* field is initialized to be the sum of the counts of all the

leaves and its *first_leaf* field is set to 0, the index of the first leaf node. The next available node pointer is set to 258 (the next element after the leaf node), and the nodes array and the pointers to the root node and next available node are passed to the recursive subroutine *subdivide_node.*

subdivide_node is the routine which actually splits a list of symbols into two halves of approximately equal weight. It starts scanning at the leaf node pointed to be the current node's *first_leaf* and continues adding leaf counts until it exceeds half of the current node's *count.* Once the halfway point has been determined, the list of nodes is separated into two halves. If the number of nodes in either half is one, the appropriate child pointer in the current node is set to point to that leaf node; otherwise, a new node is created in the element indexed by the *next_free_node* pointer, the node's *count* is set to the sum of the counts of the nodes in its sublist, its *first_leaf* pointer is set to the first leaf node in the sublist, and the appropriate child pointer of the current node is set to the new node. *next_free_node* is incremented, and *subdivide_node* is called recursively with the *nodes* array, new node pointer, and *next_free_node* as parameters.

Once *subdivide_node* completes, the entire Shannon-Fano coding tree is built. The decoder can then use the tree directly to decode the input bit stream; however, the tree is not as useful to the encoder, since it would need to traverse the tree from a leaf node to the parent to encode a symbol. This would require that each node have an additional *parent* pointer, and the code generated by the traversal from the leaf to the root would be backward. Also, locating the leaf node corresponding to a symbol would require a linear scan of the leaf nodes, which would adversely affect execution speed. While these problems could be solved, the encoder uses a different technique. It allocates space for *code,* which is an array of 257 *Code_T* structures, each of which contains an unsigned short code value and an integer length (in bits). Once the coding tree is built, *convert_tree_to_code* is called to perform an in-order traversal of the tree, accumulating bits of the code as it visits each left or right child. When a leaf node is reached, the accumulated code value and associated length are stored in the *code* entry corresponding to the leaf node's symbol value. Once *convert_tree_to_code* completes, the *code* array has been built, and the encoder can just read input symbols and output the corresponding entries in the array.

The compression and decompression routines in *shanfano.c* were used to create the *shane* and *shand* programs (for Shannon-Fano encoder and decoder, respectively), and both have been debugged and tested. Their memory usage and compression efficiency are compared to those of the Huffman and arithmetic coders described below in the last section of this chapter.

Huffman Coding

In 1952, David E. Huffman of M.I.T. published an alternative method of constructing variable-length compression codes in the paper entitled *A Method for the Construction of Minimum-Redundancy Codes* ([HUFFMAN]). The idea was similar to that proposed by Shannon and Fano, but a different algorithm for forming the coding tree was proposed. This algorithm overcame some problems of the Shannon-Fano technique, which occasionally assigns a longer code to a more probable message than it does to a less probable one; because of this, the algorithm typically compresses better than Shannon-Fano coding. In fact, Huffman was able to prove that his algorithm was optimal; that is, given a source's probability information *P*, no other algorithm which assigns variable-length codes with integral code lengths can generate a shorter average code length than Huffman's technique. Since Shannon's code was shown to vary at most one bit from the entropy of the source, and Huffman proved his code to be at least as good as Shannon's, its average length *L* likewise has an upper bound of *H(S)* plus one bit. In fact, this upper bound has been proven to actually be $p + 0.086$, where p is the probability of the most likely symbol ([GALLAGER]). If this probability is relatively small, this is a tighter bound than that of the Shannon-Fano code; however, for sources in which there is a symbol whose probability approaches unity, performance is similar to that of the Shannon-Fano code. Again, this is due to the fact that the code words are constrained to have an integral number of bits.

Encoding Algorithm

As with the Shannon-Fano technique, the primary task of the encoder is building the coding tree which is used to assign codes to the source symbols; once the tree has been built and the codes assigned, the encoding process is identical. Huffman's algorithm to build the coding tree is as follows:

1. Generate a list of symbols occurring in the input, along with the relative frequency of each (i.e. the count of the number of times each symbol occurred).
2. Locate the two elements of the list with the smallest frequencies.
3. Join these two elements together into a single parent node, whose frequency is the sum of their individual frequencies.
4. Replace the two elements in the list with the new combined element.
5. Repeat steps 2 through 4 until only a single element remains in the list.

This algorithm works from the bottom up, starting with the leaf nodes and working toward the root, rather than from the top down, working from the root toward the individual leaves like the Shannon-Fano algorithm. Fig. 5 shows the example of compressing the string "aaaaaaaaaccccccbbbdd" used in the previous section. As shown, the Huffman algorithm produces the same coding tree as the Shannon-Fano algorithm for this particular case. However, Fig. 6 (on the next page) shows a different source, with S = {a, b, c, d, e, f} and P = {0.2, 0.3, 0.1, 0.2, 0.1, 0.1}, for which the algorithms produce different codings. For the Shannon-Fano code, L is given by

$$L = 0.2\ (2) + 0.3\ (2) + 0.1\ (3) + 0.2\ (3) + 0.1\ (3) + 0.1\ (3) = 2.5 \text{ bits / symbol}$$

and for the Huffman code, L is given by

$$L = 0.2\ (2) + 0.3\ (2) + 0.1\ (3) + 0.2\ (2) + 0.1\ (4) + 0.1\ (4) = 2.4 \text{ bits / symbol.}$$

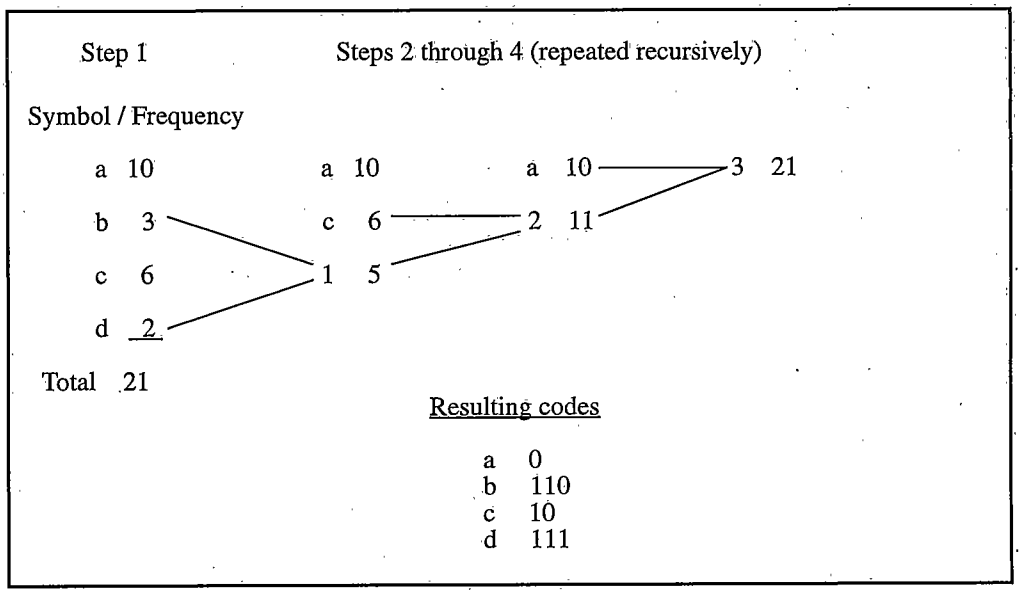

Figure 5: Example of Huffman Code Generation

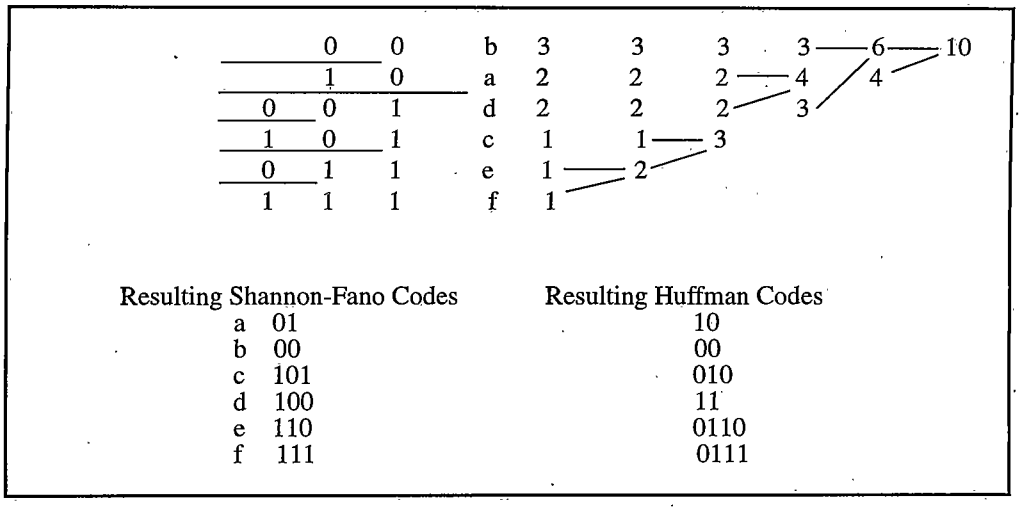

Figure 6: Example Comparing Shannon-Fano and Huffman Codes

The Huffman code does outperform the Shannon-Fano code in this instance. Note that the Huffman code has the same prefix property and associated unique decodability as the Shannon-Fano code, for the reasons discussed in the previous section.

Decoding Algorithm

The decoding algorithm for the Huffman code is identical to that used for the Shannon-Fano code, once the coding tree has been built; the decoder just traverses the tree from the root, consuming one bit of input at a time and visiting successive nodes' children until a leaf is reached and the corresponding symbol is output. Like the Shannon-Fano decoder, an EOS symbol is encoded to allow the decoder to detect the end of the compressed data.

Implementation Details

An implementation of the Huffman encoding and decoding algorithms is included in the file *huffman.c* (for information on obtaining this source code, see the last section of Chapter 1). The majority of the module is identical to *shanfano.c*; the only differences are in the fields of the *Node_T* structures used to build the coding tree and the *build_tree* function. Rather than including a *first_leaf* pointer, each node includes a *next_free* pointer; this pointer is used to link together all nodes which have not yet been combined to former

parent nodes. An array of 513 nodes is still allocated, and the first 257 entries are again reserved for leaf nodes; however, the 258th entry is used for the first internal node above the leaves, rather than for the root. Also, the *next_free* pointer is not used to hold the symbol value; the leaf nodes are not sorted, so the index of a leaf is its symbol value.

build_tree copies the symbol counts from their array into their corresponding leaves in the tree and links any with non-zero counts together into a linked list using the *next_free* pointers. Once this is done, a loop is entered which scans through this free list and locates the two nodes with the smallest counts. They are removed from the list, and a new node is created whose children are the two nodes and whose *count* is the sum of its children's counts. This new node is added to the end of the linked list of nodes, the next free node pointer is incremented, and the loop is repeated until there is only one node left in the linked list. This node is the root of the tree.

The compression and decompression routines in *huffman.c* were used to create the *huffe* and *huffd* programs (for Huffman encoder and decoder, respectively), and both have been debugged and tested. Their memory usage and compression efficiency are compared to those of the other two coders in the last section of this chapter.

Arithmetic Coding

Since the publication of Huffman's original paper, a great deal of research has been conducted to try to optimize Huffman coding algorithms for a variety of circumstances. However, another statistical coding method, referred to as *arithmetic coding*, is gaining popularity. Unlike the previous coding techniques, there is no single paper which gave birth to arithmetic coding. The original paper by Shannon hinted at the technique, and brief references to the idea were found in publications in the early sixties by Elias and Abramson ([ABRAMSON]). However, no major advances were made until Pasco and Rissanen independently discovered means of implementing an arithmetic coder using finite-precision arithmetic in 1976 ([PASCO], [RISSANEN76]). Work continued for several years, with publications by Rissanen and Langdon, Rubin, and Guazzo continuing to refine the technique and describe more practical implementations

([RISSANEN79], [RUBIN79], [GUAZZO]). Tutorial articles were published by Langdon in 1984 and Witten, et. al., in 1987 ([LANGDON84] and [WITTEN87]) which greatly increased the understandability of the technique. The latter article included a complete software implementation of an arithmetic coder, which helped to spur some research into the algorithm. However, it is still not as widely known or accepted as other statistical methods.

Although arithmetic coding also produces variable-length codes, unlike the two techniques discussed previously, it does not require that the length of those codes is an integral number of bits. This allows for increased compression efficiency, and in fact arithmetic coding can actually achieve the entropy limit for any source, regardless of the probability distribution of source symbols ([BELL90], p. 108). In particular, its performance when compressing a source which has one symbol with a probability approaching one is significantly better than that of a Huffman or Shannon-Fano coder, due to the fact that, although the information conveyed by the transmission of the highly probable symbol is minimal, Huffman and Shannon-Fano techniques must code the symbol using at least one bit, while the arithmetic coder is not so constrained.

Another significant advance made during the development of the arithmetic coder was the explicit separation of the compressor into distinct modelling and coding modules. That is, a modeler is responsible for providing the probability information described previously, and the encoder uses that information to efficiently encode each symbol. Fig. 7 shows a block diagram of a compression system which separates the modelling and coding modules. The advantage of making this separation is that the coder is no longer tightly coupled to a specific source model; it should be possible to replace the order-0 semi-adaptive model described in this section with a higher order adaptive Markov model without making any modifications to the coding module.

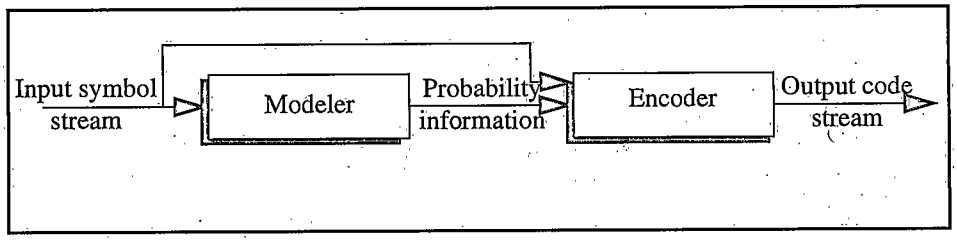

Figure 7: Block Diagram of Compressor Divided into Modelling and Coding Modules

Encoding Algorithm

Arithmetic coding is based on the concept that a message can be represented by an interval of the real numbers in the range [0, 1). As a message becomes longer, the interval needed to represent it narrows, and correspondingly the number of bits required to represent that interval increases. Successive symbols of the message reduce the size of the interval according to their probabilities; compression is achieved by narrowing the interval less for more likely symbols than for unlikely symbols -- narrowing the interval slightly will add fewer bits than narrowing it substantially.

For example, consider the source used in the example shown in Fig. 6, with S = {a, b, c, d, e, f} and P = {0.2, 0.3, 0.1, 0.2, 0.1, 0.1}. Initially, the message spans the entire interval [0, 1); to determine which portion of this range each symbol spans, calculate the set of cumulative probabilities P_c = {0.0, 0.2, 0.5, 0.6, 0.8, 0.9, 1.0}. Note that there is one more element of P_c than in P; the first q elements of P_c correspond directly to the symbols whose probabilities are given in P, and the additional element will always be 1.0. This set becomes the model used to compress the input symbols; as each input symbol is processed, the cumulative probabilities $p_{c\,i}$ and $p_{c\,i+1}$ (the elements of P_c corresponding to the input symbol and the next symbol S) are used to narrow the interval spanned by the message. For example, if the first symbol to be encoded is b, the interval is narrowed from [0, 1) to [0.2, 0.5). This interval is then subdivided by multiplying its range by the elements of P_c; if the next symbol is a, the new interval is [0.2 + 0.0 * (0.5 - 0.2), 0.2 + 0.2 * (0.5 - 0.2)), or [0.2, 0.26). The coding algorithm can be stated as follows:

1. low = 0.0
2. high = 1.0
3. While there are input symbols, perform steps 4 through 7.
4. Get the next input symbol.
5. range = high - low
6. low = low + range * $p_{c\,i}$
7. high = low + range * $p_{c\,i+1}$
8. Choose any real number in the range [low, high) and output to specify the encoded message.

Fig. 8 (on the next page) shows an example using this model to compress the message "bacd". As shown, the final range is [0.2336, 0.2348). The value 0.234375, or 0.001111 in binary, falls within this range, so the bit stream 001111 can be used to represent the message. A standard encoding of S would require three bits per symbol, so the compression ratio is (1 - 6 / 12) * 100% = 50%.

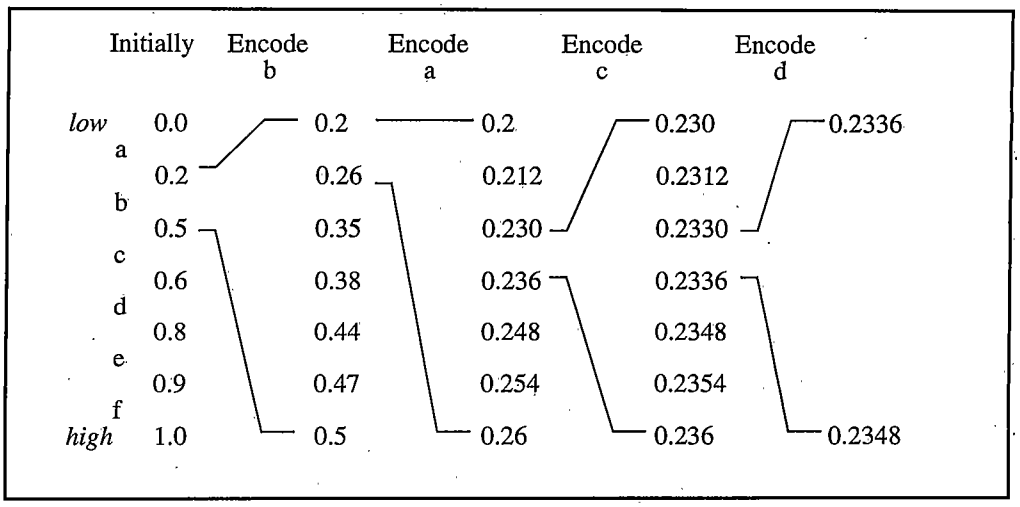

Figure 8: Example of Arithmetic Coding

The obvious problem with this method is that as the message gets longer, the precision needed to maintain the *low* and *high* values and perform the arithmetic operations increases substantially; at some point, this would exceed the capabilities of any digital computer. Fortunately, Pasco and Rissanen independently discovered a means of performing the calculations using fixed point arithmetic; that is, they developed a method which uses only finite-precision integer math. This is the technique which has been developed into the arithmetic coding algorithms used today (i.e. the algorithms presented in [WITTEN87], [BELL90], and [NELSON]).

The first modification required is to change the initial value of *high* from 1.0 to $0.111\overline{1}$; it is agreed that this infinite fraction is equivalent to 1.0. To perform coding, load as many of the most significant bits of *low* and *high* as possible into two fixed-length registers (typically 16 bits for most implementations). The algorithm then proceeds as before, except that when the range is calculated, one is added; that is

$$range = high - low + 1$$

due to the fact that *high* actually has an infinite number of trailing one bits which have been truncated. The new values of *high* and *low* are calculated as before; now, however, after the calculation of the new values, if the most significant bit of each matches, this bit can be shifted left and output. This can be done because the algorithm guarantees that the two values will continue to grow closer together; once their MSBs match, they

will never change, so they can be discarded. This process of shifting bits off the left end continues until they no longer match; the algorithm is then ready to process the next symbol.

This scheme as presented allows the arithmetic coder to use only integer math; obviously, if sixteen bit registers are used, the computer must be capable of performing 32 bit arithmetic operations. Also, if only integer arithmetic is to be used, the cumulative probabilities must be expressed in a different form. An equivalent representation would be to store cumulative symbol counts, along with the total number of symbols. For example, a message with the previous probabilities containing 60 symbols would have the probabilities $P = \{12 / 60, 18 / 60, 6 / 60, 12 / 60, 6 / 60, 6 / 60\}$ and the cumulative probabilities $P_c = \{0 / 60, 12 / 60, 30 / 60, 36 / 60, 48 / 60, 54 / 60, 60 / 60\}$; this can be stored simply as $P_c' = \{0, 12, 30, 36, 48, 54, 60\}$. The desired p_{ci} can then be calculated by $p_{ci} = p_{ci}' / p_{cq+1}'$. The integer arithmetic version of the algorithm is now as follows (where all arithmetic operations, including the divisions, use integer math):

1. low = 0x0000
2. high = 0xffff
3. While there are input symbols, perform steps 4 through 11.
4. Get the next input symbol.
5. range = high - low + 1
6. low = low + (range * $p_{c\,i}'$) / $p_{c\,q+1}'$
7. high = low + (range * $p_{c\,i+1}$) / $p_{c\,q+1}'$ - 1
8. While MSB(low) == MSB(high), perform steps 8 through 10.
9. Output MSB(low).
10. Shift low left one, shifting a zero into the LSB.
11. Shift high left one, shifting a one into the LSB.
12. If the next to most significant bit of low is 0, output 01; otherwise, output 10.

The last step is required to flush any remaining information out of the registers; since it is known that the MSBs of *low* and *high* are different, the next bit is checked to see whether the portion of the range it specifies is the second or third quarter. The pair of bits specify one of those two quarters, so the decoder will be able to correctly decode the last symbol. Since this algorithm produces variable-length codes, it suffers from the same problem as the Huffman and Shannon-Fano coders described previously; the decoder needs some way to determine where the data stream ends, in the event that the end of the code stream is padded out to fill a byte. Like the previous algorithms, the arithmetic coder reserves a special EOS symbol to signify this. It is typically the last entry in the symbol set and is assigned a symbol count of 1.

One final problem with this algorithm must still be resolved. Since *low* and *high* are actually truncated representations of potentially much larger values, care must be taken to ensure that the precision defined by their length is not exceeded. Consider the case where the symbol probabilities are such that *low* and *high* are converging, but the precision required to represent them exceeds sixteen bits before their MSBs match. That is, the target interval spans 1/2, but the endpoints do not cross that division, so their MSBs will not match and cannot be shifted out. Several different methods of dealing with this problem have been proposed; probably the best is that presented in Witten, et. al., in their paper *Arithmetic Coding for Data Compression* ([WITTEN87]); the authors' solution is to prevent this problem from ever happening. It can be detected by looking for situations where *low* = $0.011\overline{1}$ and *high* = $0.100\overline{0}$, so their modified algorithm performs an additional check if their MSBs do not match to determine whether the next most significant bits are 1 and 0, respectively. If they are, that bit is removed from both *low* and *high* and the lowest fourteen bits are shifted left, just as if the MSBs had matched. This process continues until the condition has been corrected; each time a bit is deleted, an underflow counter is incremented. The coder can then continue without fear of underflowing the precision of the registers. Note that a secondary requirement of this is that the range of cumulative counts must be two bits smaller than the range of the registers (fourteen bits for most applications).

Once a symbol is finally encoded which causes the MSBs of *low* and *high* to match, after the bit is shifted out, the bits that were deleted in the previous process must be output. These bits will all be the same, and their value will be the opposite of the MSB that was shifted out; the number of bits to be output is given by the current value of the underflow counter. Once they have been output, the underflow counter is reset to zero and the algorithm continues as before. For a more thorough discussion of the underflow problem and the corresponding word-length constraints, along with a detailed development of the arithmetic coding algorithm, see Section 5.2 of Text Compression ([BELL90]).

The updated version of the algorithm is shown on the next page. It should be noted that there is implicitly an EOS symbol at the end of the input stream, and that this is encoded using the same procedure as the other symbols.

1. low = 0x0000, high = 0xffff
2. underflow_count = 0
3. While there are input symbols, perform steps 4 through 18.
4. Get the next input symbol.
5. range = high - low + 1
6. low = low + (range * $p_{c\,i}'$) / $p_{c\,q+1}'$
7. high = low + (range * $p_{c\,i+1}$) / $p_{c\,q+1}'$ - 1
8. While MSB(low) == MSB(high), perform steps 9 through 14.
9. Output MSB(low).
10. While underflow_count > 0, perform steps 11 and 12.
11. Output the opposite bit of MSB(low).
12. Decrement underflow_count.
13. Shift low left one, shifting a zero into the LSB.
14. Shift high left one, shifting a one into the LSB.
15. While next MSB(low) == 1 and next MSB(high) == 0, perform steps 16 through 18.
16. Shift the lower 14 bits of low left one, shifting a zero into the LSB.
17. Shift the lower 14 bits of high left one, shifting a one into the LSB.
18. Increment underflow_count.
19. If the next to most significant bit of low is 0, output 01; otherwise, output 10.

Decoding Algorithm

The decoder is faced with a slightly different problem than the encoder; given the input value, it must determine which symbol, when encoded, would reduce the range to an interval which still contains the value. It uses the same *low* and *high* registers as the arithmetic decoder; in addition, it uses a *code* register which contains the next sixteen bits from the input stream. Likewise, it requires the same cumulative probabilities P_c' used by the encoder. The algorithm is as follows:

1. low = 0x0000, high = 0xffff
2. code = first sixteen bits of input
3. Repeat steps 4 through 18 until the EOS symbol has been decoded.
4. range = high - low + 1
5. count = ((code - low + 1) * $p_{c\,q+1}'$ - 1) / range
6. i = q + 1
7. While count < $p_{c\,i}'$, decrement i.
8. Output symbol corresponding to i.
9. low = low + (range * $p_{c\,i}'$) / $p_{c\,q+1}'$
10. high = low + (range * $p_{c\,i+1}$) / $p_{c\,q+1}'$ - 1
11. While MSB(low) == MSB(high), perform steps12 through 14.
12. Shift low left one, shifting a zero into the LSB.
13. Shift high left one, shifting a one into the LSB.
14. Shift code left one, shifting the next input bit into the LSB.
15. While next MSB(low) == 1 and next MSB(high) == 0, perform steps 16 through 18.
16. Shift the lower 14 bits of low left one, shifting a zero into the LSB.
17. Shift the lower 14 bits of high left one, shifting a one into the LSB.
18. Shift code left one, shifting the next input bit into the LSB.

The algorithm's main loop (steps 4 through 18) can basically be separated into two sections; the first, steps 4 through 8, actually decodes the source symbol, and the second, steps 9 through 18, removes as many bits as possible from the input after the symbol is decoded. The calculation of the *count* value in step 5 is essentially a scaling of the input *code* from the range [*low*, *high*) to the range [0, $p_{c\,q+1}'$). Once this value has been calculated, steps 6 and 7 just perform a linear search of the cumulative probabilities to find the symbol whose probability range encompasses the value. This is the symbol which must have been encoded, so it is output. The second section is nearly identical to the main loop of the encoding algorithm, except that it just discards bits rather than outputs them; for this reason, it does not need to keep track of the number of underflow bits.

Implementation Details

An implementation of the Shannon-Fano encoding and decoding algorithms is included in the file *arith.c* (for information on obtaining this source code, see the last section of Chapter 1). As is done in the Huffman and Shannon-Fano implementations, the utility routines CountBytes, OutputCounts, and InputCounts are used to retrieve the required symbol probabilities. A custom version of the *scale_counts* routine is used, because the counts must be constrained so their total does not exceed fourteen bits. Once the counts are available, both the encoder and decoder construct a model; however, this process is much simpler than the tree construction in the previous algorithms. An array of 258 unsigned shorts, *totals*, is allocated and filled with the cumulative symbol counts; the 258th entry is the total count.

Once the encoder has initialized the *totals* array, it enters a loop in which it reads source symbols, retrieves their probability information from the model, and passes this information to the arithmetic encoder. As illustrated in the algorithm, this probability information consists of the cumulative symbol counts associated with the symbol and with the following symbol, along with the total count, which is retrieved from the last entry. The arithmetic encoder is a straightforward implementation of the algorithm given previously. Once all of the input symbols have been encoded, the EOS is encoded, the final information is flushed out of the encoder, and a block of 16 zero bits is output. This is to ensure that when the decoder is reading data,

there will always be at least enough bits following those used to encode the EOS symbol that its *code* register can be filled.

The decoder is also a very straightforward implementation of the algorithm given in the previous subsection. It uses the same routine to build the *totals* array, then it enters the loop in which it determines which symbol was encoded, based on the cumulative probabilities in *totals*, then removes all possible bits from the input. This continues until it decodes the EOS symbol.

The compression and decompression routines in *arith.c* were used to create the *arithe* and *arithd* programs, and both have been debugged and tested. Their memory usage and compression efficiency are compared to those of the other two coders in the next section.

Comparison of the Different Implementations

As mentioned in the first chapter, the emphasis in this thesis is on implementations which are memory-efficient and have reasonable compression efficiency, so these two criteria are analyzed in detail. Although a detailed analysis of the programs' execution speed is not present, some observations relating to the relative speed of the different programs are also presented. The programs being evaluated are *shane* and *shand*, the Shannon-Fano compressor and decompressor, *huffe* and *huffd*, the Huffman compressor and decompressor, and *arithe* and *arithd*, the arithmetic coding compressor and decompressor.

Memory Usage

All of the compressors dynamically allocate an array of 256 unsigned longs which is used to hold the symbol counts as they are being accumulated; the decompressors allocate the same data structure, although their counts are guaranteed to require only one byte each. The Huffman and Shannon-Fano programs all allocate an array of 513 nodes from which to build their coding trees; although the fields used differ between the Huffman and Shannon-Fano algorithms, the size of each node is eight bytes in both implementations. In addition, the Huffman and Shannon-Fano compressors both allocate an array of 257 code structures, each of which is four bytes long. The arithmetic coding programs both allocate an array of 258 unsigned shorts which hold the cumulative symbol counts; in addition, each compressor requires two

unsigned shorts to hold the low and high registers and a long to hold the underflow counter. Each decompressor requires the same low and high values and an unsigned short to hold the code register. Table 2 shows the memory thus required by each of these programs to hold their data structures. It can be seen that the arithmetic coding programs make more efficient use of memory, due to the simplicity of their probability models; however, none of the programs' memory requirements are sufficient to cause implementation difficulties on any platform.

Executable File Name	Memory Required (in Bytes)
shane	6156
shand	5128
huffe	6156
huffd	5128
arithe	1548
arithd	1546

Table 2: Memory Requirements of Semi-Adaptive Order-0 Programs

Compression Efficiency

Each of the compression programs was run on each of the files in the Calgary compression corpus; Table 3 (on the next page) shows the resulting compression ratios. As is expected, *shane* and *huffe* produce very similar results, with *huffe* performing slightly better in each case. *arithe* also performs slightly better than *huffe* in each case.

These results are compared with the ratios of the other compressors described in the following chapters and with the ratios of some publicly available compressors in a table in Chapter 7.

Execution Speed

As should be expected, the computational overhead of the numerous arithmetic operations performed by *arithe* and *arithd* significantly slow their respective execution speeds in comparison with the other compressors and decompressors. Since the portions of the Shannon-Fano and Huffman programs which actually encode and decode the data are identical, the only distinction lies in the time required to build the coding trees. Since Huffman codes have been shown to be optimal, the usefulness of implementing a

Shannon-Fano code is very debatable; however, construction of the Huffman trees is somewhat more complicated than construction of the Shannon-Fano trees, so the Shannon-Fano programs do run marginally faster. In an application which required that the coding trees be rebuilt frequently, it could be worthwhile to use a Shannon-Fano code.

		Output of shane		Output of huffe		Output of arithe	
File Name	Size	Size	Ratio	Size	Ratio	Size	Ratio
bib	111,261	73,075	35%	72,933	35%	72,496	35%
book1	768,771	440,871	43%	440,112	43%	436,775	44%
book2	610,856	370,280	40%	369,145	40%	366,864	40%
geo	102,400	73,659	29%	73,394	29%	73,054	29%
news	377,109	247,235	35%	246,814	35%	244,998	36%
obj1	21,504	16,451	24%	16,415	24%	16,372	24%
obj2	246,814	195,793	21%	195,152	21%	194,260	22%
paper1	53,161	33,532	37%	33,491	38%	33,274	38%
paper2	82,199	47,923	42%	47,833	42%	47,516	43%
paper3	46,526	27,427	42%	27,415	42%	27,273	42%
paper4	13,286	7,969	41%	7,966	41%	7,914	41%
paper5	11,954	7,586	37%	7,549	37%	7,493	38%
paper6	38,105	24,260	37%	24,165	37%	24,004	38%
pic	513,224	122,209	77%	122,039	77%	108,475	79%
progc	39,611	26,207	34%	26,042	35%	25,879	35%
progl	71,646	43,510	40%	43,217	40%	42,924	41%
progp	49,379	30,457	39%	30,456	39%	30,274	39%
trans	93,695	65,552	31%	65,414	31%	64,982	31%

Table 3: Compression Efficiency of Semi-Adaptive Order-0 Programs

CHAPTER 3

ADAPTIVE COMPRESSION ALGORITHMS

As was mentioned in Chapter 1, there are a number of problems with semi-adaptive compression algorithms. First and perhaps foremost among these is the fact that they cannot be used to compress stream-oriented data; i.e., a semi-adaptive algorithm could not be implemented in a modem, tape controller, or similar device to perform on-the-fly compression of data passing through the device. In a file compression system, this is probably not a problem; however, the associated overhead required to scan each file twice might still be unacceptable. Also, the requirement that some form of statistical information must be included along with each compressed file can adversely affect compression performance; for order-0 models, this overhead is relatively small, but might still be significant when small files are being compressed. However, the amount of statistical data accumulated by higher order models such as those discussed in Chapters 4 and 5 increases exponentially as the order is increased. This quickly eliminates higher order semi-adaptive compressors from consideration.

Fully adaptive compression techniques address all of these problems. Like static compression techniques, these methods make an *a priori* assumption regarding the statistics of the message source to be compressed and begin compressing data using this model. However, unlike static methods, they continually update the model as they continue to compress data. As more data is compressed, the probabilities produced by the model should converge on the actual characteristics of the message source, assuming that the source is ergodic. There may be a degradation of compression efficiency during the early phases of compression while the model is adapting to the characteristics of the data, but an adaptive algorithm should perform as well as or better than a semi-adaptive version of the algorithm after some amount of input has been processed. Also, if the characteristics of the data change substantially at some point in the file, the adaptive algorithms can adjust to the new statistics, while the semi-adaptive algorithms must produce a global model

of the data which does not accurately reflect these local statistics. On short input messages, the fact that the adaptive algorithm is not required to include its model along with the compressed data may also allow it to perform better than a semi-adaptive algorithm. Perhaps the most significant benefit of an adaptive algorithm is that it is possible to effectively utilize higher order modelling techniques in the compressor, since the statistics are not transmitted to the decompressor.

Figs. 9 and 10 show block diagrams of the adaptive compressor and decompressor.

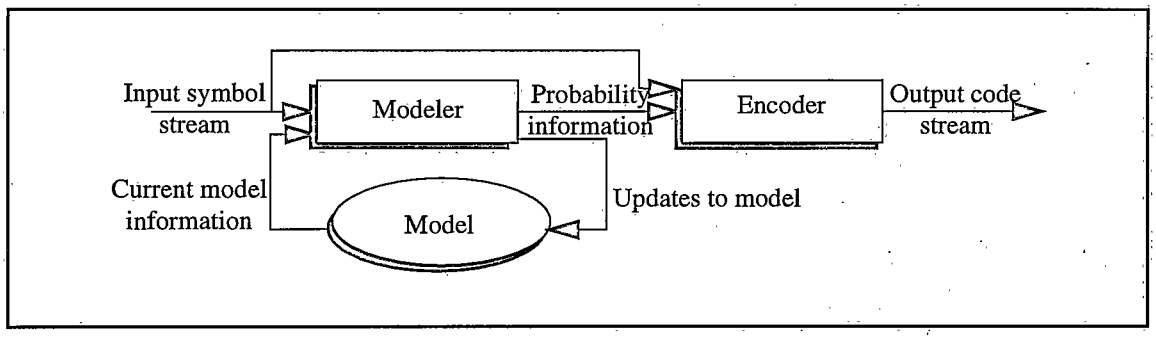

Figure 9: Block Diagram of an Adaptive Compressor

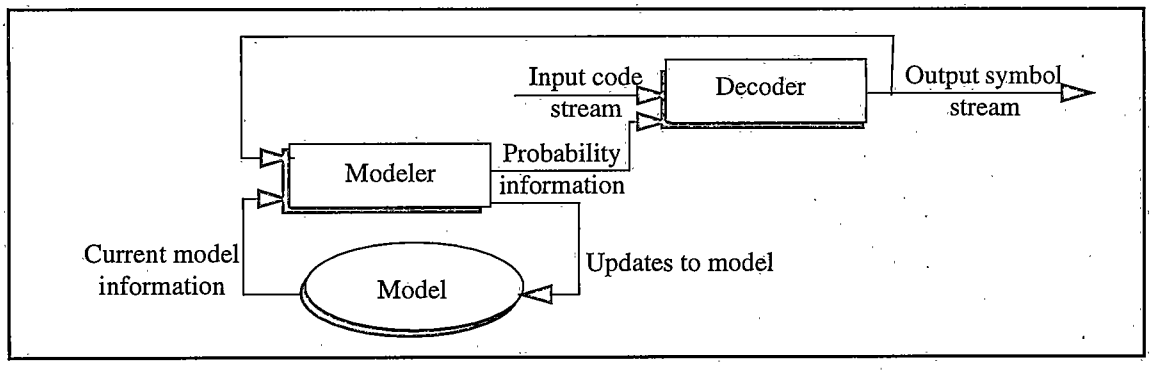

Figure 10: Block Diagram of an Adaptive Decompressor

The remainder of this chapter discusses the modifications required to make the Huffman and arithmetic coding algorithms presented in Chapter 2 adaptive. An adaptive Shannon-Fano algorithm is not presented; while a method is discussed below which allows a Huffman tree to be updated without performing a complete reconstruction of the tree each time, a similar technique has not been presented for Shannon-Fano coding. Although such a technique may actually exist, the increased compression efficiency provided by

Huffman codes and the widespread acceptance of those codes make it unlikely that an adaptive Shannon-Fano algorithm will be actively pursued.

Adaptive Huffman Coding

Huffman's coding technique became very widely accepted in information theory circles following the publication of his paper in 1952 ([HUFFMAN]). A significant amount of research was subsequently conducted into various means of improving the efficiency of Huffman codes, including examinations of methods by which Huffman coding could be made adaptive. It is trivial to perform adaptive compression by simply rebuilding the Huffman tree each time the model is to be updated; however, the computational effort required to rebuild the tree makes this approach infeasible. In 1978, in the paper *Variations on a Theme by Huffman* which was published to honor the twenty-fifth anniversary of the birth of Huffman coding, Robert Gallager introduced a method by which the Huffman tree could be incrementally updated ([GALLAGER]). This method allowed the count of a symbol to be incremented and then allowed the coding tree to be adjusted as if it had been rebuilt, using a relatively small number of operations. These results were later generalized by Cormack and Horspool to allow for arbitrary positive or negative adjustments of symbol counts (in [CORMACK84]); an independent generalization by Knuth allowed increments or decrements by one only, but presented a detailed implementation (in [KNUTH85]).

Tree Update Algorithm

Gallager defined a property of binary code trees which he termed the *sibling property*; the definition was stated as follows:

> A binary code tree has the *sibling property* if each node (except the root) has a sibling and if the nodes can be listed in order of nonincreasing probability with each node being adjacent in the list to its sibling. ([GALLAGER])

He also showed that for each even numbered node $2k$ in the list, the node numbered $2k - 1$ must be its sibling in order for the sibling property to hold, and he proved that any binary code which obeys the prefix property (and is therefore uniquely decodable) is a Huffman code if and only if the code tree correspond-

ing to that code has the sibling property. Fig. 11 shows a sample Huffman tree and the corresponding node list, which does in fact obey the sibling property.

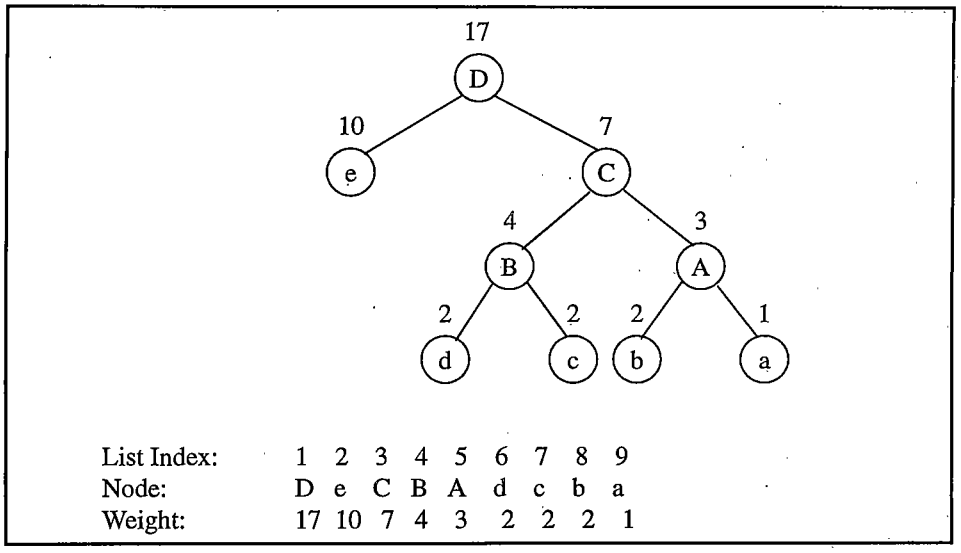

Figure 11: Sample Huffman Tree Exhibiting Sibling Property

A consequence of this observance is that it points out a straightforward method by which to adaptively update the Huffman tree. After a symbol has been encoded, the model must be updated to correctly reflect the symbol's probability. The following algorithm can be used to accomplish this:

1. Locate the leaf node corresponding to the symbol to be updated.
2. Repeat steps 3 and 4 until the tree's root node has been updated.
3. Increment the current node's count.
4. Move to the current node's parent node.

This iterative algorithm is required because the count of any internal node is meant to be the sum of the counts of all leaf nodes in that node's subtree; if the count of one of these leaves is incremented, the count of each of its ancestors must also be incremented. The probability of any symbol occurring is thus the count of the symbol's leaf node divided by the count of the root node.

Obviously, if an *a* were encoded using the tree shown in Fig. 11, this algorithm could be used to update the tree with no difficulty; however, if a *b* were encoded, once the first increment operation was performed, the tree would no longer obey the sibling property, and therefore would not be a valid Huffman cod-

ing tree. However, the update algorithm can be modified as shown below to restore the sibling property to the tree each time a node's count is incremented:

1. Locate the leaf node corresponding to the symbol to be updated.
2. i = list index of leaf node
3. Repeat steps 4 through 9 until the root node has been updated.
4. Increment the count of the ith node.
5. j = i - 1
6. While the count of the jth node is less than the count of the ith node, decrement j.
7. j = j + 1
8. If j is not equal to i, swap the jth node and the ith node and set i = j.
9. i = list index of the ith node's parent

This algorithm requires that the tree be stored in a slightly different manner than that discussed in Chapter 2. The tree is still stored within an array of node structures; however, rather than reserving the first elements of the array for the leaves, the root occupies the first element of the array and the tree grows toward the nodes with higher indices. That is, the indices of a node's children will always be greater than the node's index. As before, each node must contain its frequency count and a flag indicating whether it is a leaf node or an internal node; if it is an internal node, it must also contain the pointers to its children. However, due to the properties of the Huffman tree, it is known that its left and right children are always adjacent, so only a pointer to the left child is required; that is, the right child's index is always the left's plus one. Again, if the flag indicates that the node is a leaf, this pointer can contain the actual value of the symbol. In addition, each node must have a pointer which indexes its parent node. Since one child pointer is lost but the parent pointer is gained in each node, the overall size of the node and therefore the size of the entire tree remain the same as the sizes required by the semi-adaptive algorithm.

The first step of the algorithm requires that the leaf node corresponding to a symbol be located. Since the first q nodes are no longer reserved for the leaf nodes containing the symbols, it is no longer possible to directly index a leaf node given the symbol value. One alternative would be to perform a linear search of the nodes to locate the leaf, but this would be computationally infeasible. Another approach is to maintain a separate array of q pointers to the leaf nodes in the tree, one for each source symbol. Each leaf pointer is directly indexed by the symbol value, and the pointer indicates the desired node in the tree.

Steps 4 through 8 of the algorithm shown on the previous page are the ones which ensure that the Huffman tree maintains the proper order. Once the count of node *i* has been incremented, if it is greater than the count of node *i* - 1, the tree no longer exhibits the sibling property. If the count of node *i* has been incremented from *C* to *C* + 1, steps 4 through 7 find the node with the smallest index less than *i* whose count is *C*. If there is no such node, nothing needs to be done. However, if there is such a node (i.e., if *i* is not equal to *j*), node *i* should be moved to position *j* in the tree; this will restore the sibling property to the tree.

Before moving node *i* into position *j*, something must be done with the node that currently occupies that position. Since node *j* has count *C*, as do all nodes between *i* and *j*, the two nodes can be swapped without violating the sibling property. Swapping the nodes actually implies exchanging the positions of the subtrees with nodes *i* and *j* as their roots. For a binary tree, this is normally a simple matter of exchanging the contents of the two nodes; the child pointers are moved into their correct positions, which in effect moves the subtrees. In this case, this swapping operation is complicated somewhat by the presence of the parent pointers. The parent pointers of nodes *i* and *j* should not be exchanged along with the other contents of the nodes, since the parents' child pointers are not being changed. However, the parent pointers of node *i*'s children should be changed to point to node *j*, and vice versa; in the case where node *i* is a leaf node, the corresponding entry in the array of leaf pointers should be changed to *j*, and vice versa.

Once this swapping operation is completed, the update of *i*'s ancestor nodes can continue; note that this update continues from *i*'s new position in the tree. As an example of this process, consider the execution of this algorithm after the symbol *b* has been encoded using the tree shown in Fig. 11. The leaf pointer for *b* indexes node 8; node 8's count will be incremented to 3, so a node swap is necessary. Node 6 will be identified as the last node with a count less than 3, so these two nodes will be exchanged. Since neither has children, no parent pointers must be changed; however, the leaf pointer for *b* will be changed to 6 and the leaf pointer for *d* will be changed to 8. The update will continue to increment the weights of nodes 4, 3, and 1; none of these increments will require swapping. The resulting tree is show in Fig. 12.

If this tree is then used to encode another *b*, a different code will be produced, since the symbol is in a different position in the tree. The update process will then proceed as follows: the leaf pointer for *b* now

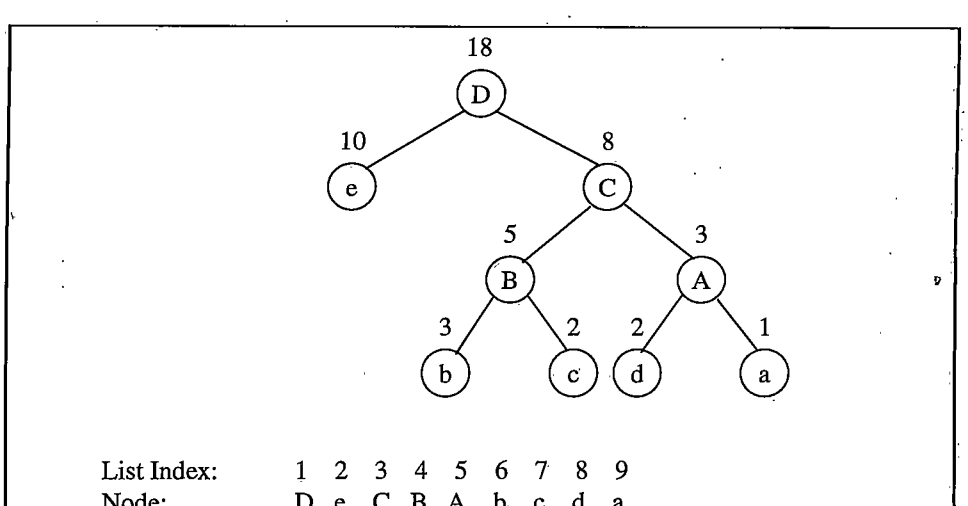

Figure 12: Updated Huffman Tree after Encoding the Symbol *b*.

indexes node 6, so node 6's count will be incremented to 4. Node 5 is the last node with a weight of 3, so 5 and 6 must be swapped. Node 6 is the leaf node for *b*, so *b*'s leaf pointer is changed from 6 to 5; the parent pointers of nodes 8 and 9 are changed from 5 to 6. Once the swap is complete, the update continues with nodes 3 and 1; neither of these increments will necessitate a swap. The updated tree is shown in Fig. 13.

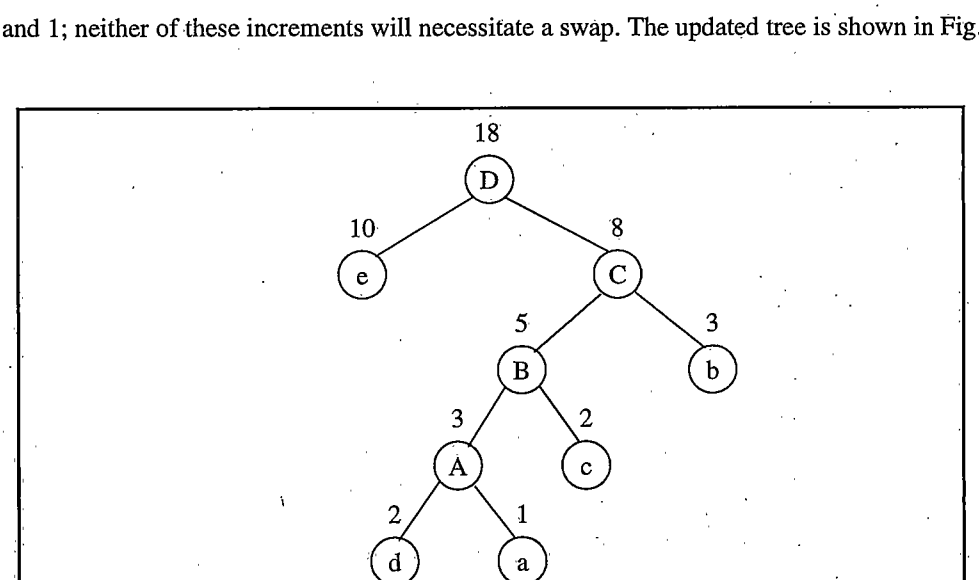

Figure 13: Updated Huffman Tree after Encoding a Second *b*.

Note that after this update, not only has the value of the code for *b* changed, but its length has decreased by one. This is the desired effect, as the initial estimate of *b*'s probability was too small. If a string of several *b*s were encoded, its code would eventually be one bit long, with the lengths of the other symbols' codes increasing as required to maintain a proper Huffman coding tree.

Initialization of the Tree

As stated previously, an adaptive compressor must begin with some assumption about the statistics of the message to be compressed in order to initialize its model. There are a number of schemes which can be used to accomplish this. One of the simplest is to make an initial assumption that all symbols occur with a count of one. This will result in an initial set of code words which nearly all have the same length; there will be two which are 9 bits long due to the fact that there are 257 symbols, including the EOS. As input data is compressed, these code lengths will adjust, as was shown in the previous example. However, the fact that each symbol is included in the model even though it never occurs in the input has an adverse affect on compression efficiency. This is because some portion of the code space available to the encoder is allocated for codes that will never be generated. It is thus desirable to include only those symbols in the model which actually occur in the input.

Without an accurate *a priori* knowledge of the message source's characteristics, the model cannot always be initialized to contain only the necessary symbols. This problem can be addressed by modifying the encoding algorithm so that it allows new symbols to be added to the tree after compression has begun. This can be done by adding another special symbol to the source alphabet; this symbol is commonly called the escape symbol. If the compressor encounters a symbol in the input data which is not currently in the model, it encodes the escape symbol instead. After the escape's code has been produced, the binary value of the input symbol is written to the output, and the new symbol is added to the Huffman tree. When the decompressor decodes an escape symbol, it reads the following symbol value from the compressed data, outputs it, and also adds it to the Huffman tree. The normal update procedure is performed on the new node, and compression continues as usual.

Given this mechanism for adding new symbols to the tree, it is not necessary to include all possible symbols in the initial tree; however, this still does not address the question of exactly how the tree should be initialized. While an attempt could be made to guess the characteristics of the input, perhaps by examining a block of the first symbols to be compressed and initializing the tree with a best guess of those characteristics, a much simpler initialization is to include only the EOS and escape symbols in the initial tree. This guarantees that each symbol present in the input message will generate an escape symbol the first time the input symbol occurs, which may affect the compression efficiency somewhat, but the ease of implementation of this method makes it an attractive alternative.

If the tree is initialized with only the EOS and escape symbols, each of these must be assigned a starting count. The EOS symbol's count is of course initialized to one and will never be updated; however, the appropriate method of handling the escape is less obvious. A number of possibilities have been explored; this is discussed in more detail in Chapter 4. The simplest technique is to treat the escape the same as the EOS; that is, to initialize its count to one and never update it while compressing data. As new symbols are added to the tree, the length of the code for the escape will necessarily increase; this is probably desirable behavior. As more symbols are added to the model, the chance that an unrecognized symbol will be encountered decreases, so the corresponding probability that an escape will be encoded also decreases. This should correspond to an increase in the length of the escape symbol's code.

The implementations of the adaptive Huffman encoder and decoder which are analyzed in the last section of the chapter use these simple techniques for initializing the tree and handling the escape probabilities.

Adding New Nodes

The structure of the tree makes it relatively simple to add a node for the new symbol. This is done by modifying the last node, which should be the leaf node for the symbol with the smallest count. The node is changed from a leaf node to an internal node with the same count, and two new nodes are added to the tree as its children. The first child becomes a leaf node containing the symbol that was just displaced, with the same symbol value and count, and the second becomes a leaf node containing the new symbol. This new

node's count is initialized to zero; this ensures that the count contained in the new nodes' parent node is still correct and that the tree still obeys the sibling property. After the nodes are added, the leaf pointers for the two symbols are updated to point to the new nodes, and the addition of the symbol is complete.

Once the new node is added, it is updated using the standard algorithm described previously. This will increment all of the necessary counts and will ensure that the tree is adjusted as required to maintain the sibling property. Fig. 14 shows an example of a tree initialized to contain only the EOS and escape symbols then used to encode the source message "ab".

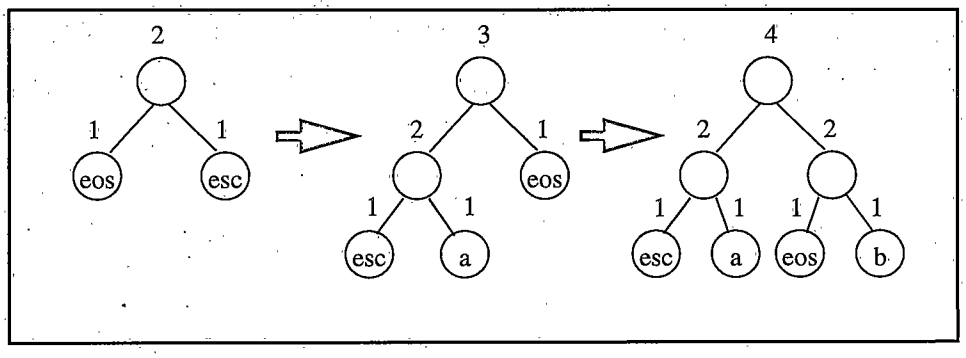

Figure 14: Example of Adding Nodes to an Initialized Tree

Modifications to the Encoding and Decoding Algorithms

The decompression algorithm needs to be modified only minimally from the semi-adaptive version. Rather than read the symbol counts from the input file, the algorithm just initializes its tree and corresponding leaf pointer array with the EOS and escape symbols, then enters the normal decompression loop. The decoder which uses the tree to process the input bits and recover the source symbol is identical; in the loop, after the symbol is decoded, it is checked to see if it is an escape. If so, the new symbol's value is read from the code stream and the new node is added to the tree. After the decoded symbol is output, the update routine described previously is called, then the decompressor is ready to process the next input. This continues until the EOS symbol is decoded.

The compression routine is modified in a similar fashion, using the same routines to initialize and update the tree and to add new nodes when necessary. An important difference between the adaptive version

and the semi-adaptive version is that since the Huffman codes to be produced by the adaptive algorithm are potentially changing with each input symbol processed, it is not practical to maintain the lookup table which was used in the semi-adaptive version. Instead, each symbol is encoded by locating the appropriate leaf node in the tree using the leaf pointer array and then following the parent pointers to the root of the tree, accumulating the next bit to be output at each step. Note that this produces the code to be output in reverse order, so the bits must be accumulated in a last in-first out fashion before they are output.

Limiting the Depth of the Tree

The requirement that the encoder accumulates code bits and reverses them before outputting the code implies that codes should be limited to some maximum length to prevent overrunning the space in which the bits are accumulated. Typically, the bit reversal stack will be a 16 or 32 bit register, so the codes should be limited to this length. The maximum length of any code is the same as the maximum depth of the Huffman tree, so the tree's depth must somehow be managed. Keeping track of the maximum depth during the update process would be a difficult task; fortunately, there is a definite relationship between the maximum possible depth of the tree and the count of the tree's root. Consider the symbol counts which would increase the depth of the tree most rapidly; this worst case scenario is shown in Fig. 15.

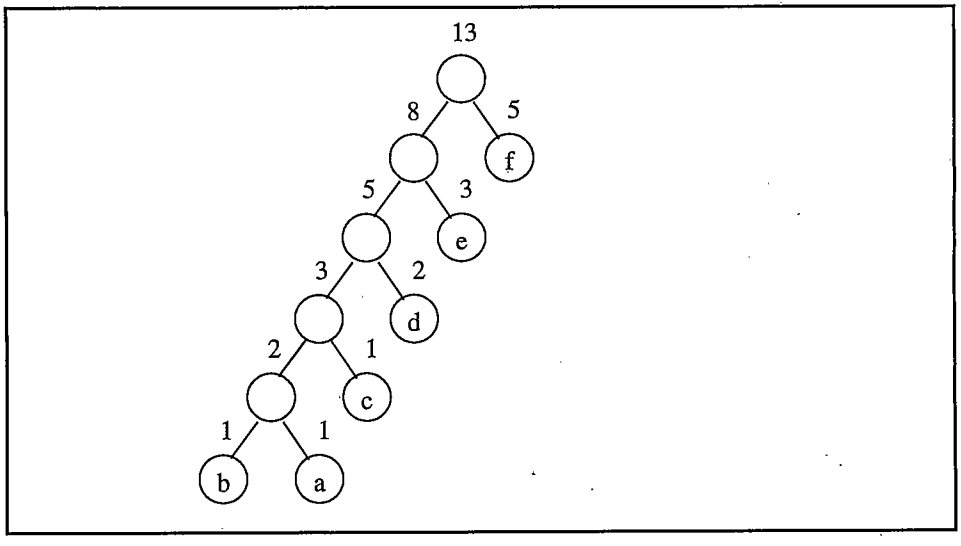

Figure 15: Huffman Tree of Maximum Depth with Minimum Root Node Count

The sequence of weights which produces this degenerate tree, {1, 1, 2, 3, 5,...}, is the Fibonacci sequence; this sequence is defined recursively as

$F_n = F_{n-1} + F_{n-2}$ n > 2
$F_n = 1$ n = 1 or 2

As can be seen from this worst case example, if the count of the root node is F_m, the maximum depth of the tree is m-1 bits. In order to guarantee that the depth of the tree cannot exceed n bits, it is only necessary to ensure that the root node's count is less than F_{n+2}. To guarantee a code length of 16 bits or less, the root node's count must be less than 4181, and to guarantee a code length of 32 bits or less, the count must be less than 3,524,578. In practicality, if unsigned shorts are used to store the node counts, the maximum value of the root node would be 65535, which would limit the maximum code length to 22 bits.

Even if the code length limit is 32 bits, steps must be taken to prevent the node counts from overflowing. This can be accomplished by dividing each of the counts in the tree by two if the root node's count becomes too large; this appears to be a fairly simple operation, but due to the fact that integer division must be used, the resulting tree might not obey the sibling property. The easiest method of avoiding this is to simply rebuild the tree each time the counts must be scaled; the following algorithm can be used to rebuild the tree:

1. j = index of last node in tree, i = j
2. While i > index of the root node, perform steps 3 through 7.
3. If the ith node is a leaf, perform steps 4 through 6.
4. Copy the entire node to the jth position in the tree.
5. Increment the node's count, then divide it by 2.
6. Decrement j.
7. Decrement i.
8. i = index of last node in tree - 1
9. While j >= index of the root node, perform steps 10 through 16.
10. weight = count of ith node + count of (i+1)st node
11. k = j + 1
12. While weight is less than count of jth node, increment k.
13. Decrement k.
14. Move every node from j+1 to k up one, into nodes j to k+1.
15. Set the kth nodes count = weight, child pointer = i, and leaf flag to false.
16. i = i - 2, j = j - 1
17. For i = last free node down to index of root node, perform steps 18 and 19.
18. If the ith node is a leaf, set the leaf pointer for the node's symbol to i.
19. Otherwise, set the parent pointer of the ith node's child to i.

Steps 1 through 7 collect all of the leaf nodes, move them to the last half of the node list, and divide all of their counts by 2; 1 is added to the count before the division in order to prevent counts of 1 from being changed to zero. After this is done, steps 8 through 16 construct the internal nodes of the tree by creating a new node whose children are the smallest pair of nodes that have not yet been placed back into the tree. The pointer *i* indexes this pair of nodes, and *j* points to the location of the new node. When this node is created, it is possible that it is in the wrong location in the tree because its count is not the largest one; if this is the case, the proper location for that count is located, all of the nodes from *j*+1 to that point are moved up one, and the new node is inserted in the appropriate place. Note that the *i* pointer is decremented by two each time the *j* pointer is decremented by one, so by the time *j* reaches the root node, *i* will point to the first two child nodes, and the entire tree will be constructed. Steps 17 through 19 go through the tree and set the parent and leaf node pointers to their appropriate nodes.

Implementation Details

Implementations of the compression and decompression algorithms are contained in the file *adap_huf.c* (for information on obtaining this source code, see the last section of Chapter 1). They are a fairly straightforward implementation of the algorithms discussed above. These routines were used to create the *ahuffe* and *ahuffd* programs; both of these programs have been debugged and tested. Their memory usage and compression efficiency are compared to those of the adaptive arithmetic coding programs in the last section of this chapter.

Adaptive Arithmetic Coding

The modifications required to make the arithmetic coding algorithms adaptive are very straightforward. In the encoding and decoding algorithms which do not use escape codes, the model does not change in any way, although its initialization is simpler; rather than scanning the input and determining symbol counts, each symbol is assigned a count of one, so the cumulative count for any symbol is identical to its binary value. The encoding and decoding functions remain the same; however, in the main loop of both the compressor and decompressor, a call to the *update_model* function is added after each source symbol has been

processed. This function simply increments each of the cumulative counts from that of the symbol just processed to the end of the list. Once this increment is performed, the total count in the last position is checked to make sure that it does not contain more than fourteen significant bits (i.e., that it is not greater than 16383); if it has exceeded this limit, all of the counts are scaled down by a factor of two.

Some additional modification must be made to these adaptive algorithms in order to use escape codes. The model is initialized to contain only the escape and EOS symbols, each with a count of one, and all other counts are set to zero. Also, in the compressor's main loop, if a symbol is not present in the model, the escape symbol is encoded instead. Once this is done, the input symbol is encoded using a special escape model; this model is the same as the initial model used by the previous algorithms, and because every symbol is present, the compressor is guaranteed to be able to encode or decode any symbol using this model. In the escape model, each symbol's cumulative count is the same as its binary value; the total is therefore 256, since the escape and EOS symbols are not present. Once the escaped symbol is encoded, only that symbol is updated, and only in the regular model; the escape symbol's count remains one, and the special escape model is never changed. The same *update_model* function used in the previous algorithm is used here.

The decompressor is modified in a similar way; if the symbol decoded is the escape symbol, it is discarded, and the next symbol is decoded and removed from the compressed data stream using the escape model. The regular model is then updated for the latter symbol; again, the escape symbol's count remains one, and the escape model is never updated.

Implementation Details

Implementations of the compression and decompression algorithms with and without the escape symbol are contained in the files *adap_ari.c* and *adap_are.c*, respectively (for information on obtaining this source code, see the last section of Chapter 1). These files are very similar to the file *arith.c*, with the differences noted in the previous section. It should be noted that due to its simplicity, the escape model is not actually stored in memory; the required counts are computed each time the model is needed.

The routines in *adap_ari.c* were used to create the *aarithe* and *aarithd* programs, and those in *adap_are.c* were used to create the *aarithee* and *aarithed* programs. All of these programs have been debugged

and tested. Their memory usage and compression efficiency are compared to those of the adaptive Huffman programs in the next section.

Comparison of the Different Implementations

This section presents an analysis of the memory usage and the compression efficiency of each of the programs described in this chapter. along with some observations relating to the relative speed of the different programs. The programs being evaluated are *ahuffe* and *ahuffd*, the adaptive Huffman compressor and decompressor, *aarithe* and *aarithd*, the adaptive arithmetic coding compressor and decompressor, and *aarithee* and *aarithed*, the adaptive arithmetic coding programs which use the escape symbol.

Memory Usage

Each of these programs has an initial advantage in memory usage over its semi-adaptive counterpart, since the 1 Kbyte array of symbol counts is no longer necessary. The Huffman programs allocate an array of 515 nodes from which to build their coding trees; the size of each node is eight bytes, assuming that the compiler pads byte fields in the structure to word boundaries. However, it should be noted that this could be reduced to six bytes; the *child_is_leaf* flag requires only one bit, and none of the pointers will ever index an array of more than 32000 entries, so the flag could be combined with one of the pointers. However, since this would require a significant amount of additional computation in order to extract the pointer or flag from the combined field each time it was used, and the memory requirements are not overly large, this was not done for this implementation. In addition, the programs allocate an array of 258 unsigned shorts for the array of leaf pointers and a single unsigned short for the next free node pointer. The adaptive arithmetic coding programs require only an array of unsigned shorts in which to store the cumulative symbol counts, one for each symbol; the version which uses the escape symbol has an alphabet of 258 symbols, while the other version has only 257 symbols. In addition, each compressor requires two unsigned shorts to hold the low and high registers and a long to hold the underflow counter. Each decompressor requires the same low and high values and an unsigned short to hold the code register. Table 4 shows the memory thus required by each of these programs to hold their data structures; the numbers in parentheses indicate the adjusted sizes of the

Huffman programs if the flag is combined with one of the pointers in each node of the tree. All of these programs make more efficient use of memory than their semi-adaptive counterparts; the arithmetic coding programs' memory usage efficiency is now even more pronounced.

Executable File Name	Memory Required (in Bytes)
ahuffe	4638 (3608)
ahuffd	4638 (3608)
aarithe	522
aarithd	520
aarithee	524
aarithed	522

Table 4: Memory Requirements of Adaptive Order-0 Programs

Compression Efficiency

Each of the compression programs was run on each of the files in the Calgary compression corpus; Table 5 shows the resulting compression ratios. As is expected, the arithmetic compressor *aarithe* outperforms the Huffman compressor in nearly all cases, and *aarithee* always produces better compression. *ahuffe* and *aarithee* are able to compress better than *aarithe* in most cases because they are using a better model of the data; for most of the files, the former compressors are using a model of the data which does not waste code space on symbols which do not occur in the input. However, for files in which more of the possible source symbols occur with a more uniform frequency distribution (specifically, the files *geo*, *obj1*, and *obj2*), the overhead of encoding the escape symbol followed by a literal symbol each time a new symbol is encountered outweighs the savings achieved by maintaining a minimal code space. These results are compared with the ratios of the compressors described in other chapters and with the ratios of some publicly available compressors in a table in Chapter 7.

Execution Speed

The execution speed of these programs is generally slower than that of the corresponding semi-adaptive versions due to the fact that the adaptive programs must update their probability model after each symbol is processed. For short input files, the time savings gained by dispensing with the initial scan of the

input might compensate for this increased computational complexity; however, at some point, the adaptive algorithm will be slower than the semi-adaptive algorithm. Again, the computational overhead of the arithmetic operations performed by the arithmetic coding programs significantly slow their execution speeds in comparison with the Huffman programs, and the update of their models may take substantially longer; on average, if m symbols are present in the model, $m/2$ entries will have to be updated, while in the Huffman models, the average number of updates is the same as the average code length. This is balanced somewhat by the simplicity of the individual updates in the arithmetic model, which are simple increment operations, as compared to the updates in the Huffman model, which may involve exchanging a number of nodes in the tree. The update of the arithmetic model can be improved by sorting the symbols in descending order by frequency, decreasing the average number of increment operations. The Huffman encoder itself is significantly slower than its semi-adaptive counterpart, since the encoding process involves a tree traversal, rather than a simple table lookup.

		Output of ahuffe		Output of aarithe		Output of aarithee	
File Name	Size	Size	Ratio	Size	Ratio	Size	Ratio
bib	111,261	72,863	35%	72,793	35%	72,697	35%
book1	768,771	438,175	44%	436,923	44%	436,817	44%
book2	610,856	366,401	41%	364,788	41%	364,713	41%
geo	102,400	72,880	29%	72,407	30%	72,577	30%
news	377,109	245,825	35%	244,499	36%	244,409	36%
obj1	21,504	16,367	24%	16,040	26%	16,202	25%
obj2	246,814	190,793	23%	187,312	25%	187,458	25%
paper1	53,161	33,339	38%	33,131	38%	33,049	38%
paper2	82,199	47,652	43%	47,542	43%	47,434	43%
paper3	46,526	27,384	42%	27,392	42%	27,294	42%
paper4	13,286	7,963	41%	8,000	40%	7,905	41%
paper5	11,954	7,548	37%	7,561	37%	7,487	38%
paper6	38,105	24,019	37%	23,839	38%	23,753	38%
pic	513,224	106,284	80%	75,069	86%	74,841	86%
progc	39,611	26,029	35%	25,924	35%	25,852	35%
progl	71,646	42,970	41%	42,618	41%	42,496	41%
progp	49,379	30,320	39%	30,208	39%	30,118	40%
trans	93,695	64,909	31%	64,343	32%	64,286	32%

Table 5: Compression Efficiency of Adaptive Order-0 Programs

CHAPTER 4

HIGHER-ORDER MODELLING

As was mentioned in the first chapter, some assumptions must be made about a message source in order to compress messages which it produces. A typical assumption is that the source behaves as a Markov process whose production of symbols can be described by a state machine in which there is a probability associated with each transition between states. The algorithms presented in Chapters 2 and 3 all assume an order-0, or *memoryless*, source; that is, they assume that the probability of any symbol being produced is totally independent of which symbol or symbols have been produced immediately prior to it. In practice, many message sources are not memoryless; the probability of a symbol being produced in fact depends on which symbols have preceded it. The order of a Markov source corresponds to the number of preceding symbols which are considered in determining the probability of the next symbol. In an order-0 model of English text, for example, the letter *e* is most likely to be produced at any given time; however, if an order-1 model is considered and the preceding character was a *q*, the letter *u* is most probable. Typically, if the source is modelled as a higher order Markov process, it is possible to obtain better predictions of what the next symbol might be.

It is perhaps not immediately apparent that this is of any benefit in compression; however, consider any one of the statistical encoders discussed thus far. The ability to make a better prediction of the next symbol is equivalent to that symbol having a very high probability of being produced given the current state of the Markov state machine. If the probability of a symbol occurring is high, it is encoded using fewer bits than if the probability is low; it is therefore desirable to maximize the probability of the next symbol that will be produced at any point in the message. Again, consider a source with produces English text; if an order-0 model is used, the probability that a *u* will occur is relatively small (2.4%, according to Bell, et. al. in their book Text Compression - [BELL90], p. 34), but if an order-1 model is used and the preceding symbol was a

q, the probability is very close to one (99.1%, according to the same source). Likewise, consider an order-4 model of English text in which the string "quie" has occurred. The next symbol can be either s or t, so even if either one has an equal probability of occurring, both probabilities are still 1/2; the number of symbols occupying the code space has been reduced sufficiently so that either one can be represented using only one bit. Intuitively, it seems that increasing the order of the model should increase the probability of any symbol given that symbol's state, or context; this approach is in fact used to significantly increase the compression efficiency of statistical encoders.

Two different approaches to implementing these higher-order models have emerged; one is the accumulation of statistics based on the preceding symbols which form the context in which a symbol occurs, and the other is the simulation of a state machine and its associated probabilities which attempts to duplicate the one which is assumed to drive the Markov source. Both methods are discussed briefly in the following sections.

Context Modelling

This method of modelling the message source seems fairly intuitive; notice that the examples of higher-order models given in the previous section relied on the idea of a group of preceding symbols creating a context in which to estimate the probability of the next symbols. In an adaptive model, the probability estimates are simply the frequency counts of symbols occurring in each of the contexts. The implementation of an order-n context modeler seems fairly straightforward; maintain a separate model for each of the possible contexts that can occur and choose the appropriate one based on the last n symbols which have been processed. The problem with this is the number of models which must be maintained; the number of possible contexts which can occur in an order-n model is q^n, which obviously increases exponentially with n.

Fortunately, only a tiny fraction of the possible contexts occur in any given message; for example, if a four-symbol context is used, even a huge file would contain only a few of the 256^5 possible five-byte strings. Similarly, a limited number of symbols will ever occur in those contexts. Some mechanism is therefore required by which contexts can be introduced as needed and expanded to include new symbols. One

solution is to utilize the escape symbol mechanism described in the previous chapter. If a symbol to be compressed does not occur in the order-*n* context, the special escape symbol is encoded instead, and the reduced context of order-*n-1* is used to encode the symbol. This process continues until either the symbol is located and encoded using one of the contexts or the order-0 context encodes an escape symbol. If the order-0 model does not contain the symbol, it is encoded using the special order -1 context described previously. Once the symbol has been encoded, its count must be updated in each of the models, which requires that it be added to any context in which it was not found. As was discussed previously, if a context of order m has not been encountered yet, it is initialized to contain only the escape and EOS symbols, and the escape is then encoded accordingly.

This is the basic context modelling algorithm. There are a number of parameters of this algorithm which can be tuned; some of these are discussed in the following subsections.

Assigning Escape Probabilities

One decision which must be made is how a probability will be assigned to the escape symbol in each of the contexts. A simple method is to assign the escape a fixed count of one in each of the contexts; however, other assignments are possible which can improve compression. The probability of the escape symbol being encoded in a given context is equivalent to the probability that some symbol will not occur in that context, so the escape symbol's probability should be adjusted to accurately reflect this. A number of different methods have been proposed for calculating the escape probability, including the following (as described in Text Compression, [BELL90]):

Method A: This is the simple method discussed above; that is, one additional count is allocated above the number of times that a context has been seen to account for the occurrence of new symbols. The probability of the escape symbol is thus $1 / (C_m + 1)$, where C_m is the number of times the context has predicted a symbol (i.e., a total of all non-escape counts in the context).

Method B: In this method, no symbol is predicted until it has occurred twice in a context; this provides some amount of filtering of anomalous occurrences of symbols. The escape probability is equal to the number of symbols seen in the context, so it takes into account the first occurrence of each symbol. Its probability is given by q_m / C_m, where q_m is the number of different symbols which have occurred in context m and Cm is again the number of times the context has occurred (this is no longer the total of all escape counts, since the

first time each symbol occurs, the escape count is incremented instead of the symbol count).

Method C: This is similar to method B, but does not exclude the symbol the first time it occurs. The escape probability is thus $q_m / (C_m + q_m)$.

There are a number of other methods which could be used to compute the escape count; for instance, in The Data Compression Book, Nelson proposes a technique which accounts for several different factors, including the following: first, as the number of symbols defined in a context increases, the escape probability should decrease, since it is less likely that an unknown symbol will be seen. Second, if the probabilities of the symbols in the context are roughly equal, then the context is fairly random, and the probability that an unknown symbol could occur is relatively high; conversely, if the context is not random (as indicated by the fact that the maximum symbol count is much higher than the random count), the probability that an unknown symbol could occur should be lower. Finally, as more symbols have been encoded using the context, the probability that an unknown symbol should occur decreases. His escape probability is given by $(256 - q_m) * q_m) / ((256 *$ the highest symbol count) $* C_m)$ ([NELSON]). This and other complex escape probabilities have been proposed, but none has yet been commonly accepted and labelled "Method D".

Exclusion

The process of encoding an escape symbol and moving to a shorter context to encode a symbol is often referred to as *exclusion* in compression literature. It is so named because lower-order contexts are excluded from contributing to the prediction of a symbol if it occurs in a higher-order context. However, there are a number of associated issues which can be considered when performing this operation. One method which can be used to improve compression is to eliminate any symbols which were present in a higher-order model from the probability calculations of the order-m model. This is a consequence of the method of encoding a symbol; if it is present in a higher-order context, it will never again be encoded in the shorter contexts which are suffixes of that context, so the symbol need not be included in those contexts. In an arithmetic coder, this technique can be accomplished by keeping track of all symbols which were present in each higher-order context which produced an escape while encoding a symbol; when encoding a symbol

in the order-m context, the counts for any of these excluded symbols are subtracted from the cumulative counts used to generate the symbol's probability. In practice, this *full exclusion* technique is costly to implement and is therefore seldom used; however, an implementation of it is discussed and analyzed in Chapter 6. The alternative to full exclusion is *lazy exclusion*, which uses the escape technique but does not keep track of symbols which occurred in any higher-order contexts which produced escapes. The advantage in execution speed typically outweighs the small degradation in compression efficiency.

A related issue involves the update of the models after a symbol has been encoded. One possibility is to update the symbol's count in all contexts from orders 0 to n; this will result in the order-0 context accurately reflecting the order-0 statistics of the source. An alternative known as *update exclusion* updates only those contexts which were actually used to encode the symbol. That is, if the contexts of order $m+1$ through n encoded an escape symbol and the context of order m finally contained and encoded the source symbol, only the contexts of order m through n are updated. This can be explained by defining the lower-order contexts not as the raw symbol probabilities, but as the probabilities that those contexts will actually be required to encode a symbol which does not occur in a longer context. Obviously, this technique increases execution speed, since potentially fewer contexts must be updated for each symbol; in addition, the technique improves compression efficiency slightly.

Scaling Symbol Counts

As was stated in the previous chapters, it is necessary to limit the counts in a context's model so that they do not overflow various registers and counters. The typical method is to just divide each count by two and continue compression. A somewhat surprising side effect of this is that compression efficiency is often improved by the scaling operation; it seems that scaling essentially discards some statistical information due to truncation introduced by integer division, and that this should adversely affect compression. However, this is balanced by the fact that message sources are in fact not typically ergodic, so that their statistics do not actually converge to a steady state; instead, they continually change. The effect of scaling symbol counts is to discard some of the previous history information, which effectively gives more weight to

recent statistics. As a result, some context modelers periodically scale the counts, even if there is no danger of overflow.

Some Common Implementations

A number of different coding schemes using these context modelling techniques have been proposed; the following is a list of the significant methods which have been proposed (as summarized in Text Compression, [BELL90]):

DAFC: an early scheme which includes order-0 and order-1 contexts, but only builds order one contexts based on the most frequent characters. It uses method A to assign escape probabilities (described in [LANGDON83b]).

PPMA (for Prediction by Partial Matching, method A): uses method A to assign escape probabilities, utilizes full exclusion, and does not scale symbol counts; that is, it limits the size of the input such that none of the symbol counts can ever overflow (described in [CLEARY84b]).

PPMB: similar to method A but uses method B to assign escape probabilities (described in [CLEARY84b]).

PPMC: a newer version of PPM which has been tuned to improve compression efficiency and increase execution speed. It uses method C to assign escape probabilities, uses full exclusion and update exclusion, and scales symbol counts to eight bits (described in [MOFFAT88b]).

PPMC': a streamlined version of PPMC which uses lazy exclusion and update exclusion and discards and rebuilds the entire model whenever a memory usage limit is reached (described in [MOFFAT88b]).

WORD: separates the source symbols into "words", or alphabetic characters, and "nonwords", or nonalphabetic characters. The input stream is divided into a stream of words and a stream of nonwords; separate order-1 context modelers are used to encode each stream. Method B is used to assign escape probabilities, and lazy exclusion and update exclusion are utilized. Also, when the model reaches a predetermined maximum size, the statistics continue to be updated, but no new contexts are added (described in [MOFFAT87]).

State-Based Modelling

Obviously, even with the techniques described previously to add contexts and symbols within contexts only as necessary, high-order context modelers can use huge amounts of memory, and execution speed can be drastically reduced as the order is increased. An alternative is to use a different abstraction of the models; instead of dealing with tables of symbol counts, consider an implementation which simulates a state

machine. These implementations should be very fast; each time a symbol is processed, a single transition arc assocated with that symbol is located in the current state and followed to a new state, and the probability associated with that link is used to encode the symbol. Also, the data structures required to represent the state machine are simpler than those required to represent the context models described in the previous section.

Unfortunately, although there appear to be a number of advantages to implementing state-based modelers, they are in practice much less useful than might be expected. Only one practical implementation, referred to as Dynamic Markov Coding (DMC), has gained any popularity; this technique was developed by Cormack and Horspool (see [HORSPOOL86] and [CORMACK87] for details). It starts with a limited finite-state model and adapts by adding new states as needed. Due to the fact that it is implemented as a state machine, its execution speed is very high, and empirical results show that its compression efficiency is very similar to that of the PPMC context modeler. In fact, Bell and Moffat proved that DMC was in fact equivalent to a context modeler (see [BELL89] or [BELL90], Section 7.4).

The remainder of this thesis will concentrate on context modelling, since it performs as well as any practical implementations of state-based modelers and is more intuitive.

CHAPTER 5

MEMORY-EFFICIENT IMPLEMENTATIONS OF HIGH-ORDER MODELS

As was mentioned in the previous chapter, the remainder of this thesis deals with context modelers. These algorithms are very resource-hungry, and as is the case with many applications, there is a definite trade-off between memory usage and execution speed. The focus of this chapter is to discuss implementations which have reasonable memory requirements, in most cases at the expense of speed; specifically, the goal has been to produce programs which can execute effectively in the constrained memory space that is available on an Intel processor-based personal computer running the DOS operating system.

The algorithm chosen for implementation is the PPM scheme described in the last section of the previous chapter. It is not exactly the same as any of the versions described, although it is most similar to PPMC'. That is, it utilizes the lazy exclusion and update exclusion techniques and scales the symbol counts to eight bits; however, it uses method A to assign escape probabilities, which just sets the escape symbol's count to one. Also, the PPMC' method dumps and rebuilds the model whenever its size reaches a predefined limit; each of the algorithms presented here performs more like the WORD scheme in this regard, freezing the addition of new contexts or symbols to the model when it reaches a maximum size but continuing to use the model and to update its statistics. In no circumstances will any of the algorithms halt before the input file has been compressed because they expend their memory resources. A PPM modeler is described for both the Huffman and the arithmetic coders, and the last section evaluates their relative performance.

Memory Constraints

If the programs are to run on an Intel processor-based personal computer with only the DOS operating system, they must satisfy some stringent memory requirements. DOS requires that all programs, device drivers, and the operating system fit within 640 Kbytes of memory. Since the typical DOS environment

loads a number of device drivers into memory, a program should be able to run in less than 500 Kbytes of memory. A portion of this is required for a 64 Kbyte code segment, so the program's data should occupy no more than 400 to 450 Kbytes. In addition to this maximum size restriction, most DOS compilers, including the Borland C++ V3.1 compiler used to develop these programs, require that any single data item (i.e., a structure or an array) fit within one 64 Kbyte segment.

PPM Modeler for Huffman Coding

As was mentioned in the previous chapter, a PPM modeler is essentially a collection of models, where each is associated with a particular context. One of the first problems to be solved is that of finding the correct model given the string of symbols defining its context. There are a number of ways to do this; one is to store pointers to the different models in a multiway tree which is traversed using the symbols of the context to find the correct model. An alternative method is to store the pointers to the models in a hash table and index them using a hash value computed from the symbols in the context string; this is the technique that is used in this implementation. Collisions in the hash table are resolved using overflow lists; the array which holds the hash table is split into two sections, the first of which is directly indexed by the hash function and the second of which holds the overflow entries.

Each entry in the hash table has a corresponding entry in an array of structures used to maintain the Huffman tree information; once the desired context entry is found in the hash table, the same index locates that context's tree in the array of tree structures.

In order to efficiently use available memory, a complete set of nodes is not allocated for each tree; instead, individual nodes are allocated on an as-needed basis from a large pool of nodes. This allows each tree to occupy the smallest space possible, but it requires that the program manage the node pool. A similar technique is used to allocate the lists of leaf node pointers; the array of all 256 possible pointers is broken up into a number of small blocks, each containing a set of adjacent symbol's leaf pointers. These blocks are maintained in a large pool similar to the node pool and allocated on an as-needed basis to the individual trees; each tree maintains a list of the pointer blocks which it has acquired.

Although the mechanism used to store the Huffman trees has changed, they function identically to those used in the earlier adaptive order-0 Huffman programs (*ahuffe* and *ahuffd*). Once a tree has been located for a given context, the compressor or decompressor traverses it in exactly the same fashion, and an identical encoding is produced. One minor change is that the EOS symbol has been replaced with a *control* symbol. It is handled in a similar fashion; each tree is initialized to contain the escape and control symbols, and the control symbol's weight is assigned a fixed value of one. However, when a control symbol is encountered in the data stream, the following fixed-length bit string is a literal value representing one of the various control codes. Currently, the only control code defined for this implementation is the end-of-stream code; however, this mechanism allows the addition of other special codes, such as a command to flush the model and reinitialize it in the event that the model fills and compression performance begins to diminish, or a command to scale all of the symbol counts by two.

Model Data Structures

In the following discussion, the Ushort (unsigned short) data type is used for all pointer values; these pointers will always index an array of less than 64K elements, so the value 0xFFFF is used to represent the null pointer.

The data structures required to store the model are the hash table, the corresponding array of trees, a pool of leaf pointer blocks, and a pool of nodes. The hash table is an array of the following structures:

```
typedef struct
{
    Byte    context [MAX_ORDER];
    Byte    length;
    Ushort  next_tree;
}   Hash_T;

Hash_T  hash_table [NUM_TREES];
```

The *context* and *length* fields together define the context string which held in the entry. The *next_tree* pointer is used to resolve collisions; it points to the next table entry which hashes to the same value as the current entry. For this implementation, *MAX_ORDER* was defined to be seven, so this structure occupies ten bytes (assuming that the compiler aligns elements on a word boundary). When the *hash_table* is initial-

ized, each *next_tree* pointer is set to the null pointer and each *length* is set to 0xFF, which is used to indicate an empty table entry.

As explained previously, there is a one-to-one correspondence between hash table entries and trees; this eliminates the requirement to store a pointer to the tree in the *Hash_T* structure. In order to resolve collisions, the *hash_table* is subdivided into two sections, one indexed directly by the hash function and the other used for overflow nodes. The dividing point for these sections is the number chosen as the divisor in the hash function; according to Knuth in his book The Art of Computer Programming, Vol. 3: Searching and Sorting ([KNUTH73b]), the best hashing performance (i.e., the maximum usage of all hash entries) is obtained by making this divisor a prime number. In order to efficiently use the table, this prime number is chosen to be relatively close to the halfway point in the table.

The data structure used to store each tree's information is the following:

```
typedef struct
{
    Ushort   weight;
    Ushort   child;
    Ushort   esc_ptr;
    Ushort   ctrl_ptr;
    Ushort   last_node;
    Ushort   first_block;
}   Tree_T;

Tree_T   tree [NUM_TREES];
```

The *weight* field is the total of the counts of all symbols in the tree; this value is used to determine whether it is necessary to scale the tree's counts. The *child* field is the pointer to the root node's children in the node pool; as was explained in Chapter 3, sibling nodes in a Huffman tree are always adjacent, so only one pointer is required to locate them. The fields *esc_ptr* and *ctrl_ptr* index the nodes containing the escape and control symbols, respectively. The *last_node* field points to the lowest-weight node in the tree; this is the node which will be split if a new symbol is to be added to the tree. The *first_block* field is a pointer to the first block of leaf node pointers which have been allocated to the tree from the block pool; the remainder of the leaf pointer blocks follow this one in a linked list. The size of the structure is thus 12 bytes; in order to fit the tree array into a 64 Kbyte segment, *NUM_TREES* must be no greater than (64K / 12), or 5461. This value is used in the implementation. This is the maximum number of contexts for which trees can be created;

in practice, the hash function may not hit every entry in the first section of the table, so the number of usable entries might be slightly lower than this value.

The data structure used to store each block of leaf pointers is the following:

```
typedef struct
{
    Byte    block_num;
    Ushort  next_block;
    Ushort  leaf_ptr [BLOCK_SIZE];
}   Block_T;

Block_T  block [NUM_BLOCKS];
```

The *block_num* field is used to determine which symbols' leaf pointers are stored in the block; this is calculated by dividing the symbol's value by *BLOCK_SIZE* using integer division. The *next_block* field is the pointer to the next block in the linked list for a tree, and the *leaf_ptr* array holds the actual pointers to each of the leaf nodes for the symbols contained in the block. This array is indexed by dividing the symbol's value by *BLOCK_SIZE* using modulo division. The blocks are stored in the linked list in ascending order sorted by *block_num*, so it is not necessary for the entire list to be searched to determine that a block is not present.

It is desirable to maximize the number of blocks available in the pool; if there are no more blocks available, symbols which do not fit within one of the blocks already allocated cannot be added to a tree. The depletion of the free block pool thus freezes the addition of many new nodes to the trees in the model, which adversely affects the compression ratio. Since the *block* array must fit within a single 64 Kbyte segment, the following measure is taken to increase the number of blocks.

```
typedef struct
{
    Byte    block_num;
    Ushort  next_block;
}   Block_T;

Block_T  block [NUM_BLOCKS];
Ushort   block_leaf [NUM_BLOCKS][BLOCK_SIZE];
```

Each *Block_T* now requires four bytes and is independent of *BLOCK_SIZE*; this allows up to 16K entries. The limiting factor now becomes the *block_leaf* array, whose size is governed by the value of *BLOCK_SIZE*. Initially, this was set to 16; however, it was experimentally determined that smaller block sizes produce much better results, since that increases the number of available blocks. Also, it should be

noted that the storage of the leaf pointers in these blocks implies that some space in each block will be wasted. For instance, if a tree contains symbols with values 0x41 and 0x61 and *BLOCK_SIZE* is less than 32, the symbols' leaf pointers will be stored in two separate blocks, and the other leaf pointers in the blocks will be unused. Smaller values of *BLOCK_SIZE* reduce the amount of space that is wasted. The use of smaller blocks adversely affects execution speed, since location of a symbol's leaf pointer requires a linear scan of the linked list of block pointers and this list usually grows longer if the blocks are smaller. However, the emphasis is on memory efficiency rather than execution speed, so this compromise is accepted. Note that the limit of this compromise is a block size of one; this would require six bytes for each leaf pointer (one for the block number, which is now the same as the symbol value), one for the next block or next symbol pointer, and one for the leaf pointer. This would require more memory than the previous blocking scheme and would decrease the execution speed even more, so the final value chosen for the implementation was four. The maximum value of *NUM_BLOCKS* is 64K / (2 * *BLOCK_SIZE*), or 8K. This value is used for the implementation.

When the block pool is initialized, all blocks are linked together via the *next_block* pointer to form a free block list. This list is used to manage the allocation of new blocks to trees as required. It would not be necessary to manage the free blocks as a list; a *next_available_block* index could be kept instead. However, organizing the block pool as a free block list allows trees to be deleted and their blocks to be returned to the pool and reused. While the current implementation does not do this, the capability is available if desired in future enhancements.

As was explained in Chapter 3, sibling nodes in the Huffman tree are always adjacent. In order to save space, the structure which holds a node is modified to hold a pair of sibling nodes; the data structure is as follows:

```
typedef struct
{
    Ushort    parent;
    Ushort    child [2];
    Ushort    count [2];
    Ushort    next_node;
}   Node_T;

Node_T  node [NUM_NODES];
```

The *parent* and *child* fields are the pointers to the pair of nodes' single parent and four child nodes, and the *count* array contains the cumulative counts or node weights for the two nodes. The *next_node* pointer is used to link together adjacent pairs of nodes in the tree; since the nodes are allocated to trees from a free node pool, this linked list is required to be able to traverse the node list when updating the tree.

One consequence of combining siblings into a single *Node_T* structure is that it is now necessary to distinguish between the two halves of a node when it is indexed by a pointer (i.e., the *parent* pointer or the leaf pointers stored in the block pool). If it is guaranteed that the pointers will never need to index more than 32K nodes, the MSB of the pointer can be used to index the *child* and *weight* arrays, thus selecting the desired half of the pair. The 64K segment limitation effectively imposes this 32K element limitation on the *node* array size, so each leaf or parent pointer's MSB is used to select the half of a node pair. In addition, this bit is used in the *child* pointers as the flag indicating that the child is a leaf and the pointer contains the symbol's value.

In order to maximize the number of nodes available in the node pool, the individual fields of the Node_T structure are split into separate arrays, as follows:

```
Ushort   node_parent [NUM_NODES];
Ushort   node_child [NUM_NODES][2];
Ushort   node_count [NUM_NODES][2];
Ushort   node_next [NUM_NODES];
```

The maximum value of *NUM_NODES* is 64K / 4, or 16K; this value is used in the implementation. As is done with the block pool, during the initialization phase the elements of *node_next* are all linked together into a free node list.

As an aside, after the idea of combining sibling nodes into one structure and the use of one bit in each pointer to select the appropriate half of the pair was conceived, implemented, and tested, it was discovered that Gallager had proposed a nearly identical implementation in 1978 (see [GALLAGER]).

Fig. 16 shows an example of these data structures being used to contain the tree shown at the top. This tree is associated with the context string "ij"; as shown, this context string collided with another one already in *hash_table* and was added to the overflow list. The corresponding *tree* array entry is the root node for the tree; the figure shows the various pointers linking the tree, its blocks of leaf pointers, and its nodes.

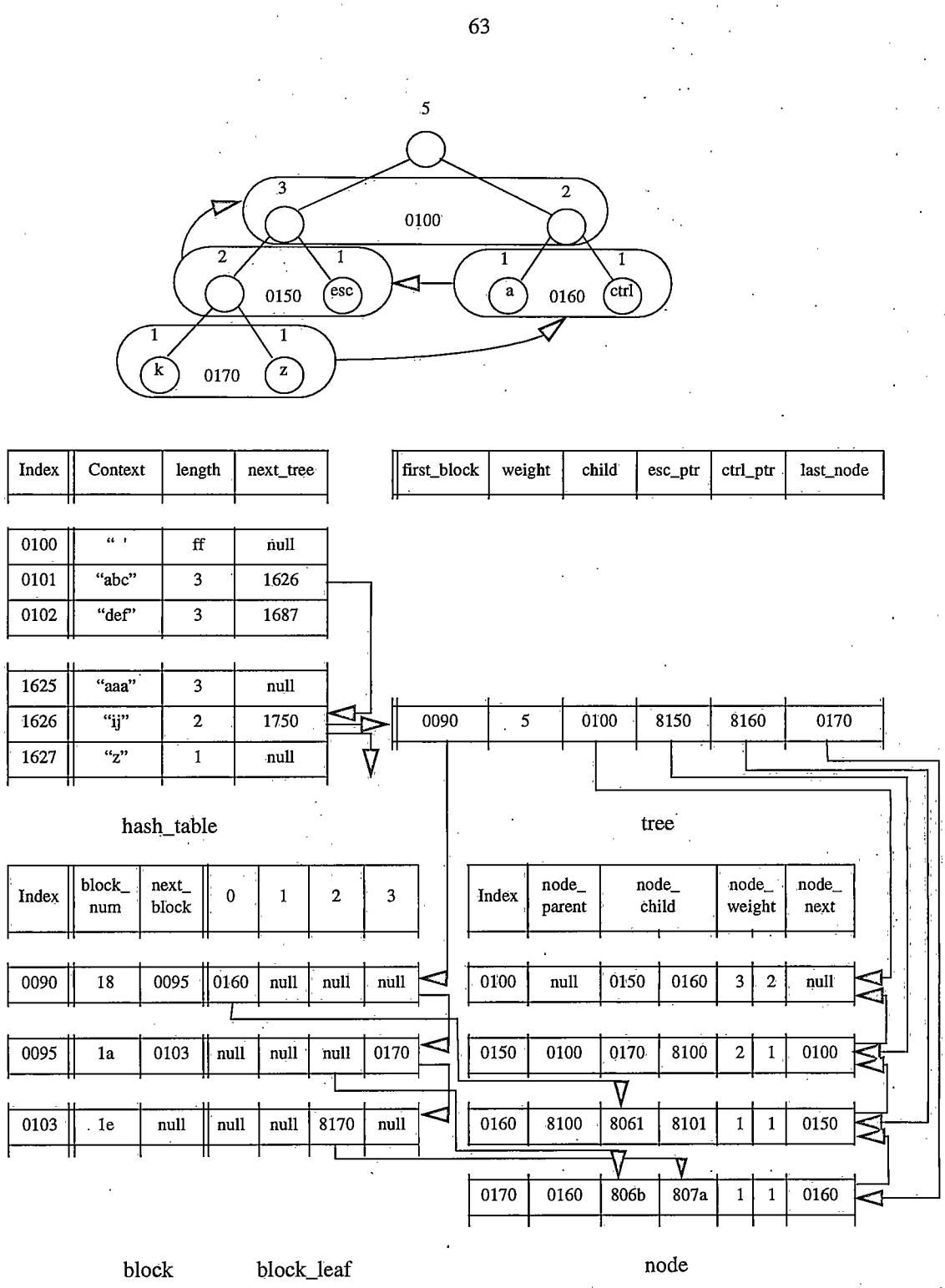

Figure 16: Example of Data Structures Used to Store Huffman Tree

Modifications to the Adaptive Order-0 Algorithms

Obviously, some changes are required to the order-0 adaptive Huffman algorithms presented in Chapter 3 to utilize these new data structures. The major difference is that it is no longer possible to directly index a node or a leaf from the node or leaf pointer arrays; instead, linked lists must be traversed to locate a leaf node or scan the node list when updating the tree.

Other modifications are required to add support for higher-order contexts. When the routine to code a symbol is invoked, it initially searches the hash table for the context string accumulated thus far, using as a length the order which was specified when the program was invoked. If the tree is not present in the table, it is added and initialized to contain only the escape and control symbols. Once this is done, the block list is searched to find the appropriate block and the desired leaf pointer within that block. If this pointer is null, then the symbol is not in the tree, so the escape symbol (located using the *esc_ptr* field in the *tree* entry) is encoded; otherwise, the symbol is encoded. If an escape is encoded, the length of the context is reduced by one and the process of locating a tree and encoding the symbol using that tree repeats. If the length is reduced to zero and an escape is still encoded, the symbol's value is output (i.e., this is the order -1 model). The technique for encoding a symbol is the same as that used in the order-0 algorithms, although each time a *parent* pointer is processed, the MSB must be stripped and used to select the correct half of the node pair indexed by the pointer's lower fifteen bits.

Similar changes are required to the decoder. It uses the same steps to locate the tree associated with the current context, then decodes the symbol. If the decoded result is the escape symbol, the length is reduced and the process is repeated; if an escape is decoded in the order-0 context, the next eight bits are read as the value of the symbol. Again, the technique for decoding a symbol is the same as that used in the order-0 version, except that the leaf flag must be retrieved from the MSB of the *child* pointer.

Both the encoder and the decoder use the same routine to update the model. This routine is similar to the order-0 version as well, but it must update the tree for each context from the order-*n* one to the one that was actually used to encode the symbol. It is responsible for adding new blocks and new nodes to the model, although new trees are added in the encode and decode routines. Once the appropriate trees have been

updated, the context string is updated by shifting it over one symbol and adding the symbol which was just processed to the end.

In order to make these algorithms robust, they must both handle the depletion of the pools of trees, blocks, or nodes, and they must handle these occurrences in the same way so that the models remain in sync between the encoder and decoder. If an attempt is made to locate a tree in the encoder or decoder and a new tree cannot be added (because the overflow space in the hash table has been exhausted), the order is reduced by one automatically; note that in this case an escape cannot be generated, since there is no tree which can be used to encode it. If the encoder is unable to locate a block containing a symbol to be encoded, this implies that the symbol is not in the model and the escape symbol is encoded. In the update routine, if a block cannot be allocated to hold the leaf pointer for the new symbol, the symbol cannot be added to the tree. Likewise, if the node pool is exhausted, the symbol cannot be added. This will result in further occurrences of the symbol in the affected context again being encoded using an escape to a lower-order context, but it will not prevent the algorithm from continuing to process data.

Implementation Details

Implementations of these compression and decompression algorithms are contained in the files *ahuff_n.h*, *ahuff_n.c*, and *ahn[1-6].c* (for information on obtaining this source code, see the last section of Chapter 1). It was necessary to separate the definition of the large data structures used (i.e. the *hash_table, tree, block, block_leaf, node_parent, node_child, node_count*, and *node_next* arrays) into separate files; in addition to the 64K limit on the size of an array, the Borland C++ compiler imposes a similar limit on the size of the data defined in any single source file. These files were used to create the *ahuffe* and *ahuffd* programs; both of these programs have been debugged and tested. Their memory usage and compression efficiency are compared to those of the adaptive arithmetic coding programs in the last section of this chapter.

PPM Modeler for Arithmetic Coding

The PPM modeler for the arithmetic coder is very similar to the one used with the Huffman coder. The same hash table scheme is used to locate the arithmetic coding module for a particular context, and the

same block scheme is used to store the symbol counts for each model. There is no need for a node pool, so the required data structures are the hash table, the corresponding model array, and a pool of symbol count blocks. The EOS symbol was again converted to a control symbol and an EOS control code for this implementation; a special control model is used to encode the control code after a control symbol is encoded. This model is similar to the order -1 model; each control code is assigned a count of one and the total count is equal to the number of control codes. Since the model is so simple, it is not stored in memory but computed when needed.

Model Data Structures

The *Hash_T* structure used in the Huffman algorithm is split into two sections in order to provide models for more possible contexts. The modified structure is as follows:

```
typedef struct
{
    Byte    context [MAX_ORDER];
    Byte    length;
}   Hash_T;

Hash_T  hash_table [NUM_MODELS];
Ushort  hash_next_model [NUM_MODELS];
```

Again, roughly half of the hash table is directly indexed by the hash function and the other half is used for overflow nodes; since the size of the table is increased, a new prime number must be chosen to divide the table.

Each model is stored in the following data structure:

```
typedef struct
{
    Ushort  weight;
    Ushort  first_block;
}   Model_T;

Model_T model [NUM_MODELS];
```

The *weight* field is again the total count of all symbols in the model, including the escape and control symbols. The escape and control symbol counts are always assigned the value of one, so they are not stored in the model's count blocks. Their cumulative counts are always *weight* - 1 and *weight* - 2, respectively. The *first_block* field is a pointer to the head of the linked list of blocks containing the model's symbol counts; if it is null, the model contains only the escape and control symbols, and *weight* must be two.

The size of this structure is four bytes, so the maximum value of *NUM_MODELS* should be 64K / 4, or 16K; however, the same value dictates the size of *hash_table*, and the size of the *Hash_T* structure is eight bytes. This gives a maximum value of 64K / 8, or 8K, for *NUM_MODELS*; this value is used in the implementation. This is one and a half times the number of trees available in the Huffman modeler, so the modeler is capable of storing the statistics for more contexts.

The blocks of symbol counts are stored in the following structure:

```
typedef struct
{
    Byte    block_num;
    Byte    count [BLOCK_SIZE];
    Ushort  cum_count;
    Ushort  next_block;
}   Block_T;

Block_T  model [NUM_BLOCKS];
```

The *block_num* field is the same as that used in the Huffman algorithm's leaf pointer blocks; its value is the value of the symbols it contains divided by *BLOCK_SIZE*. The *count* array contains the frequency counts of the symbols contained in the block; these are the individual symbols' counts, not cumulative counts. To prevent the encoder and decoder from scanning the block list from the block containing the desired symbol to the end of the list adding counts to find a cumulative value, the *cum_count* field contains the cumulative total for the symbols for all blocks following the current one in the list. To find the cumulative count for a symbol, it is only necessary to add each element of the *count* array from the one for the desired symbol to the last one in the array together with the *cum_count* value. This allows the cumulative total to be computed relatively quickly without requiring a short value to store each count. The *next_block* field points to the next block in the linked list; the blocks are sorted in ascending order by *block_num*.

In order to increase the number of blocks that will fit into a 64K segment, this structure is broken into pieces as follows:

```
typedef struct
{
    Byte    block_num;
    Ushort  next_block;
}   Block_T;

Block_T  block [NUM_BLOCKS];
Byte     block_count [NUM_BLOCKS][BLOCK_SIZE];
Ushort   block_cum_count [NUM_BLOCKS];
```

A block size of four was chosen for the same reason it was used in the Huffman algorithms. For this value of *BLOCK_SIZE*, the maximum value of *NUM_BLOCKS* is 64K / *BLOCK_SIZE*, or 16K; this value is used in the implementation.

Fig. 17 shows an example of these data structures used to contain the model corresponding to the Huffman tree shown in Fig. 16. The *hash_table* is identical to the one generated in the Huffman program; the figure shows how this identifies the model and associated blocks of symbol counts.

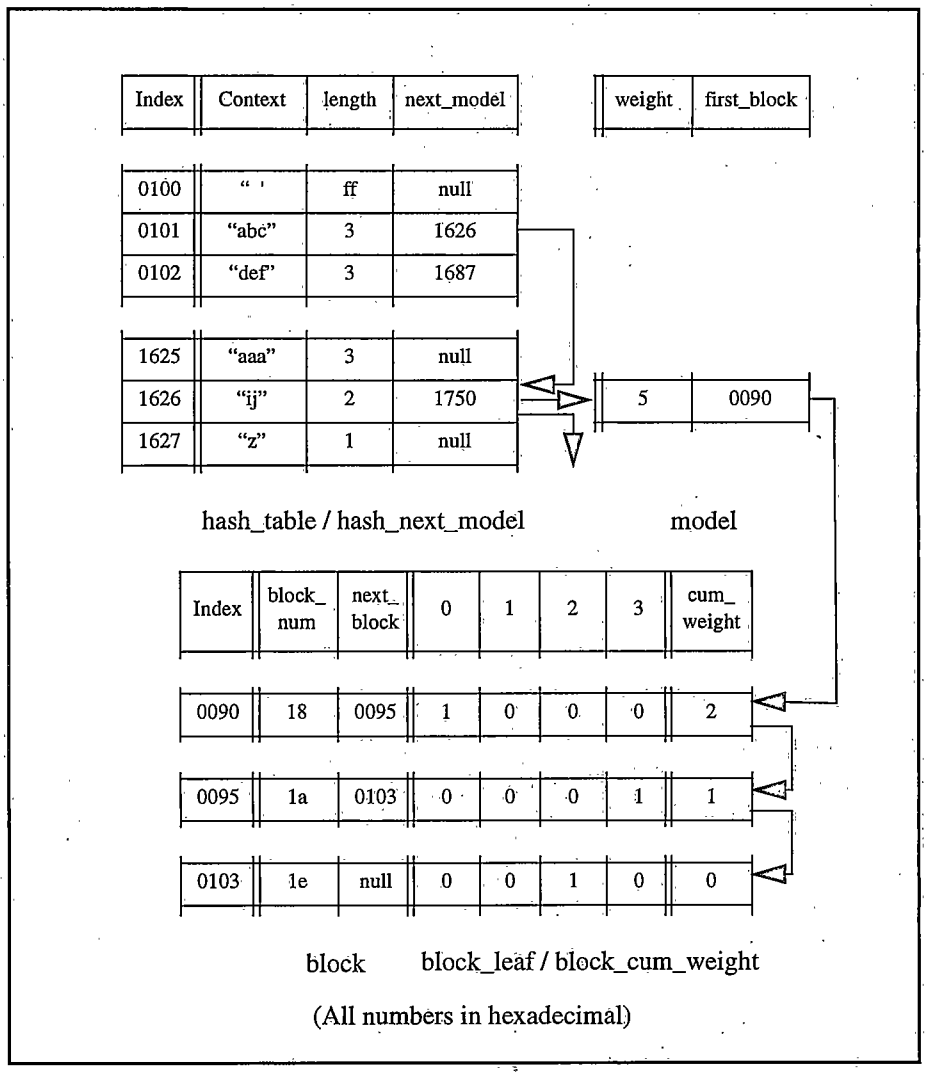

Figure 17: Example of Data Structures Used to Store Arithmetic Coding Models

Modifications to the Adaptive Order-0 Algorithms

The modifications to the order-0 arithmetic coding algorithms are very similar to the modifications that were required for the Huffman algorithms. That is, the encoder and decoder must continue to select the model associated with the context of decreasing length until either a non-escape symbol is encoded or decoded, or a length of -1 is reached. When the context length reaches -1, the same order -1 model used in the order-0 algorithm is used to encode or decode the symbol. Also, if a control symbol is encoded or decoded, the special control model is used to encode or decode the value of the control code following it.

The update routine requires a similar modification to update the models for each of the contexts from the one that encoded the symbol to the one of maximum length. The process to update each model is similar to that used in the order-0 model; however, once a count is incremented, only the *block_cum_count* field of each of the blocks which precede it in the list must be incremented, since the counts themselves are not cumulative.

These routines are required to gracefully handle the situation where either the model pool or the block pool is exhausted; they react similarly to the Huffman algorithms. If a tree cannot be created for a given context, the order is reduced by one automatically without generating an escape symbol. If a block cannot be allocated to add a new symbol to a model, the add is ignored, with the same consequences as the failure to add a new node to a Huffman tree.

Implementation Details

Implementations of these compression and decompression algorithms are contained in the files *arith_n.h*, *arith_n.c*, and *aan[1-5].c* (for information on obtaining this source code, see the last section of Chapter 1). It was necessary to separate the definition of the large data structures used (i.e. the *hash_table, hash_next_model, model, block, block_counter,* and *block_cum_weight* arrays) into separate files due to the limit of 64K on the data space used by any single source file. These routines were used to create the *ahuffe* and *ahuffd* programs; both of these programs have been debugged and tested. Their memory usage and compression efficiency are compared to those of the adaptive arithmetic coding programs in the last section of this chapter.

Comparison of the Different Implementations

This section presents an analysis of the memory usage and the compression efficiency of each of the programs described in this chapter, along with some observations relating to the relative speed of the different programs. The programs being evaluated are *ahuffne* and *ahuffnd*, the adaptive order-*n* Huffman compressor and decompressor, and *aarithne* and *aarithnd*, the adaptive order-*n* arithmetic coding compressor and decompressor.

Memory Usage

For both the Huffman and arithmetic coding programs, the compressor and decompressor have the same memory requirements, since the decompressor maintains the same model as the compressor. The data structures used by the Huffman coder are shown in Table 6, along with their sizes. This memory usage fits within the stated bounds and leaves ample space for the program's stack and code segment.

Data Structure	Memory Required (in Kbytes)
node_parent	32
node_child	64
node_count	64
node_next	32
hash_table	54
tree	64
block	32
block_leaf	64

Total	406

Table 6: Memory Requirements of Adaptive Order-*n* Huffman Programs

The data structures used by the arithmetic coder are shown in Table 7, along with their sizes. The arithmetic coder uses a significantly smaller amount of memory than the Huffman coder; however, due to the simplicity of its models, it actually has more available space for context models and symbols within those models.

Data Structure	Memory Required (in Kbytes)
hash_table	64
hash_next_model	16
model	32
block	64
block_counter	64
block_cum_weight	32

Total	272

Table 7: Memory Requirements of Adaptive Order-*n* Arithmetic Coding Programs

Compression Efficiency

Each of the compression programs was run on each of the files in the Calgary compression corpus; Tables 8, 9, and 10 show the resulting compression ratios when the programs are run with orders of 1, 2, and 3, respectively. The *h*, *b*, and *n* symbols which appear between the size and ratio columns indicate that the compressor exhausted the space in its hash table, block pool, or node pool, respectively, while compressing the file; the order of the symbols indicates the order in which the individual resources were exhausted. Experiments have shown that increasing the order beyond three drastically decreases the compression ratio on most input files because the model resources are exhausted too quickly.

There are two consequences of freezing the models which adversely affect the compression ratio. The first is the fact that the model can no longer adapt to changes in the data; if the model is filled too quickly, it may not get sufficiently close to the characteristics of the message source to ever compress well. The second, and probably the overriding, factor is that it is no longer possible to add symbols to the higher contexts if the block or node pools are exhausted, so a large number of extraneous escape symbols will be generated. As an experiment, the size of each of the data structures used in the arithmetic coder, *aarithne*, was quadrupled, and the program was recompiled on a computer running the Unix operating system and providing virtual memory support. The performance of this program when run with orders of three and four is shown in Table 11. It can be seen that even though the modeler does not deplete its resources on some of the files, it actually performs worse than the version with the smaller model space. Part of this is due to the fact that the modeler now must produce additional escape sequences to add new symbols and new contexts

to the model; in the original version, when the hash table is full and a new context is to be added, no escape is generated before the next lower order context is used, so a savings is obtained. It is possible that the use of a different method of assigning the escape probabilities (i.e., method B or method C) might result in a more efficient encoding of the escape symbols and a corresponding improvement in the compression ratio of the higher-order models and the models with a larger address space. However, if method A is used, the results seem to indicate that the version which can execute in a smaller memory space executed with an order of three is probably a suitable fit for compressing many data files.

As is expected, the arithmetic compressor outperforms the Huffman compressor in nearly all cases; the normal difference between the coding algorithms is enhanced by the fact that the arithmetic coder typically takes much longer to fill up its models, so its model is able to adapt more closely to match the characteristics of the source. These results are compared with the ratios of the compressors described in other chapters and with the ratios of some publicly available compressors in a table in Chapter 7.

Execution Speed

The execution speed of these programs is substantially slower than that of the order-0 adaptive versions, primarily because the higher-order models must be used and updated. Also, the data structures have been modified in such a way that more computation is required to manipulate the models. The increased overhead of the bit manipulations on the pointers in the Huffman coder actually makes it perform slower than the arithmetic coder on an IBM RS/6000 workstation. This result may be somewhat less apparent or may be reversed on other computers which may not perform as strongly on arithmetic operations, but it appears that the arithmetic coder is probably the best choice all around; it requires less memory than the Huffman coder, or alternatively provides space for more contexts in the same memory, its compression efficiency is better, and it actually executes faster.

File Name	Size	Output of ahuffne Size		Ratio	Output of aarithne Size	Ratio
bib	111,261	49,421		56%	48,278	57%
book1	768,771	350,101		55%	346,893	55%
book2	610,856	292,014		53%	285,847	54%
geo	102,400	68,090	b	34%	66,185	36%
news	377,109	199,009		48%	195,746	49%
obj1	21,504	14,202		34%	13,818	36%
obj2	246,814	133,446		46%	128,097	49%
paper1	53,161	25983		52%	25,501	53%
paper2	82,199	37,845		54%	37,305	55%
paper3	46,526	22,064		53%	21,782	54%
paper4	13,286	6,632		51%	6,552	51%
paper5	11,954	6,266		48%	6,186	49%
paper6	38,105	18,777		51%	18,465	52%
pic	513,224	98,442		81%	75,110	86%
progc	39,611	19,706		51%	19,286	52%
progl	71,646	30,599		58%	29,460	59%
progp	49,379	21,377		57%	20,866	58%
trans	93,695	42,530		55%	40,659	57%

Table 8: Compression Efficiency of Adaptive Order-n Programs, for n=1

File Name	Size	Output of ahuffne Size		Ratio	Output of aarithne Size		Ratio
bib	111,261	41,322		63%	38,715		66%
book1	768,771	289,625	b	63%	282,218		64%
book2	610,856	239,46 7	bn	61%	223,665		64%
geo	102,400	82,204	bhn	20%	78,038	bh	24%
news	377,109	186,052	bn	51%	163,737	b	57%
obj1	21,504	15,511	bh	28%	13,986		35%
obj2	246,814	212,512	bhn	14%	168,798	hb	32%
paper1	53,161	21,601		60%	20,885		61%
paper2	82,199	31,869		62%	30,889		63%
paper3	46,526	19,719		58%	19,252		59%
paper4	13,286	6,251		53%	6,110		55%
paper5	11,954	5,878		51%	5,746		52%
paper6	38,105	15,925		59%	15,415		60%
pic	513,224	103,675	bn	80%	76,219		86%
progc	39,611	16,637		58%	15,947		60%
progl	71,646	24,253		67%	22,625		69%
progp	49,379	16,673		67%	15,339		69%
trans	93,695	32,162		66%	29,362		69%

Table 9: Compression Efficiency of Adaptive Order-n Programs, for n=2

		Output of ahuffne			Output of aarithne		
File Name	Size	Size		Ratio	Size		Ratio
bib	111,261	43,288	bhn	62%	35,225	hb	69%
book1	768,771	379,836	bhn	51%	288,419	bh	63%
book2	610,856	315,844	bhn	49%	232,996	hb	62%
geo	102,400	93,511	hbn	9%	93,585	hb	9%
news	377,109	243,466	bhn	36%	184,608	hb	52%
obj1	21,504	17,575	hb	19%	14,763	h	32%
obj2	246,814	271,496	hbn	-10%	178,464	hb	28%
paper1	53,161	26,038	bh	52%	20,563	h	62%
paper2	82,199	35,999	bhn	57%	29,605	hb	64%
paper3	46,526	21,469	bhn	54%	19,369		59%
paper4	13,286	6,801		49%	6,609		51%
paper5	11,954	6,424		47%	6,257		48%
paper6	38,105	19,033	bh	51%	15,486	h	60%
pic	513,224	113,232	hbn	78%	78,610	hb	85%
progc	39,611	21,108	bh	47%	16,165	h	60%
progl	71,646	27,940	bh	47%	20,472		72%
progp	49,379	18,708	bh	47%	14,475		71%
trans	93,695	39,081	bh	47%	25,322	h	73%

Table 10: Compression Efficiency of Adaptive Order-*n* Programs, for *n*=3

		Output for order-3			Output for order-4		
File Name	Size	Size		Ratio	Size		Ratio
bib	111,261	35,174		69%	38,204		66%
book1	768,771	250,880		68%	278,163	hb	64%
book2	610,856	188,321		70%	202,052	hb	67%
geo	102,400	98,701	bh	4%	100,971	hb	2%
news	377,109	152,371	b	60%	168,246	hb	56%
obj1	21,504	15,515		28%	17,280		20%
obj2	246,814	112,423	hb	55%	112,782	hb	55%
paper1	53,161	20,607		62%	23,477		56%
paper2	82,199	29,608		64%	33,539		60%
paper3	46,526	19,369		59%	22,378		52%
paper4	13,286	6,609		51%	7,747		42%
paper5	11,954	6,257		48%	7,279		40%
paper6	38,105	15,491		60%	17,785		54%
pic	513,224	80,024		85%	83,575	hb	84%
progc	39,611	16,261		59%	18,611		54%
progl	71,646	20,472		72%	22,254		69%
progp	49,379	14,475		71%	16,162		68%
trans	93,695	25,343		73%	26,549		72%

Table 11: Compression Efficiency of Arithmetic Coder with Expanded Model Memory

CHAPTER 6

FURTHER ENHANCEMENTS TO IMPROVE COMPRESSION

There are obviously a number of different parameters of the order-*n* adaptive algorithms which can be tuned to affect compression performance. This chapter discusses a few optimizations which attempt to improve the compression of the adaptive order-*n* arithmetic coder; similar modifications are not explored for the Huffman coder in light of the fact that the arithmetic coder seems to outperform it in all areas.

Translating the Symbol Set to Improve Block Utilization

One problem which adversely affects the compression efficiency of the adaptive coder is the depletion of its block and model pools. As was mentioned in the previous chapter, there are some inefficiencies involved in blocking the symbol counts; it is likely that some of the counts in many of the blocks will be unused. If the algorithm could be improved to more efficiently utilize the space in each block, fewer blocks would be required per tree and the block pool would be exhausted less quickly.

Since symbols are assigned to blocks on the basis of their binary values, one method which would seem to alleviate this problem would be to translate their values so that symbols which are likely to occur in a given context are close to each other (i.e., they have binary values which are close). This translation is straightforward using a 256-entry forward translation table and a corresponding 256-entry reverse translation table in the decompressor. This technique was implemented, and a number of experiments were performed with different translation tables.

One problem is that a translation table which works well for grouping the symbols from English text may work very poorly for grouping symbols from an executable binary image. However, an assumption was made that the compressor would be able to somehow choose a translation table which would work well for a given input file. If English text is considered, the most obvious organization of this table is to sort the

symbols in descending order of frequency, according to the standard distribution for the English language. This approach provided disappointing results, however; for some input files, the version which translated the input alphabet actually exhausted its block pool faster than the unmodified version. In no circumstance was a substantial improvement observed. After a number of attempts at reorganizing the translation table, this technique was abandoned.

This failure can be explained by considering the reasons why an order-1 model compresses better than an order-0 model; in each of the order-1 contexts, the distribution of symbols is potentially different than the distribution in other contexts, so a single translation table is likely to adversely affect block usage in some contexts while it improves usage in others. It appears that if only a single symbol ordering is to be used, the ASCII ordering works as well as any other.

Scoreboarding

Scoreboarding is another term for the technique of *full exclusion* described in Chapter 4. That is, for each context that is used to encode a symbol, any symbols that were present in any higher-order contexts that did not contain the symbol being encoded are excluded from the probability calculations. An implementation of this technique requires a *scoreboard*, or list of flags, one for each input symbol. Each flag is initialized to false when the highest-order context is searched for an input symbol; if the symbol is not found, the flag corresponding to each symbol that is present in the context is set. This procedure is repeated for each lower-order context used; if the context does not contain the input symbol, additional flags are set for each symbol present in the context. An example of the use of scoreboarding is shown in Fig. 18.

For each of the lower-order contexts, the associated model's probabilities must be adjusted using the scoreboard information. As with the other algorithms, it is first determined whether the symbol is present in the model; if not, the escape symbol is encoded instead. Once the symbol is known, its probability range is calculated as it was for the version without scoreboarding; that is, its low range is the cumulative total of all symbols before it, its high range is its low range plus its count, and its scale is the total weight of the model. This probability range is then adjusted as follows; the count for each of the symbols present in the

Context / Total Weight	Symbol / Weight							
ABC - 5	I - 3							
BC - 8	I - 1	J - 5						
C - 12	I - 4	J - 2	M - 3	N - 1				
"" - 70	A - 10	B - 12	C - 11	I - 9	J - 13	M - 6	N - 5	Z - 2

Encoding the symbol Z in the context ABC will produce the following output:

	ESC - 1/5		ESC - 1/8		ESC - 1/12		Z - 2/70	
Optimal length:	2.32	+	3	+	3.58	+	5.13	= 14.03 bits

Applying scoreboarding to the model when encoding the symbol Z will produce the following output:

Context / Total Weight	Symbol / Weight							
ABC - 5	I - 3							
BC - ~~8~~ 7	~~I - 1~~	J - 5						
C - ~~12~~ 6	~~I - 4~~	~~J - 2~~	M - 3	N - 1				
"" - ~~70~~ 37	A - 10	B - 12	C - 11	~~I - 9~~	~~J - 13~~	~~M - 6~~	~~N - 5~~	Z - 2

	ESC - 1/5		ESC - 1/7		ESC - 1/6		Z - 2/37	
Optimal length:	2.32	+	2.81	+	2.58	+	4.21	= 11.92 bits

which is a 15% decrease in the output length for this symbol.

Figure 18: Example of the Use of Scoreboarding

model whose flag is also set in the scoreboard is first subtracted from the scale. Then, if the symbol to be encoded is the escape, its high and low count values are set to the new scale value minus one and minus two, respectively; otherwise, the count for each symbol which follows the symbol to be encoded in the list of counts and whose flag is also set in the scoreboard is subtracted from the low and high counts. This adjusted probability is then used to encode the symbol. The scoreboarding process causes significant deterioration in the execution speed since it is no longer possible to use the cumulative totals stored for each block; the list of symbol counts must always be completely scanned in order to find the counts of all symbols which are flagged in the scoreboard.

The decoding process is slightly more difficult. The scoreboard is initialized and updated in the same way; each time a context is used to decode an escape, the flags corresponding to all of the symbols present in the context are set in the scoreboard. When each lower-order context is used to decode a symbol, the weight of its model is first adjusted by subtracting the counts of the symbols flagged in the scoreboard.

This adjusted weight is then used to calculate the target value which is to be decoded. The list of symbol counts is scanned to locate this value; the cumulative count for each symbol is calculated by subtracting its count from the cumulative count of the previous symbol, ignoring those symbols flagged in the scoreboard, until the symbol whose low and high counts bracket the target value is found. As was the case in the encoding algorithm, adding scoreboarding slows execution because the cumulative block counts can no longer be used. The decoding algorithm is somewhat slower than the encoding algorithm because the list must be fully scanned once to adjust the model's weight and then scanned again until the correct symbol is found.

This increase in complexity does result in improved compression efficiency. Table 12 shows a comparison of compression ratios for the program run on the files in the Calgary compression corpus; they are run with and without scoreboarding for orders of one, two, and three. The ratios for the runs without scoreboarding are the same as those given in Tables 8 through 10 in Chapter 3. The improvement is typically less than one percent for order one but usually increases to two or more percent for order three. This is actually a fairly significant increase, although it is gained at the expense of a substantial decrease in execution speed (a factor of seven decrease was observed on an IBM RS/6000).

Dynamic Order Adjustment

Another problem with using higher-order models which was revealed in the previous chapter is the expense of encoding the escape symbol for each of the higher-order contexts until the model adapts sufficiently to the characteristics of the input. Although it may be possible to shorten the number of bits to encode an escape by using a different method of assigning escape probabilities, when the encoder begins processing, it must encode a large number of escape symbols. An approach which can be used to moderate this is the dynamic adjustment of the maximum order as the encoder runs. That is, if the encoder is invoked with a maximum order of three, it initially encodes symbols using only the order-0 context. Although it does not use any of the higher-order contexts for encoding, it updates them as if they were used. At some point, when the model has been developed sufficiently, the encoding order can be increased to one. This continues until the maximum context is reached.

		Order one ratios		Order two ratios		Order three ratios	
File Name	Size	w/o SB	w/ SB	w/o SB	w/ SB	w/o SB	w/ SB
bib	111,261	57%	57%	66%	66%	69%	71%
book1	768,771	55%	55%	64%	64%	63%	65%
book2	610,856	54%	54%	64%	64%	62%	64%
geo	102,400	36%	37%	24%	28%	9%	17%
news	377,109	49%	49%	57%	58%	52%	54%
obj1	21,504	36%	37%	35%	38%	32%	35%
obj2	246,814	49%	49%	32%	34%	28%	30%
paper1	53,161	53%	53%	61%	62%	62%	64%
paper2	82,199	55%	55%	63%	64%	64%	66%
paper3	46,526	54%	54%	59%	60%	59%	62%
paper4	13,286	51%	52%	55%	56%	51%	54%
paper5	11,954	49%	49%	52%	54%	48%	51%
paper6	38,105	52%	52%	60%	61%	60%	62%
pic	513,224	86%	86%	86%	86%	85%	86%
progc	39,611	52%	52%	60%	61%	60%	62%
progl	71,646	59%	60%	69%	70%	72%	73%
progp	49,379	58%	58%	69%	70%	71%	73%
trans	93,695	57%	57%	69%	70%	73%	75%

Table 12: Compression Efficiency of Arithmetic Coder with and without Scoreboarding

The implementation is fairly straightforward; the encoder sets its order to the temporary value being used for processing and encodes the symbol, and it then returns the order to the maximum value specified on invocation and updates the models. In order for this approach to be effective, a heuristic must be developed which governs the points at which the encoder increases the order used to actually encode the data. The decoder must use the same heuristic to dynamically adjust the order while decompressing data in order to maintain synchronization with the encoder.

Although the implementation is relatively simple and it intuitively appears that this approach should increase compression efficiency, a number of different heuristics were tried without achieving noticeable improvements. These heuristics ranged from simply counting the number of times the encoder produced an escape using the reduced order versus the number of times it did not produce an escape, to tracking the fraction of the models which had been used and increasing the order when it reached a threshold, to monitoring the number of times a higher-order model would have succeeded in encoding each symbol versus the

number of times all higher-order models failed. Although certain combinations of parameters produced minimal increases in compression efficiency for certain inputs, the overall results were insignificant.

Further research might reveal a heuristic which would produce acceptable results, and it may be possible through detailed analysis to derive accurate formulas for the order-switching thresholds. For the time being, however, the dynamic order adjustment technique has been abandoned.

Preloading Models Based on Input Type

Another means by which the encoding of such a large number of escape symbols can be avoided is to preload the model with statistics which are estimated to be appropriate for the input data; this technique is also referred to as *priming* the model. Intuitively, if an appropriate initial guess at the model can be made, a great many of the escapes that are produced to build up the higher-order contexts can be avoided. Generalizing the process of analyzing the input data and determining which models to preload is a complex pattern recognition problem and is outside the scope of this thesis; however, in order to evaluate the effect on compression efficiency, the arithmetic coder with scoreboarding was modified to allow it to dump out its model data when it finishes compressing a file and to read this model data again when compressing a new file. The encoder was then used to produce the model information necessary to prime compression for a variety of input files; it was used to compress samples of C source files, relocatable object files, and executable images and to dump the model data when if finished compressing each file. By simply placing the appropriate model data in the dump file before compressing or decompressing, significant improvements in compression efficiency were observed.

Table 13 shows the results obtained for a number of different types of input files; an order of three was used for all runs. It can be seen that both priming and scoreboarding have a more pronounced effect on the compression of small files; they also seem to produce better results on the object and executable files, in which the distribution is more random than in source files. In order to produce the preload files for the C source files, a sample file containing the templates for the standard file and function header comments and a

set of standard C constructs and reserved words was compressed; for the object files, the file *arith.o* was compressed; and for the executable files, the file *huffe* was compressed.

File Name	Size	w/ Neither Ratio	w/ SB Ratio	w/ Preload Ratio	w/ Both Ratio
encode.c	8,334	65%	67%	72%	73%
computil.c	15,227	69%	71%	73%	74%
bit_io.c	23,215	76%	78%	79%	80%
shanfano.c	37171	75%	77%	77%	78%
huffman.c	30503	75%	76%	77%	78%
arith.c	38,348	76%	78%	78%	79%
adap_huf.c	42,025	76%	77%	77%	78%
adap_ari.c	36,207	76%	78%	78%	79%
ahuff_n.c	96,293	80%	80%	80%	81%
aarith_n.c	108,037	79%	80%	80%	81%
encode.o	3,231	41%	46%	53%	57%
bit_io.o	5,148	40%	45%	47%	51%
shanfano.o	6,102	33%	38%	43%	47%
ahuff_n.o	12,973	30%	34%	36%	40%
aarith_n.o	35,334	49%	53%	53%	56%
shane	29,200	46%	49%	71%	71%
aarithne	60,382	47%	50%	58%	60%

Table 13: Compression Efficiency of Arithmetic Coder with Preloading and/or Scoreboarding

The effect of choosing a more appropriate model to preload is shown in Table 14. This is a repeat of the run for C source files, but rather than compressing a sample C test file to produce the model, the file *arith.c* was compressed. Since each of the source files was coded using the same relatively strict coding standard, far greater compression was achieved by using one of them to create the preload model data.

The effect of choosing a model that is a less appropriate match is shown in Table 15. The preload model is the one generated by compressing *arith.c*, but the input files are source code from the software packages *zip* and *kermit*. Although the characteristics of these files are not strictly like those of *arith.c*, some improvement in compression is observed. It appears that more benefit is gained when compressing small files; this seems reasonable, since during the compression of a large file, the model should have time to adapt more properly to the source probabilities and thus be less dependent on the initial state of the model.

File Name	Size	w/ Preload Ratio	w/ Both Ratio
encode.c	8,334	78%	79%
computil.c	15,227	79%	80%
bit_io.c	23,215	82%	83%
shanfano.c	37171	81%	81%
huffman.c	30503	81%	82%
arith.c	38,348	86%	86%
adap_huf.c	42,025	81%	81%
adap_ari.c	36,207	87%	87%
ahuff_n.c	96,293	82%	82%
aarith_n.c	108,037	82%	83%

Table 14: Compression Efficiency of Arithmetic Coder with Preloading of Specific Model Data

File Name	Size	w/o Preload Ratio	w/ Preload Ratio
zip.c	32,880	66%	67%
zipnote.c	9,993	56%	60%
zipsplit.c	17,114	61%	64%
zipup.c	12,085	53%	57%
ckcfn2.c	15,891	61%	63%
ckcfns.c	51,476	62%	63%
ckcmai.c	11,425	56%	57%

Table 15: Compression Efficiency of Arithmetic Coder with Less Specific Preloading

It seems apparent that if a means of fitting an appropriate model to a data file is determined, a significant gain in compression efficiency can be made using this method. However, there are a number of related issues which complicate the use of this technique, one of which is the issue of storing and cataloging the preload models. Also, these models could not be static; for instance, the models which fit the object files on one type of computer might be a very poor match to the object files on another type of computer. However, a compressing archival program could be modified to collect and update statistics on the various types of files on a system in order to generate efficient models for preloading.

CHAPTER 7

CONCLUSIONS

Although the basic statistical compression methods discussed here often compress less efficiently than dictionary-based methods and they tend to have greater memory requirements, they also have greater potential for improvement. The results presented in Chapter 5 indicate that a high-order context modeler coupled with an arithmetic encoder and decoder achieve very good compression efficiency at the expense of execution speed and memory usage. Today's advances in processor and memory technology make these resources more affordable; also, the viability of implementing an arithmetic coder and modeler in hardware further reduces the consideration that must be given to execution speed. The enhancements discussed in Chapter 6 just begin to explore means by which compression efficiency can be improved; the techniques of scoreboarding and preloading have been shown to produce very promising results. With continued research into tuning of the various parameters of the high-order context modelers, the arithmetic coding techniques discussed herein should be capable of outperforming other popular lossless text compression methods.

Comparison with Other Compression Algorithms

Table 16 compares the compression efficiency of the algorithms listed in this thesis to the efficiency of the publicly available *compress*, *PKZIP* v1.1, and *zip* v1.9 programs. All three of these compressors use variations of dictionary-based compression schemes (LZW for *compress* and LZ77 for *PKZIP* and *zip*); the latter two further compress the output of the dictionary-based compressor using either Huffman or Shannon-Fano coding. *zip* currently uses one of the most efficient compression algorithms known, although there are other new programs which outperform it in some tests (i.e., *UltraCompressor II, HAP*); *zip* is probably the most popular compression program available.

Table 17 presents a more detailed comparison of *ahuffne, aarithne, compress, PKZIP*, and *zip*, using higher orders and the scoreboarding technique presented in Chapter 6 for the adaptive statistical compressors. The performance of the order-3 adaptive arithmetic coder with scoreboarding is very similar to the performance of *zip* on text files; it is not as suitable for compression of files with a broader distribution of symbols, such as the object and binary data files. Finally, Table 18 compares *aarithne* using scoreboarding and preloading of a very specific model to *compress* and *zip*. The arithmetic encoder outperforms even *zip* in these tests; obviously, this is a special situation, but it shows the promise of the preloading technique.

The arithmetic coder holds great promise for delivering excellent compression performance; this must be weighed against its resource requirements to determine whether it provides a practical solution to a particular data compression problem. Its advantages over the Huffman coder and other statistical compressors are apparent if the coder is to be combined with a high-order context modeler: its encoding of a source message is proven optimal for a given source model and is provably superior to that of any encoders which assign a fixed-length code to each source symbol; the memory requirements of its probability models are more reasonable than those of models which require a tree or similar complex representation of the probability information; and its representation of the probability information is amenable to alterations which improved compression efficiency. Its main drawback, the computational overhead associated with the required arithmetic operations, is compensated for by the simplicity of its models when it is used in conjunction with a high-order modeler. Overall, it is the best statistical text compression technique available.

Further Research

The programs developed thus far can be used as a framework in which to experiment with various techniques for tuning the parameters of adaptive compression algorithms. Also, similar topics related to dictionary-based compressors can be pursued; for instance, it may be possible to improve the compression efficiency of LZ78-based compressors by preloading or priming their dictionaries. As is evidenced by the volume of traffic on the Internet compression-related newsgroups and the proliferation of articles discussing various compression techniques, the field of data compression is far from being completely explored.

File Name	Size	shane	huffe	arithe	ahuffe	aarithe	aarithee	ahuffne	aarithne	comp.	PKZIP	zip
bib	111,261	35%	35%	35%	35%	35%	35%	56%	57%	58%	63%	69%
book1	768,771	43%	43%	44%	44%	44%	44%	55%	55%	57%	54%	59%
book2	610,856	40%	40%	40%	41%	41%	41%	53%	54%	59%	62%	66%
geo	102,400	29%	29%	29%	29%	30%	30%	34%	36%	24%	26%	33%
news	377,109	35%	35%	36%	35%	36%	36%	48%	49%	52%	58%	62%
obj1	21,504	24%	24%	24%	24%	26%	25%	34%	36%	35%	51%	52%
obj2	246,814	21%	21%	22%	23%	25%	25%	46%	49%	48%	63%	67%
paper1	53,161	37%	38%	38%	38%	38%	38%	52%	53%	53%	62%	65%
paper2	82,199	42%	42%	43%	43%	43%	43%	54%	55%	56%	60%	64%
paper3	46,526	42%	42%	42%	42%	42%	42%	53%	54%	52%	58%	61%
paper4	13,286	41%	41%	41%	41%	40%	41%	51%	51%	48%	57%	59%
paper5	11,954	37%	37%	38%	37%	37%	38%	48%	49%	45%	57%	58%
paper6	38,105	37%	37%	38%	37%	38%	38%	51%	52%	51%	63%	65%
pic	513,224	77%	77%	79%	80%	86%	86%	81%	86%	88%	88%	89%
progc	39,611	34%	35%	35%	35%	35%	35%	51%	52%	51%	64%	67%
progl	71,646	40%	40%	41%	41%	41%	41%	58%	59%	62%	76%	77%
progp	49,379	39%	39%	39%	39%	39%	40%	57%	58%	61%	76%	77%
trans	93,695	31%	31%	31%	31%	32%	32%	55%	57%	59%	75%	80%

Table 16: Comparison of Compression Efficiency of Various Programs

(ahuffne and aarithne both run with orders of one)

File Name	Size	ahuffne n=2	aarithne n=2	aarithne n=3	aarithne n=1, SB	aarithne n=2, SB	aarithne n=3, SB	comp.	PKZIP	zip
bib	111,261	63%	66%	69%	57%	66%	71%	58%	63%	69%
book1	768,771	63%	64%	63%	55%	64%	65%	57%	54%	59%
book2	610,856	61%	64%	62%	54%	64%	64%	59%	62%	66%
geo	102,400	20%	24%	9%	37%	28%	17%	24%	26%	33%
news	377,109	51%	57%	52%	49%	58%	54%	52%	58%	62%
obj1	21,504	28%	35%	32%	37%	38%	35%	35%	51%	52%
obj2	246,814	14%	32%	28%	49%	34%	30%	48%	63%	67%
paper1	53,161	60%	61%	62%	53%	62%	64%	53%	62%	65%
paper2	82,199	62%	63%	64%	55%	64%	66%	56%	60%	64%
paper3	46,526	58%	59%	59%	54%	60%	62%	52%	58%	61%
paper4	13,286	53%	55%	51%	52%	56%	54%	48%	57%	59%
paper5	11,954	51%	52%	48%	49%	54%	51%	45%	57%	58%
paper6	38,105	59%	60%	60%	52%	61%	62%	51%	63%	65%
pic	513,224	80%	86%	85%	86%	86%	86%	88%	88%	89%
progc	39,611	58%	60%	60%	52%	61%	62%	51%	64%	67%
progl	71,646	67%	69%	72%	60%	70%	73%	62%	76%	77%
progp	49,379	67%	69%	71%	58%	70%	73%	61%	76%	77%
trans	93,695	66%	69%	73%	57%	70%	75%	59%	75%	80%

Table 17: Comparison of Compression Efficiency of Variations of Higher-Order Programs

File Name	Size	Neither	SB	Preload	Both	compress	zip
encode.c	8,334	65%	67%	78%	79%	60%	74%
computil.c	15,227	69%	71%	79%	80%	62%	75%
bit_io.c	23,215	76%	78%	82%	83%	68%	82%
shanfano.c	37171	75%	77%	81%	81%	67%	79%
huffman.c	30503	75%	76%	81%	82%	66%	79%
arith.c	38,348	76%	78%	86%	86%	68%	80%
adap_huf.c	42,025	76%	77%	81%	81%	67%	79%
adap_ari.c	36,207	76%	78%	87%	87%	68%	80%
ahuff_n.c	96,293	80%	80%	82%	82%	71%	81%
aarith_n.c	108,037	79%	80%	82%	83%	71%	81%

Table 18: Compression Efficiency of Arithmetic Coder with Preloading and/or Scoreboarding

BIBLIOGRAPHY

[ABRAHAM] Abrahamson, D. M. *An Adaptive Dependency Source Model for Data Compression.* Communications of the ACM, Vol. 32, No. 1, Jan. 1989, pp. 77-83.

[ABRAMSON] Abramson, N. Information Theory and Coding. McGraw-Hill, New York, NY, 1963.

[BELL86] Bell, T. C. *Better OPM/L Text Compression.* IEEE Transactions on Communications, Vol. 14, No. 12, Dec. 1986, pp. 1176-1182.

[BELL87] Bell, T. C. *A Unifying Theory and Improvements for Existing Approaches to Text Compression.* Ph.D. Thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, 1987.

[BELL89] Bell, T. C., and Moffat, A. M. *A Note on the DMC Data Compression Scheme.* The Computer Journal, Vol. 32, No. 1, Feb. 1989, pp. 16-20.

[BELL90] Bell, T. C., Cleary, J. G., and Witten, I. H. Text Compression. Prentice Hall Advanced Reference Series, Englewood Cliffs, NJ, 1990.

[BENTLEY] Bentley, T. L., Sleator, D. D., Tarjan, R.E., and Wei, V. K. *A Locally Adaptive Data Compression Scheme.* Communications of the ACM, Vol. 29, No. 4, April 1986, pp. 320-330.

[BRENT] Brent, R. P. *A Linear Algorithm for Data Compression.* Australian Computer Journal, Vol. 19, No. 2, 1987, pp. 64-68.

[BUNTON] Bunton, S., and Borriello, G. *Practical Dictionary Management for Hardware Data Compression.* Washington Research Foundation Technical Publication #02-90-09, Feb. 1990.

[CLEARY84a] Cleary, J. G., and Witten, I. H. *A Comparison of Enumerative and Adaptive Codes.* IEEE Transactions on Information Theory, Vol. 30, No. 2, March 1984, pp. 306-315.

[CLEARY84b] Cleary, J. G., and Witten, I. H. *Data Compression Using Adaptive Coding and Partial String Matching*, IEEE Transactions on Communications, Vol. 32, No. 4, April 84, pp. 398-402.

[COHN] Cohn, M. *Performance of LZ Compressors with Deferred Innovation.* Technical Report CS-86-127, Computer Science Department, Brandeis University, Waltham, MA, 1986.

[CORMACK84] Cormack, G. V., and Horspool, R. N. *Algorithms for Adaptive Huffman Codes.* Information Processing Letters, Vol. 18, 1984, pp. 159-165.

[CORMACK87] Cormack, G. V., and Horspool, R. N. *Data Compression Using Dynamic Markov Modelling.* The Computer Journal, Vol. 30, No. 6, Dec. 1987, pp. 541-550.

[CORTESI] Cortesi, D. *An Effective Text Compression Algorithm.* Byte, Vol. 7, No. 1, Jan. 1982, pp. 397-403.

[COVER73] Cover, T. M. *Enumerative Source Coding.* IEEE Transactions on Information Theory, Vol. 19, No. 1, Jan. 1973, pp. 73-77.

[COVER91] Cover, T. J. Elements of Information Theory. 1991.

[DANCE] Dance, D.L., and Pooch, A. W. *An Adaptive Online Data Compression System.* The Computer Journal, Vol. 19, No. 3, 1976, pp. 216-224.

[ELIAS75] Elias, P. *Universal Codeword Sets and Representations of the Integers.* IEEE Transactions on Information Theory, Vol. 21, No. 2, March 1975, pp. 194-203.

[ELIAS87] Elias, P. *Interval and Recency Rank Source Coding; Two On-line Adaptive Variable-Length Schemes.* IEEE Transactions on Information Theory, Vol. 33, No. 1, Jan. 1987, pp. 3-10.

[FIALA] Fiala, E. R., and Greene, D. H. *Data Compression with Finite Windows.* Communications of the ACM, Vol. 32, No. 4, April 1989, pp. 490-505.

[GALLAGER] Gallager, R. G. *Variations on a Theme by Huffman.* IEEE Transactions on Information Theory, Vol. 24, No. 6, Nov. 1978, pp. 668-674.

[GONZALEZ] Gonzalez Smith, M. E.., and Storer, J. A. *Parallel Algorithms for Data Compression.* Journal of the ACM, Vol. 32, No. 2, 1985, pp. 344-373.

[GUAZZO] Guazzo, M. *A General Minimum-Redundancy Source-Coding Algorithm.* IEEE Transactions on Information Theory, Vol. 26, No. 1, Jan. 1980, pp. 15-25.

[HAMMING] Hamming, R. W. Coding and Information Theory, 2nd. ed. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[HELD] Held, G. Data Compression: Techniques and Applications, Hardware and Software Considerations. John Wiley and Sons, New York, NY, 1983.

[HORSP84] Horspool, R. N., and Cormack, G. V. *A General-Purpose Data Compression Technique with Practical Applications.* Proceedings of the CIPS, Session 84, Calgary, Canada, pp. 138-141.

[HORSP86] Horspool, R. N., and Cormack, G. V. *Dynamic Markov Modelling - A Prediction Technique.* Proceedings of the 19th Hawaii International Conference on System Sciences. Honolulu, HI, Jan.. 1986, pp. 700-707.

[HORSP91] Horspool, R. N. *Improving LZW.* Proceedings of the Data Compression Conference 1991, James A. Storer and John H. Reif, editors, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 332-341.

[HUFFMAN] Huffman, D. A. *A Method for the Construction of Minimum-Redundancy Codes.* Proceedings of the Institute of Electrical and Radio Engineers, Vol. 40, No. 9, Sept. 1952, pp. 1098-1101.

[JONESC] Jones, C. B. *An Efficient Coding System for Long Source Sequences.* IEEE Transactions on Information Theory, Vol. 27, No. 3, May 1981, pp. 280-291.

[JONESD] Jones, D. W. *Application of Splay Trees to Data Compression.* Communications of the ACM, Vol. 31, No. 8, Aug. 1988, pp. 996-1007.

[KATAJAIN] Katajainen, J., Peuttonen, M., and Teuhola, J. *Syntax-Directed Compression of Source Files.* Software -- Practice and Experience, Vol. 16, No. 3, 1986, pp. 269-276.

[KNUTH73a] Knuth, D. E. The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. Addison Wesley, Reading, MA, 1973.

[KNUTH73b] Knuth, D. E. The Art of Computer Programming, Vol. 3: Searching and Sorting. Addison Wesley, Reading, MA, 1973.

[KNUTH85] Knuth, D. E. *Dynamic Huffman Coding*. Journal of Algorithms, No. 6, 1985, pp. 163-180.

[KWAN] Kwan, R. C. Universal Coding with Different Modelers in Data Compression. Master's Thesis, Department of Computer Science, Montana State University, Bozeman, MT, July, 1987.

[LANGDON81] Langdon, G. G. Jr., and Rissanen, J. J. *Compression of Black-White Images with Arithmetic Coding*. IEEE Transactions on Communications, Vol. 29, No. 6, June 1981, pp. 858-867.

[LANGDON82] Langdon, G. G. Jr., and Jorma Rissanen. *A Simple General Binary Source Code*. IEEE Transactions on Information Theory, Vol. 28, No. 5, Sept. 1982, pp. 800-803.

[LANGDON83a] Langdon, G. G. Jr. *A Note on the Ziv-Lempel Model for Compressing Individual Sequences*. IEEE Transactions on Information Theory, Vol. 29, No. 2, March 1983, pp. 284-287.

[LANGDON83b] Langdon, G. G. Jr., and Rissanen, J. J. *A Doubly-Adaptive File Compression Algorithm*. IEEE Transactions on Communications, Vol. 31, No. 11, Nov. 1983, pp. 1253-1255.

[LANGDON84] Langdon, G. G. Jr. *An Introduction to Arithmetic Coding*. IBM Journal of Research and Development, Vol. 28, No. 2, March 1984, pp. 135-149.

[LELEWER] Lelewer, D. A., and Hirschberg, D. S. *Data Compression*. ACM Computing Surveys, Vol. 19, No. 3, Sept. 1987, pp. 261-296.

[LEMPEL] Lempel, A., and Ziv, J. *Compression of Two-Dimensional Images*. Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, ed., Springer-Verlag, Berlin, 1985, pp. 141-154.

[LLEWELL] Llewellyn, J. A. *Data Compression for a Source with Markov Characteristics*. The Computer Journal, Vol. 30, No. 2, 1986, pp. 149-156.

[LYNCH] Lynch, T. J. Data Compression: Techniques and Applications. Lifetime Publications, Belmont, CA, 1985.

[MCINTYRE] McIntyre, D. R., and Yechura, M. A. *Data Compression Using Static Huffman Code-Decode Tables*. Journal of the ACM, Vol. 28, No. 6, 1985, pp. 612-616.

[MILLER] Miller, V. S., and Wegman, M. N. *Variations on a Theme by Lempel and Ziv*. Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, ed., Springer-Verlag, Berlin, 1985, pp. 131-140.

[MOFFAT87] Moffat, A. M. *Word-Based Text Compression*. Research Report, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia, 1987.

[MOFFAT88a] Moffat, A. M. *A Data Structure for Arithmetic Encoding on Large Alphabets*. Proceedings of the 11th Australian Computer Science Conference, Brisbane, Australia, Feb. 1988, pp. 309-317.

[MOFFAT88b] Moffat, A. M. *A Note on the PPM Data Compression Algorithm.* Research Report 88/7, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia.

[MOFFAT90] Moffat, A. M. *Implementing the PPM Data Compression Scheme.* IEEE Transactions on Communications, Vol. 38, No. 11, Nov. 1990, pp. 1917-1921.

[MORITA] Morita, H., and Kobayashi, K. *On Asymptotic Optimality of a Sliding Window Variation of Lempel-Ziv Codes.* IEEE Transactions on Information Theory, Vol. 39., No. 6, Nov. 1993, pp. 1840-1846.

[MUKHERJEE] Mukherjee, A., and Bassioun, M. A. *On-the-Fly Algorithms for Data Compression.* Proceedings of the ACM/IEEE Fall Joint Computer Conference, 1987.

[NELSON89] Nelson, M. *LZW Data Compression.* Dr. Dobb's Journal, Vol. 14, No. 10, Oct. 1989, pp. 29-37.

[NELSON91] Nelson, M. The Data Compression Book. M & T Books, Redwood City, CA, 1991.

[PASCO] Pasco, R. *Source Coding Algorithm for Fast Data Compression.* Ph.D. Thesis, Department of Electrical Engineering, Stanford University, 1976.

[PERL] Perl, Y., Maram, V., and Kadakuntla, N. *The Cascading of the LZW Compression Algorithm with Arithmetic Coding.* Proceedings of the Data Compression Conference 1991, James A. Storer and John H. Reif, editors, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 277-286.

[PIKE] Pike, J. *Text Compression Using a 4-Bit Coding Scheme.* The Computer Journal, Vol. 24, No. 4, 1981, pp. 324-330.

[RAITA] Raita, J., and Teuhola, J. *Predictive Text Compression by Hashing.* ACM Conference on Information Retrieval, New Orleans, 1987.

[RAMABA] Ramabadran, T. V., and Cohn, D. L. *An Adaptive Algorithm for the Compression of Computer Data.* IEEE Transactions on Communications, Vol. 37, No. 4, April 1989, pp. 317-324.

[RISSANEN76] Rissanen, J. J. *Generalized Kraft Inequality and Arithmetic Coding.* IBM Journal of Research and Development, Vol. 20, 1976, pp. 198-203.

[RISSANEN79] Rissanen, J. J., and Langdon, G. G. Jr. *Arithmetic Coding.* IBM Journal of Research and Development, Vol. 23, No. 2, March 1979, pp. 149-162.

[RISSANEN81] Rissanen, J. J, and Langdon, G. G. Jr. *Universal Modeling and Coding.* IEEE Transactions on Information Theory, Vol. 27., No. 1, Jan. 1981, pp. 12-23.

[RISSANEN83] Rissanen, J. J. *A Universal Data Compression System.* IEEE Transactions on Information Theory, Vol. 29, No. 5, Sept. 1983, pp. 656-664.

[RISSANEN89] Rissanen, J. J., and Mohiuddin, K. M. *A Multiplication-Free Multialphabet Arithmetic Code.* IEEE Transactions on Communications, Vol. 37, No. 2, Feb. 1989, pp. 93-98.

[RISSANEN] Rissanen, J. J. *Optimum Block Models with Fixed-Length Coding.* Technical Report, IBM Research Center, San Jose, CA.

[RODEH] Rodeh, M., Pratt, V. R., and Even, S. *Linear Algorithm for Data Compression via String Matching*. Journal of the ACM, Vol. 28, No. 1, Jan. 1981, pp. 16-24.

[RUBIN76] Rubin, F. *Experiments in Text File Compression*. Communications of the ACM, Vol. 19, No. 11, Nov. 1976, pp. 617-623.

[RUBIN79] Rubin, F. *Arithmetic Stream Coding Using Fixed Precision Registers*. IEEE Transactions on Information Theory, Vol. 25, No. 6, Nov. 1979, pp. 672-675.

[SHANNON] Shannon, C. E., and Weaver, W. The Mathematical Theory of Computation. University of Illinois Press, Urbana, IL, 1949.

[STORER82] Storer, J. A., and Szymanski, T. G. *Data Compression via Textual Substitution*. Journal of the ACM, Vol. 29, No. 4, Oct. 1982, pp. 928-951.

[STORER84] Storer, J. A., and Tsang, S. K. *Data Compression Experiments Using Static and Dynamic Dictionaries*. Technical Report CS-84-118, Computer Science Department, Brandeis University, Waltham, MA, 1984.

[STORER88] Storer, J. A. Data Compression: Methods and Theory. Computer Science Press, Rockville, MD, 1988.

[STORER91a] Storer, J. A., and Reif, J. H., ed. Proceedings of the Data Compression Conference 1991. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[STORER91b] Storer, J. A., and Reif, J. H. *A Parallel Architecture for High-Speed Data Compression*. Journal of Parallel and Distributed Computing, Vol. 13, 1991, pp. 222-227.

[STORER92a] Storer, J. A., and Cohn, M., ed. Proceedings of the Data Compression Conference 1992. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[STORER92b] Storer, J. A., ed. Image and Text Compression. Kluwer Academic Publishers, Norwell, MA, 1992.

[STORER93] Storer, J. A., and Cohn, M., ed. Proceedings of the Data Compression Conference 1993. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[STORER94] Storer, J. A., and Cohn, M., ed. Proceedings of the Data Compression Conference 1994. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[TISCHER] Tischer, D. *A Modified Lempel-Ziv-Welch Data Compression Scheme*. Australian Computer Science Communications, Vol. 4, No. 1, 1987, pp. 262-272.

[VAUGHAN] Vaughan-Nichols, S. J. *Saving Space*. Byte, Vol. 15, No. 3, March 1990, pp. 237-243.

[VITTER87] Vitter, J. S. *Design and Analysis of Dynamic Huffman Codes*. Journal of the ACM, Vol. 34, No. 4, Oct. 1987, pp. 825-845.

[VITTER89] Vitter, J. S. *Dynamic Huffman Coding*. ACM Transactions on Mathematical Software, Vol. 15, No. 2, June 1989, pp. 159-167.

[WALLACE] Wallace, G. K. *The JPEG Still Picture Compression Standard.* Communications of the ACM, Vol. 34, No. 4, April 1991, pp. 30-45.

[WELCH] Welch, T. A. *A Technique for High-Performance Data Compression.* IEEE Computer, Vol. 17, No. 6, June 1984, pp. 8-19.

[WILLIAMS88] Williams, R. N. *Dynamic-History Predictive Compression.* Information Systems, Vol. 13, No. 1, Jan. 1988, pp. 129-140.

[WILLIAMS91a] Williams, R. N. Adaptive Data Compression. Kluwer Academic Publishers, Norwell, MA, 1991.

[WILLIAMS91b] Williams, R. N. *An Extremely Fast Ziv-Lempel Data Compression Algorithm.* Proceedings of the Data Compression Conference 1991, J. A. Storer and J. H. Reif, eds., IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 362-371.

[WILLIAMSb] Williams, R. N. *Predictive Data Compression.* Technical Report, Department of Computer Science, University of Adelaide, Adelaide, Australia.

[WITTEN87] Witten, I. H., Neal, R. M., and Cleary, J. G. *Arithmetic Coding for Data Compression.* Communications of the ACM, Vol. 30, No. 6, June 1987, pp. 520-540.

[WITTEN88] Witten, I. H., and Cleary, J. G. *On the Privacy Afforded by Adaptive Text Compression.* Computers and Security, Vol. 7, No. 4, Aug. 1988, pp. 397-408.

[ZIV77] Ziv, J., and Lempel, A. *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, Vol. 23, No. 3, May 1977, pp. 337-343.

[ZIV78] Ziv, J., and Lempel, A. *Compression of Individual Sequences via Variable Rate Coding.* IEEE Transactions on Information Theory, Vol. 24, No. 5, Sept. 1978, pp. 530-536.