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.

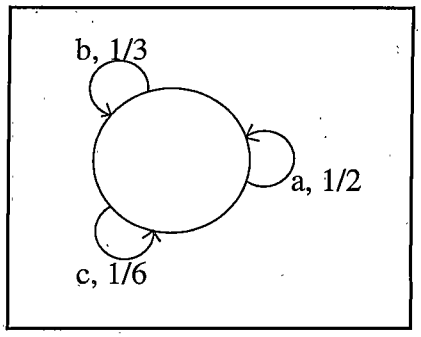

Figure 1: Example of Order-0 Markov Source

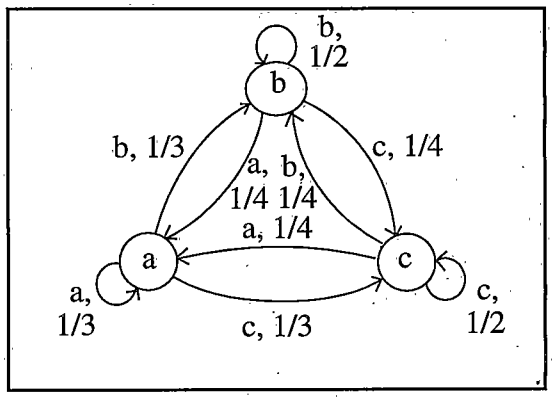

Figure 2: Example of Order-1 Markov Source

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.

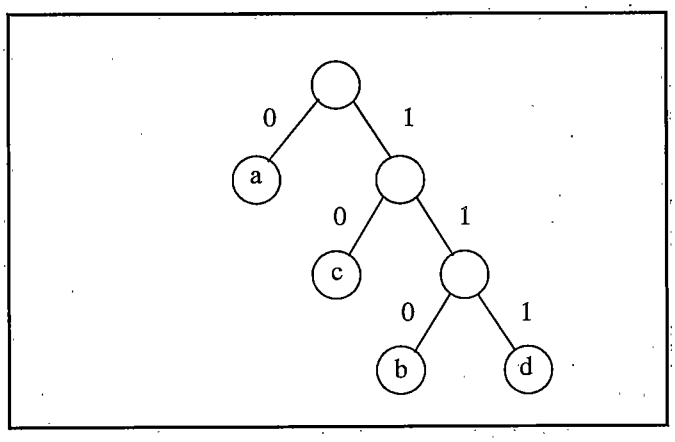

Figure 4: Tree Corresponding to Previous Shannon-Fano Example

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

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

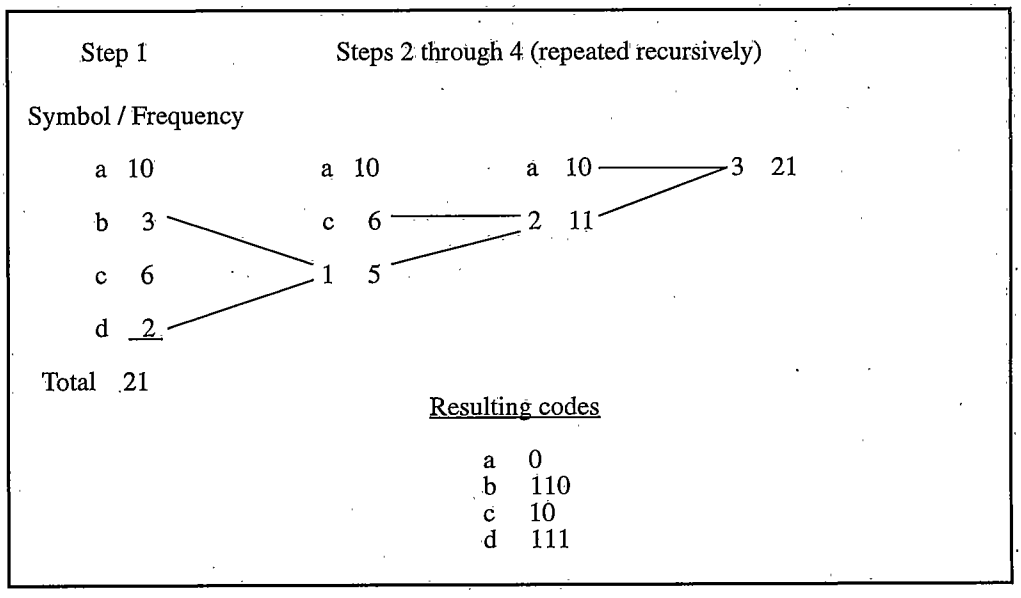

Figure 5: Example of Huffman Code Generation

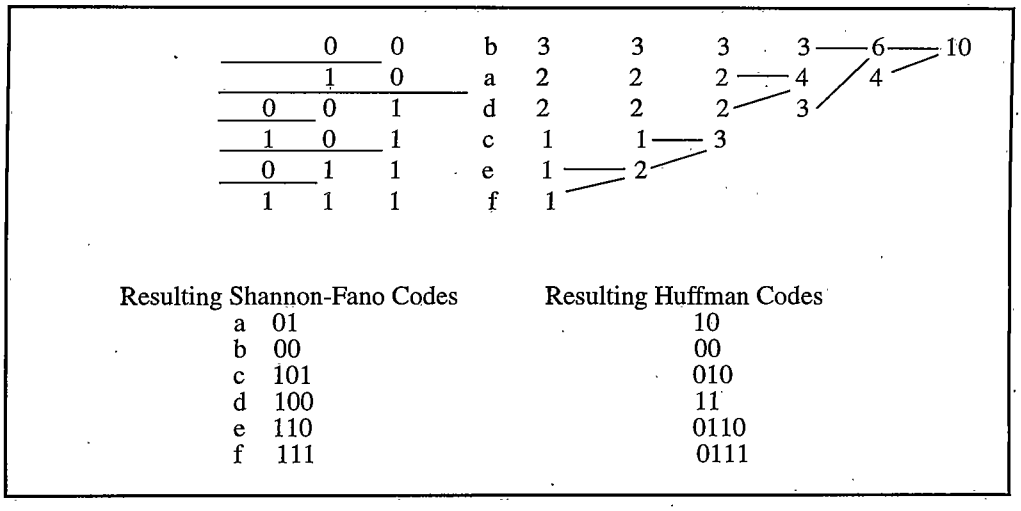

Figure 6: Example Comparing Shannon-Fano and Huffman Codes

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.

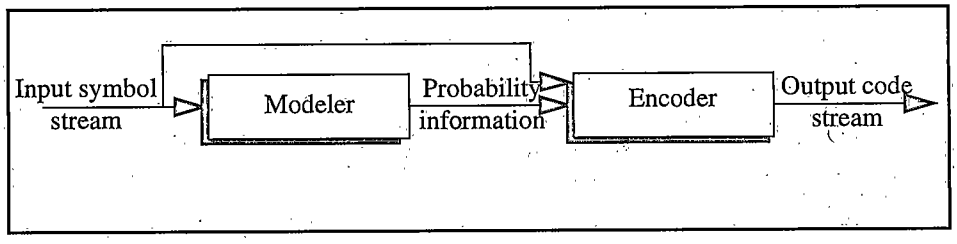

Figure 7: Block Diagram of Compressor Divided into Modelling and Coding Modules

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%.

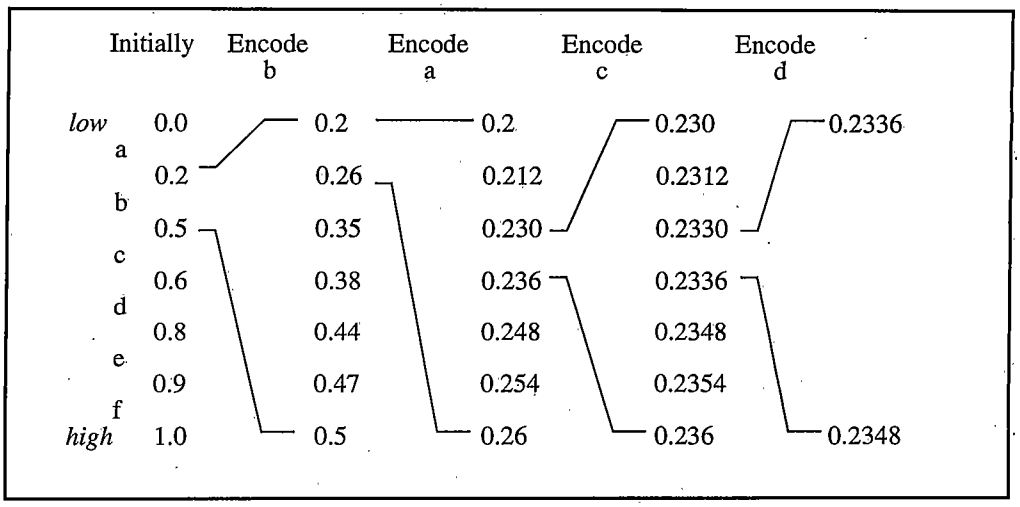

Figure 8: Example of Arithmetic Coding

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.

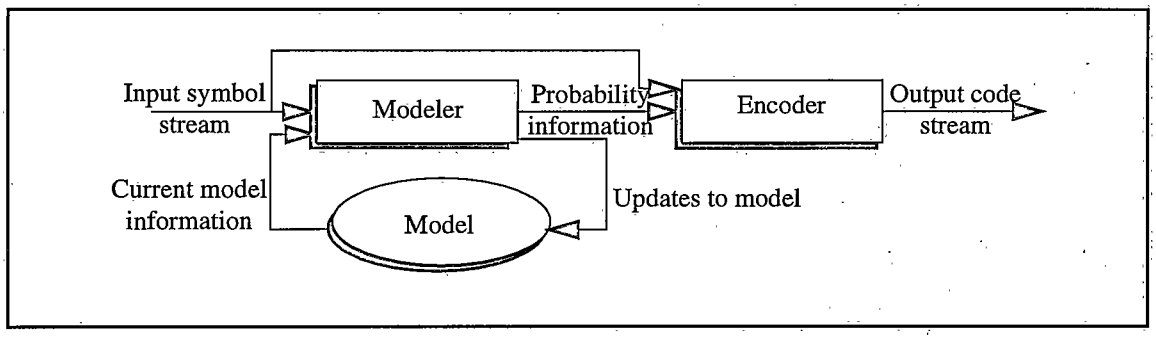

Figure 9: Block Diagram of an Adaptive Compressor

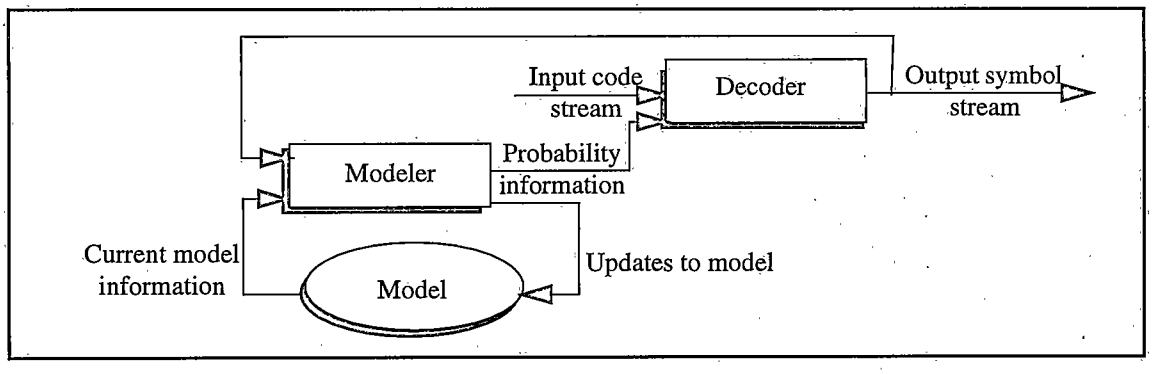

Figure 10: Block Diagram of an Adaptive Decompressor

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.

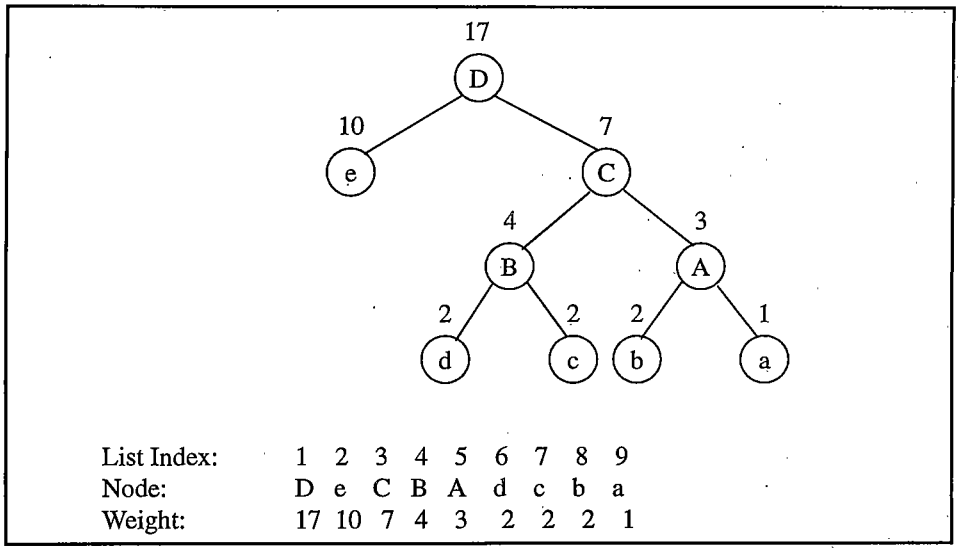

Figure 11: Sample Huffman Tree Exhibiting Sibling Property

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.