

UC Irvine

ICS Technical Reports

Title

Data compression

Permalink

<https://escholarship.org/uc/item/73n9m4nn>

Authors

Lelewer, Debra A.
Hirschberg, Daniel S.

Publication Date

1987

Peer reviewed

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES

Z
699

C3

no. 87-10

Data Compression

Debra A. Lelewer and Daniel S. Hirschberg

Technical Report No. 87-10

Abstract

This paper surveys a variety of data compression methods spanning almost forty years of research, from the work of Shannon, Fano and Huffman in the late 40's to a technique developed in 1986. The aim of data compression is to reduce redundancy in stored or communicated data, thus increasing effective data density. Data compression has important application in the areas of file storage and distributed systems.

Concepts from information theory, as they relate to the goals and evaluation of data compression methods, are discussed briefly. A framework for evaluation and comparison of methods is constructed and applied to the algorithms presented. Comparisons of both theoretical and empirical natures are reported and possibilities for future research are suggested.

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

INTRODUCTION

1. FUNDAMENTAL CONCEPTS

1.1 Definitions

1.2 Classification of Methods

1.3 A Data Compression Model

1.4 Motivation

2. SEMANTIC DEPENDENT METHODS

3. STATIC DEFINED-WORD SCHEMES

3.1 Shannon-Fano Code

3.2 Huffman Coding

3.3 Representations of the Integers

4. ADAPTIVE HUFFMAN CODING

4.1 Algorithm FGK

4.2 Algorithm V

5. OTHER ADAPTIVE METHODS

5.1 Lempel-Ziv Codes

5.2 Algorithm BSTW

6. EMPIRICAL RESULTS

7. SUSCEPTIBILITY TO ERROR

7.1 Static Codes

7.2 Adaptive Codes

8. NEW DIRECTIONS

9. SUMMARY

INTRODUCTION

Data compression is often referred to as coding. Information theory is defined to be the study of efficient coding and its consequences, in the form of speed of transmission and probability of error [Ingels 1971]. Data compression may be viewed as a branch of information theory in which the primary objective is to minimize the amount of data to be transmitted. The purpose of this paper is to present and analyze a variety of data compression algorithms.

While coding for purposes of data security (cryptography) and codes which guarantee a certain level of data integrity (error detection/correction) are topics worthy of attention, these do not fall under the umbrella of data compression. With the exception of a brief discussion of the susceptibility to error of the methods surveyed (Section 7), a discrete noiseless channel is assumed. Of course, the coding schemes described here may be combined with data security or error correcting codes.

Much of the available literature on data compression approaches the topic from the point of view of data transmission. It is important to note that data compression is of value in data storage as well. Although this discussion will be framed in the terminology of data transmission, compression and decompression of data files is essentially the same task as sending and receiving data over a communication channel. The focus of this paper is on algorithms for data compression; it does not deal with hardware aspects of data transmission. The reader is referred to Cappellini for a discussion of techniques with natural hardware implementation [Cappellini 1985].

Background concepts in the form of terminology and a model for the study of data compression are provided in Section 1. Applications of data compression are also discussed in Section 1, to provide motivation for the material which follows.

While the primary focus of this survey is data compression methods of general utility, Section 2 includes examples from the literature in which ingenuity applied to domain-specific problems has yielded interesting coding techniques. These techniques are referred to as *semantic dependent* since they are designed to exploit the context and semantics of the data to achieve redundancy reduction. Semantic dependent techniques include the use of

quadrees, run length encoding, or difference mapping for storage and transmission of image data [Gonzalez and Wintz 1977; Samet 1984].

General-purpose techniques, which assume no knowledge of the information content of the data, are described in Sections 3–5. In most cases, only worst-case analyses of these methods are feasible. To provide a more realistic picture of their effectiveness, empirical data is presented in Section 6. The susceptibility to error of the algorithms surveyed is discussed in Section 7 and possible directions for future research are considered in Section 8.

1. FUNDAMENTAL CONCEPTS

A brief introduction to information theory is provided in this section. The definitions and assumptions necessary to a comprehensive discussion and evaluation of data compression methods are discussed.

1.1 Definitions

A code is a mapping of *source messages* (words from the source alphabet α) into *codewords* (words of the code alphabet β). Codes can be categorized as block-block, block-variable, variable-block or variable-variable, where block-block indicates that the source messages and codewords are of fixed length and variable-variable codes map variable-length source messages into variable-length codewords. The oldest and most widely used codes, ASCII and EBCDIC, are examples of block-block codes. These are not discussed, as they do not provide compression. The codes featured in this survey are of the block-variable, variable-variable, and variable-block types.

When source messages of variable length are allowed, the question of how a message *ensemble* (sequence of messages) is parsed into individual messages arises. Many of the algorithms described here are *defined-word schemes*. That is, the set of source messages is determined prior to the invocation of the coding scheme. For example, in text file processing each character may constitute a message, or messages may be defined to consist of alphanumeric and non-alphanumeric strings. In Pascal source code, each token may represent a message. All codes involving fixed-length source messages are, by default, defined-word codes. In *free-parse* methods, the coding algorithm itself parses the ensemble into variable-length sequences of symbols.

A code is *distinct* if each word of the code is distinguishable from other codewords (i.e., the mapping from source messages to codewords is one-to-one). A distinct code is *uniquely*

decodable if every codeword is identifiable when immersed in a sequence of codewords. A uniquely decodable code is a *prefix code* (or *prefix-free code*) if it has the prefix property, which requires that no codeword is a proper prefix of any other codeword. Prefix codes are *instantaneously decodable*; that is, the coded message can be parsed into codewords without the need for lookahead. A *minimal* prefix code is a prefix code such that if x is a proper prefix of some codeword, then $x\sigma$ is either a codeword or a proper prefix of a codeword, for each letter σ in β . Intuitively, the minimality constraint prevents the use of codewords which are longer than necessary. The codes discussed in this paper are all minimal prefix codes.

1.2 Classification of Methods

In addition to the categorization of data compression schemes with respect to message and codeword lengths, these methods are classified as either static or dynamic. A *static* method is one in which the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the message ensemble. The classic static defined-word scheme is Huffman coding [Huffman 1952]. In Huffman coding, the assignment of codewords to source messages is based on the probabilities with which the source messages appear in the message ensemble. Messages which appear more frequently are represented by short codewords; messages with smaller probabilities map to longer codewords. These probabilities are determined before transmission begins.

A code is *dynamic* if the mapping from the set of messages to the set of codewords changes over time. Dynamic Huffman coding involves computing an approximation to the probabilities of occurrence “on the fly”, as the ensemble is being transmitted. The assignment of codewords to messages is based on the values of the relative frequencies of occurrence at each point in time. A message x may be represented by a short codeword early in the transmission because it occurs frequently at the beginning of the ensemble, even though its probability of occurrence over the total ensemble is low. Later, when the more probable messages begin to occur with higher frequency, the short codeword will be mapped to one of the higher probability messages and x will be mapped to a longer codeword.

Dynamic codes are also referred to in the literature as *adaptive*, in that they adapt to changes in ensemble characteristics over time. The term *adaptive* will be used for the remainder of this paper; the fact that these codes adapt to changing characteristics is the source of their appeal. Some adaptive methods adapt to changing patterns in the source [Welch 1984] while others exploit locality of reference [Bentley et al. 1986]. Locality of

reference is the tendency, common in a wide variety of text types, for a particular word to occur frequently for short periods of time then fall into disuse for long periods. All of the adaptive methods are one-pass methods; only one scan of the ensemble is required. Static Huffman coding requires two passes: one pass to compute probabilities and determine the mapping, and a second pass for transmission. The one-pass methods define and redefine the mapping dynamically, during transmission.

An algorithm may also be a *hybrid*, neither completely static nor completely dynamic. In a simple hybrid scheme, sender and receiver maintain identical codebooks containing k static codes. For each transmission, the sender must choose one of the k codes and inform the receiver of his choice. Hybrid methods are discussed further in Section 2 and Section 3.2.

1.3 A Data Compression Model

In order to discuss the relative merits of data compression techniques, a framework for comparison must be established. There are two dimensions along which each of the schemes discussed here may be measured, algorithm complexity and amount of compression. For the static schemes, there are three algorithms to analyze: the map construction algorithm, the encoding algorithm, and the decoding algorithm. The degree of compression is usually the primary concern in data compression, but it is necessary to ascertain that the time requirements of the algorithm are not prohibitive.

Several common measures of compression have been suggested: redundancy [Shannon and Weaver 1949], average message length [Huffman 1952], and compression ratio [Rubin 1976; Ruth and Kreutzer 1972]. These measures are defined below. Related to each of these measures are assumptions about the characteristics of the source. It is generally assumed in information theory that all statistical parameters of a message source are known with perfect accuracy [Gilbert 1971]. The most common model is that of a discrete memoryless source; a source whose output is a sequence of letters (or messages), each letter being a selection from some fixed alphabet a_1, \dots, a_n . The letters are taken to be random statistically independent selections from the alphabet, the selection being made according to some fixed probability assignment $p(a_1), \dots, p(a_n)$ [Gallager 1968]. Without loss of generality, the code alphabet is assumed to be $\{0, 1\}$ throughout this paper. The modifications necessary for larger code alphabets are straightforward.

It is assumed that any cost associated with the code letters is uniform. This assumption is reasonable, and important. The problem of constructing optimal codes over unequal letter costs is a significantly more difficult problem. Varn and Perl have developed algorithms for

minimum-redundancy prefix coding in the case of arbitrary symbol cost and equal codeword probability [Varn 1971; Perl et al. 1975]. The assumption of equal probabilities mitigates the difficulty presented by the variable symbol cost. For the more general unequal letter costs and unequal probabilities model, Karp has proposed an integer linear programming approach [Karp 1961]. There have been several approximation algorithms proposed for this more difficult problem [Krause 1962; Cot 1977; Mehlhorn 1980].

When data is compressed, the goal is to reduce redundancy, leaving only the informational content. The measure of information of a source message a_i (in bits) is $-\lg p(a_i)$ †. This definition has intuitive appeal; in the case that $p(a_i) = 1$, it is clear that a_i is not at all informative since it had to occur. Similarly, the smaller the value of $p(a_i)$, the more unlikely a_i is to appear, hence the larger its information content. The reader is referred to Abramson for a longer, more elegant discussion of the legitimacy of this technical definition of the concept of information [Abramson 1963, pp. 6–13]. The average information content over the source alphabet is a particularly important quantity known as the entropy of a source letter, and is given by $H = \sum_{i=1}^n -p(a_i) \lg p(a_i)$. Entropy defines a lower bound on the number of bits required for the coded message; the length of a codeword for message a_i must be sufficient to carry the information content of a_i . Since the value of H is generally not an integer, variable length codewords must be used if the lower bound is to be achieved. Both of these definitions of information content are due to Shannon. A derivation of the concept of entropy as it relates to information theory is presented by Shannon [Shannon and Weaver 1949].

The most common notion of a “good” code is one which is *optimal* in the sense of having minimum redundancy. *Redundancy* can be defined as: $\sum p(a_i)l_i - \sum [-p(a_i) \lg p(a_i)]$ where l_i is the length of the codeword representing message a_i . That is, redundancy is a measure of the difference between expected codeword length and average information content. If a code has minimum average length for a given discrete probability distribution, it is said to be a minimum redundancy code. Since all of the methods to be discussed here are uniquely decodable, the definition of optimality includes implicitly the requirement of unique decodability.

We define the term *local redundancy* to capture the notion of redundancy caused by local properties of a message ensemble, rather than its global characteristics. While the model used for analyzing general-purpose coding techniques assumes a random distribution of the source messages, this may not actually be the case. In particular applications the tendency

† \lg denotes the base 2 logarithm

for messages to cluster in predictable patterns may be known. The existence of predictable patterns may be exploited to minimize local redundancy. Examples of applications in which local redundancy is common, and methods for dealing with local redundancy, are discussed in Section 2 and Section 6.2.

Huffman uses *average message length*, $\sum p(a_i)l_i$, as a measure of the efficiency of a code. Clearly the meaning of this term is the average length of a *coded* message. We will use the term *average codeword length* to represent this quantity. Since entropy is constant for a given probability distribution, minimizing average codeword length minimizes redundancy.

A code is *asymptotically optimal* if it has the property that for a given probability distribution the ratio of average codeword length to entropy approaches 1 as entropy tends to infinity.

The amount of compression yielded by a coding scheme can be measured by a *compression ratio*. The term compression ratio has been defined in several ways. The definition $C = (\text{average message length})/(\text{average codeword length})$ captures the common meaning [Cappellini 1985]. A somewhat different definition by Rubin, $C = (S - O - OR)/S$, includes the representation of the code itself in the transmission cost [Rubin 1976]. In this definition S represents the length of the source ensemble, O the length of the output (coded message), and OR the size of the output (code) representation. The quantity OR constitutes a "charge" to an algorithm for transmission of information about the coding scheme.

1.4 Motivation

Data compression has wide application in terms of information storage, including representation of the abstract data type *string* [Standish 1980] and file compression. Huffman coding is used for compression in the ARC file archival system [ARC 1986], as is Lempel-Ziv coding, one of the adaptive schemes to be discussed in Section 5. An adaptive Huffman coding technique is the basis for the *compact* command of the UNIX operating system [UNIX 1984].

In the area of data transmission, Huffman coding has been passed over for years in favor of block-block codes, notably ASCII. The advantage of Huffman coding is in the average number of bits per character transmitted, which may be much smaller than the $\lg n$ bits per character (where n is the source alphabet size) of a block-block system. The primary difficulty associated with variable-length codewords is that the rate at which bits are presented to the transmission channel will fluctuate, depending on the relative frequencies of

the source messages. This requires buffering between the source and the channel. Advances in technology have both overcome this difficulty and contributed to the appeal of variable-length codes. Current data networks allocate communication resources to sources on the basis of need and provide buffering as part of the system. These systems require significant amounts of protocol, and fixed-length codes are quite inefficient for applications such as packet headers. In addition, communication costs are beginning to dominate storage and processing cost, so that variable-length coding schemes which reduce communication costs are attractive even if they are more complex. For these reasons, one could expect to see even greater use of variable-length coding in the future.

It is interesting to note that the Huffman coding algorithm has a wide variety of applications outside the sphere of data compression. These include construction of optimal search trees [Zimmerman 1959; Hu and Tucker 1971; Itai 1976], list merging [Brent and Kung 1978], and generating optimal evaluation trees in the compilation of expressions [Parker 1980]. Additional applications involve search for jumps in a monotone function of a single variable, sources of pollution along a river, and leaks in a pipeline [Glassey and Karp 1976].

2. SEMANTIC DEPENDENT METHODS

Semantic dependent data compression techniques are designed to respond to specific types of local redundancy occurring in certain applications. One area in which data compression is of great importance is image representation and processing. There are two major reasons for this. The first is that digitized images contain a large amount of local redundancy. An image is usually captured in the form of an array of pixels whereas methods which exploit the tendency for pixels of like color or intensity to cluster together may be more efficient. The second reason for the abundance of research in this area is volume. Digital images usually require a very large number of bits, and many uses of digital images involve large collections of images.

One technique used for compression of image data is *run length encoding*. In a common version of run length encoding, the sequence of image elements along a scan line (row) x_1, x_2, \dots, x_n is mapped into a sequence of pairs $(c_1, l_1), (c_2, l_2), \dots, (c_k, l_k)$ where c_i represents an intensity or color and l_i the length of the i^{th} run (sequence of pixels of equal intensity). For pictures such as weather maps, run length encoding can save a significant number of bits over the image element sequence [Gonzalez and Wintz 1977]. Another data compression technique specific to the area of image data is *difference mapping*, in which the image is represented as an array of differences in brightness (or color) between adjacent pixels rather

than the brightness values themselves. Difference mapping was used to encode the pictures of Uranus transmitted by *Voyager 2*. The 8 bits per pixel needed to represent 256 brightness levels was reduced to an average of 3 bits per pixel when difference values were transmitted [Laeser et al. 1986]. In spacecraft applications, image fidelity is a major concern due to the effect of the distance from the spacecraft to earth on transmission reliability. Difference mapping was combined with error-correcting codes to provide both compression and data integrity in the *Voyager* project. Another method which takes advantage of the tendency for images to contain large areas of constant intensity is the use of the quadtree data structure [Samet 1984]. Additional examples of coding techniques used in image processing can be found in Wilkins and Cappellini [Wilkins and Wintz 1971; Cappellini 1985].

Data compression is of interest in business data processing, both because of the cost savings it offers and because of the large volume of data manipulated in many business applications. The types of local redundancy present in business data files include runs of zeros in numeric fields, sequences of blanks in alphanumeric fields, and fields which are present in some records and null in others. Run length encoding can be used to compress sequences of zeros or blanks. Null suppression may be accomplished through the use of presence bits [Ruth and Kreutzer 1972]. Another class of methods exploits cases in which only a limited set of attribute values exist. *Dictionary substitution* entails replacing alphanumeric representations of information such as bank account type, insurance policy type, sex, month, etc. by the few bits necessary to represent the limited number of possible attribute values [Reghbati 1981].

Cormack describes a data compression system which is designed for use with database files [Cormack 1985]. The method, which is part of IBM's "Information Management System" (IMS), compresses individual records and is invoked each time a record is stored in the database file; expansion is performed each time a record is retrieved. Since records may be retrieved in any order, context information used by the compression routine is limited to a single record. In order for the routine to be applicable to any database, it must be able to adapt to the format of the record. The fact that database records are usually heterogeneous collections of small fields indicates that the local properties of the data are more important than its global characteristics. The compression routine in IMS is a hybrid method which attacks this local redundancy by using different coding schemes for different types of fields. The identified field types in IMS are *letters of the alphabet*, *numeric digits*, *packed decimal digit pairs*, *blank*, and *other*. When compression begins, a default code is used to encode the first character of the record. For each subsequent character, the type of the previous character determines the code to be used. For example, if the record "01870bABCDbbLMN" were encoded with the *letter* code as default, the leading zero would be coded using the *letter*

code; the 1, 8, 7, 0 and the first blank (*b*) would be coded by the *numeric* code. The *A* would be coded by the *blank* code; *B, C, D*, and the next blank by the *letter* code; the next blank and the *L* by the *blank* code; and the *M* and *N* by the *letter* code. Clearly, each code must define a codeword for every character; the *letter* code would assign the shortest codewords to letters, the numeric code would favor the digits, etc. In the system Cormack describes, the types of the characters are stored in the encode/decode data structures. When a character *c* is received, the decoder checks *type(c)* to detect which code table will be used in transmitting the next character. The compression algorithm might be more efficient if a special bit string were used to alert the receiver to a change in code table. Particularly if fields were reasonably long, decoding would be more rapid and the extra bits in the transmission would not be excessive. Cormack reports that the performance of the IMS compression routines is very good; at least fifty sites are currently using the system.

A variety of approaches to data compression designed with text files in mind include use of a dictionary either representing all of the words in the file so that the file itself is coded as a list of pointers to the dictionary [Hahn 1974], or representing common words and word endings so that the file consists of pointers to the dictionary and encodings of the less common words [Tropper 1982]. Hand-selection of common phrases [Wagner 1973], programmed selection of prefixes and suffixes [Fraenkel et al. 1983] and programmed selection of common character pairs [Snyderman and Hunt 1970; Cortesi 1982] have also been investigated.

This discussion of semantic dependent data compression techniques represents a limited sample of a very large body of research. These methods and others of a like nature are interesting and of great value in their intended domains. Their obvious drawback lies in their limited utility. It should be noted, however, that much of the efficiency gained through the use of semantic dependent techniques can be achieved through more general methods, albeit to a lesser degree. For example, the dictionary approaches can be implemented through either Huffman coding (Section 3.2, Section 4) or Lempel-Ziv codes (Section 5.1). Cormack's database scheme is a special case of the codebook approach (Section 3.2), and run length encoding is one of the effects of Lempel-Ziv codes.

3. STATIC DEFINED-WORD SCHEMES

The classic defined-word scheme was developed over 30 years ago in Huffman's well-known paper on minimum-redundancy coding [Huffman 1952]. Huffman's algorithm provided the first solution to the problem of constructing minimum-redundancy codes. An earlier algorithm, due independently to Shannon and Fano [Shannon and Weaver 1949; Fano

1949], is not guaranteed to provide optimal codes, but approaches optimal behavior as the number of messages approaches infinity. The Huffman algorithm is also of importance because it has provided a foundation upon which other data compression techniques, and applications in other areas, have built. We classify the codes generated by the Huffman and Shannon-Fano algorithms as variable-variable and note that they include block-variable codes as a special case, depending upon how the source messages are defined.

3.1 Shannon-Fano Coding

The Shannon-Fano technique has as an advantage its simplicity. The code is constructed as follows: the source messages a_i and their probabilities $p(a_i)$ are listed in order of non-increasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have codewords beginning with 1. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one message. Clearly the Shannon-Fano algorithm yields a minimal prefix code.

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

Figure 3.1 A Shannon-Fano Code.

Figure 3.1 shows the application of the method to a particularly simple probability distribution. The length of each codeword is equal to $-\lg p(a_i)$. This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some codewords may be of length $-\lg p(a_i) + 1$. The Shannon-Fano algorithm yields an average codeword length S which satisfies $H \leq S \leq H + 1$. That the Shannon-Fano algorithm is not guaranteed to produce an optimal code is demonstrated by the following set of probabilities: $\{.35, .17, .17, .16, .15\}$. The Shannon-Fano code for this distribution is compared with the Huffman code in Section 3.2.

3.2. Static Huffman Coding

Huffman's algorithm, expressed graphically, takes as input a list of nonnegative weights $\{w_1, \dots, w_n\}$ and constructs a full binary tree † whose leaves are labeled with the weights. When the Huffman algorithm is used to construct a code, the weights represent the probabilities associated with the source letters. Initially there is a set of singleton trees, one for each weight in the list. At each step in the algorithm the trees corresponding to the two smallest weights, w_i and w_j , are merged into a new tree whose weight is $w_i + w_j$ and whose root has two children which are the subtrees represented by w_i and w_j . The weights w_i and w_j are removed from the list and $w_i + w_j$ is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen. In Huffman's paper, the process begins with a nonincreasing list of weights. This detail is not important to the correctness of the algorithm, but it does provide a more efficient implementation [Huffman 1952]. The Huffman algorithm is demonstrated in Figure 3.2.

The Huffman algorithm determines the lengths of the codewords to be mapped to each of the source letters a_i . There are many alternatives for specifying the actual digits; it is necessary only that the code have the prefix property. The usual assignment entails labeling the edge from each parent to its left child with the digit 0 and the edge to the right child with 1. The codeword for each source letter is the sequence of labels along the path from the root to the leaf node representing that letter. The codewords for the source of Figure 3.2, in order of decreasing probability, are $\{01, 11, 001, 100, 101, 0000, 0001\}$. Clearly, this process yields a minimal prefix code. Further, the algorithm is guaranteed to produce an *optimal* (minimum redundancy) code [Huffman 1952]. Gallager has proved an upper bound on the redundancy of a Huffman code of $p_n + \lg[(2 \lg e)/e] \approx p_n + 0.086$, where p_n is the probability of the least likely source message [Gallager 1978]. Figure 3.3 shows a distribution for which the Huffman code is optimal while the Shannon-Fano code is not.

In addition to the fact that there are many ways of forming codewords of appropriate lengths, there are cases in which the Huffman algorithm does not uniquely determine these lengths due to the arbitrary choice among equal minimum weights. As an example, codes with codeword lengths of $\{1, 2, 3, 4, 4\}$ and of $\{2, 2, 2, 3, 3\}$ both yield the same average codeword length for a source with probabilities $\{.4, .2, .2, .1, .1\}$. Schwartz defined a variation of the Huffman algorithm which performs "bottom merging"; that is, orders a new parent node above existing nodes of the same weight and always merges the last two weights in the

† a binary tree is full if every node has either zero or two children

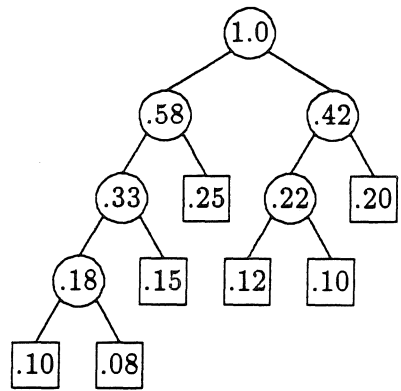
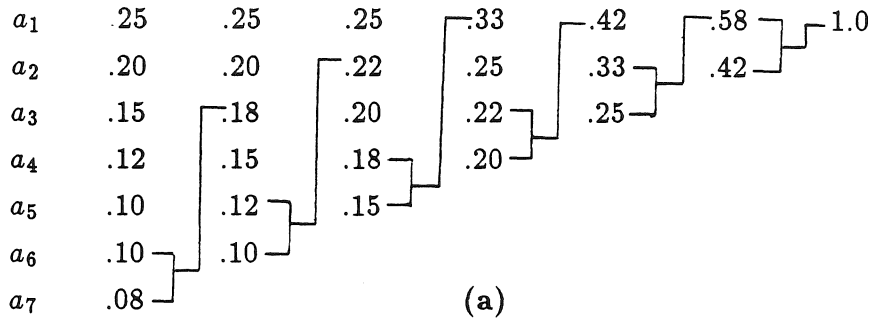


Figure 3.2 The Huffman process. a) the list. b) the tree

list. The code constructed is the Huffman code with minimum values of maximum codeword length ($\max\{l_i\}$) and total codeword length ($\sum l_i$) [Schwartz 1964]. Schwartz and Kallick describe an implementation of Huffman's algorithm with bottom merging [Schwartz and Kallick 1964]. The Schwartz-Kallick algorithm and a later algorithm by Connell [Connell 1973] use Huffman's procedure to determine the lengths of the codewords, and actual digits are assigned so that the code has the *numerical sequence property*. That is, codewords of equal length form a consecutive sequence of binary numbers. Shannon-Fano codes also have the numerical sequence property. This property can be exploited to achieve a compact representation of the code and rapid encoding and decoding.

Both the Huffman and the Shannon-Fano mappings can be generated in $O(n)$ time, where n is the number of messages in the source ensemble (assuming that the weights have been presorted). Each of these algorithms maps a source message a_i with probability p to a codeword of length l ($-\lg p \leq l \leq -\lg p + 1$). Encoding and decoding times depend upon

		S-F	Huffman
a_1	0.35	00	1
a_2	0.17	01	011
a_3	0.17	10	010
a_4	0.16	110	001
a_5	0.15	111	000
avg. code length		2.31	2.30

Figure 3.3 Comparison of Shannon-Fano and Huffman Codes.

the representation of the mapping. If the mapping is stored as a binary tree, then decoding the codeword for a_i involves following a path of length l in the tree. A table indexed by the source messages could be used for encoding; the code for a_i would be stored in position i of the table and encoding time would be $O(l)$. Connell's algorithm makes use of the *index* of the Huffman code, a representation of the distribution of codeword lengths, to encode and decode in $O(c)$ time where c is the number of different codeword lengths.

As noted earlier, the redundancy bound for Shannon-Fano codes is 1 and the bound for the Huffman method is $p_n + 0.086$ where p_n is the probability of the least likely source message (so p_n is less than or equal to .5, and generally much less). It is important to note that in defining redundancy to be average codeword length minus entropy, the cost of transmitting the code mapping computed by these algorithms is ignored. The overhead cost for any method where the source alphabet has not been established prior to transmission includes $n \lg n$ bits for sending the n source letters. For a Shannon-Fano code, a list of codewords ordered so as to correspond to the source letters could be transmitted. The additional time required is then $\sum l_i$, where the l_i are the lengths of the codewords. For Huffman coding, an encoding of the shape of the code tree might be transmitted. Since any full binary tree may be a legal Huffman code tree, encoding tree shape may require as many as $\lg 4^n = 2n$ bits. In most cases the message ensemble is very large, so that the number of bits of overhead is minute by comparison to the total length of the encoded transmission. However, it is imprudent to ignore this cost.

If a less-than-optimal code is acceptable, the overhead costs can be avoided through a prior agreement by sender and receiver as to the code mapping. Rather than using a Huffman code based upon the characteristics of the current message ensemble, the code

used could be based on statistics for a class of transmissions to which the current ensemble is assumed to belong. That is, both sender and receiver could have access to a codebook with k mappings in it; one for Pascal source, one for English text, etc. The sender would then simply alert the receiver as to which of the common codes he is using. This requires only $\lg k$ bits of overhead. Assuming that classes of transmission with relatively stable characteristics could be identified, this hybrid approach would greatly reduce the redundancy due to overhead without significantly increasing expected codeword length. In addition, the cost of computing the mapping would be amortized over all files of a given class. That is, the mapping would be computed once on a statistically significant sample and then used on a great number of files for which the sample is representative. There is clearly a substantial risk associated with assumptions about file characteristics and great care would be necessary in choosing both the sample from which the mapping is to be derived and the categories into which to partition transmissions. An extreme example of the risk associated with the codebook approach is provided by author Ernest V. Wright who wrote a novel *Gadsby* (1939) containing no occurrences of the letter E. Since E is the most commonly used letter in the English language, an encoding based upon a sample from *Gadsby* would be disastrous if used with "normal" examples of English text. Similarly, the "normal" encoding would provide poor compression of *Gadsby*.

McIntyre and Pechura describe an experiment in which the codebook approach is compared to static Huffman coding [McIntyre and Pechura 1985]. The sample used for comparison is a collection of 530 source programs in four languages. The codebook contains a Pascal code tree, a FORTRAN code tree, a COBOL code tree, a PL/1 code tree, and an ALL code tree. The Pascal code tree is the result of applying the static Huffman algorithm to the combined character frequencies of all of the Pascal programs in the sample. The ALL code tree is based upon the combined character frequencies for all of the programs. The experiment involves encoding each of the programs using the five codes in the codebook and the static Huffman algorithm. The data reported for each of the 530 programs consists of the size of the coded program for each of the five predetermined codes, and the size of the coded program plus the size of the mapping (in table form) for the static Huffman method. In every case, the code tree for the language class to which the program belongs generates the most compact encoding. Although using the Huffman algorithm on the program itself yields an optimal mapping, the overhead cost is greater than the added redundancy incurred by the less-than-optimal code. In many cases, the ALL code tree also generates a more compact encoding than the static Huffman algorithm. In the worst case, an encoding constructed from the codebook is only 6.6% larger than that constructed by the Huffman

algorithm. These results suggest that, for files of source code, the codebook approach may be appropriate.

Gilbert discusses the construction of Huffman codes based on inaccurate source probabilities [Gilbert 1971]. A simple solution to the problem of incomplete knowledge of the source is to avoid long codewords, thereby minimizing the error of underestimating badly the probability of a message. The problem becomes one of constructing the optimal binary tree subject to a height restriction (see [Knuth 1971; Hu and Tan 1972; Garey 1974]). Another approach involves collecting statistics for several sources and then constructing a code based upon some combined criterion. This approach could be applied to the problem of designing a single code for use with English, French, German, etc., sources. To accomplish this, Huffman's algorithm could be used to minimize either the average codeword length for the combined source probabilities; or the average codeword length for English, subject to constraints on average codeword lengths for the other sources.

3.3 Representations of the Integers

Elias defines a sequence of coding schemes which map the positive integers onto the set of binary codewords [Elias 1975]. These schemes are *universal* in the following sense: given any countable set of messages and any probability distribution, mapping messages in order of decreasing probability to codewords in order of increasing length gives an average codeword length that is bounded by $c_1H + c_2$. Due to the restriction of the set of source messages to the integers, this method is not a general-purpose coding technique. However, an application of the Elias codes to the general data compression problem is demonstrated in Section 5.

The first Elias code is one which is simple but not optimal. This code, γ , maps an integer x onto the binary value of x prefaced by $\lfloor \lg x \rfloor$ zeros. The binary value of x begins with a 1, which serves to delimit the prefix. The result is a prefix code since the total length of the codeword is exactly one greater than twice the number of zeros in the prefix. The code is not a minimum redundancy code since the ratio of expected codeword length to entropy goes to 2 as entropy approaches infinity. The second code, δ , maps an integer x to a codeword consisting of $\gamma(\lfloor \lg x \rfloor + 1)$ followed by the binary value of x with the leading 1 deleted. The resulting codeword has length $\lfloor \lg x \rfloor + 2\lfloor \lg(1 + \lfloor \lg x \rfloor) \rfloor + 1$. This concept can be applied recursively to shorten the codeword lengths, but the benefits decrease rapidly. The code δ is asymptotically optimal since the limit of the ratio of expected codeword length to entropy is 1. Figure 3.4 lists the values of γ and δ for a sampling of the integers.

γ	δ
1 1	1
2 010	0100
3 011	0101
4 00100	01100
5 00101	01101
6 00110	01110
7 00111	01111
8 0001000	00100000
16 000010000	001010000
17 000010001	001010001
32 00000100000	0011000000

Figure 3.4 Elias Codes.

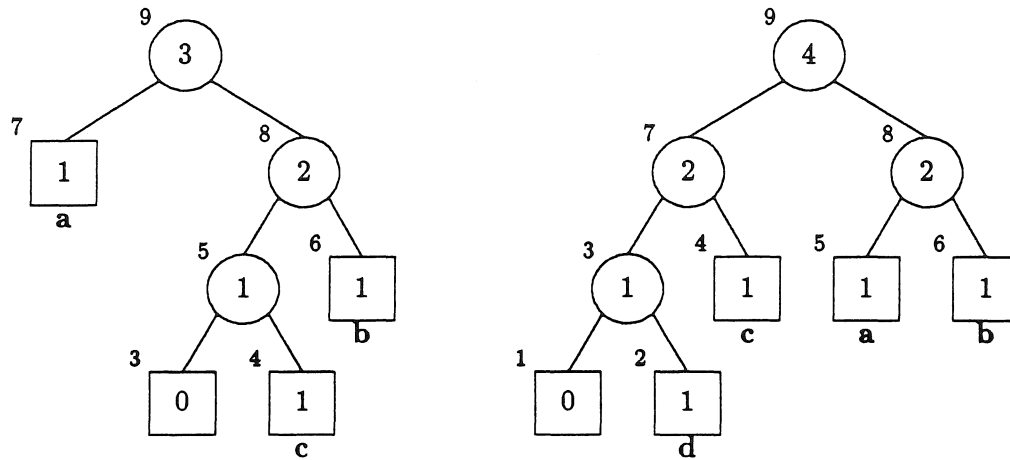
4. ADAPTIVE HUFFMAN CODING

Adaptive Huffman coding was first conceived independently by Faller and Gallager [Faller 1973; Gallager 1978]. Knuth contributed improvements to the original algorithm [Knuth 1985] and the resulting algorithm is referred to as algorithm FGK. A more recent version of adaptive Huffman coding is described by Vitter [Vitter 1985]. All of these methods are defined-word schemes which determine the mapping from source messages to codewords based upon a running estimate of the source message probabilities. The code is adaptive, changing so as to remain optimal for the current estimates. The algorithms require only one pass over the data. The performance of these methods, in terms of number of bits transmitted, can be better than that of static Huffman coding. This does not contradict the optimality of the static method as the static method is optimal only over all methods which assume a time-invariant mapping. The performance of the adaptive methods can also be worse than that of the static method. Upper bounds on the redundancy of these methods are discussed in this section.

4.1 Algorithm FGK

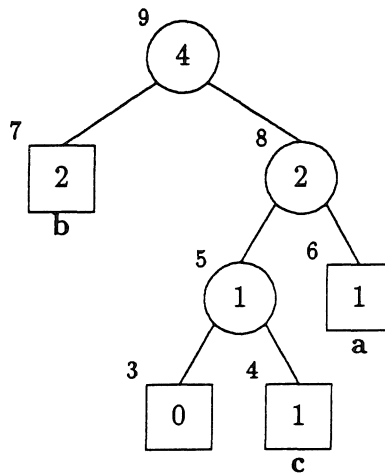
The basis for algorithm FGK is the Sibling Property, defined by Gallager [Gallager 1978]: A binary code tree has the sibling property if each node (except the root) has a sibling and if the nodes can be listed in order of nonincreasing weight with each node adjacent to its

sibling. Gallager proves that a binary prefix code is a Huffman code if and only if the code tree has the sibling property. In algorithm FGK, both sender and receiver maintain dynamically changing Huffman code trees. The leaves of the code tree represent the source messages and the weights of the leaves represent frequency counts for the messages. At any point in time, k of the n possible source messages have occurred in the message ensemble.



(a)

(b)



(c)

Figure 4.1 Algorithm FGK processing the ensemble “abcd”. a) Tree immediately before processing d ; 100 will be transmitted. b) After encoding of d . c) If the 4th letter had been b , 11 would have been transmitted; tree after update is shown.

Initially, the code tree consists of a single leaf node, called the 0-node. The 0-node is a special node used to represent the $n - k$ unused messages. For each message transmitted, both parties must increment the corresponding weight and recompute the code tree to maintain the sibling property. At the point in time when t messages have been transmitted, k of them distinct, and $k < n$, the tree is a legal Huffman code tree with $k + 1$ leaves, one for each of the k messages and one for the 0-node. If the $(t + 1)^{st}$ message is one of the k already seen, the algorithm transmits a_{t+1} 's current code, increments the appropriate counter and recomputes the tree. If an unused message occurs, the 0-node is split to create a pair of leaves, one for a_{t+1} , and a sibling which is the new 0-node. Again the tree is recomputed. In this case, the code for the 0-node is sent; in addition, the receiver must be told which of the $n - k$ unused messages has appeared. In Figure 4.1, a simple example is given. At each node a count of occurrences of the corresponding message is stored. Nodes are numbered indicating their position in the sibling property ordering. The updating of the tree can be done in a single traversal from the a_{t+1} node to the root. This traversal must increment the count for the a_{t+1} node and for each of its ancestors. Nodes may be exchanged to maintain the sibling property, but all of these exchanges involve a node on the path from a_{t+1} to the root. Figure 4.2 illustrates the tree formed by this process on the message ensemble "eteaetneteareteaetnoi".

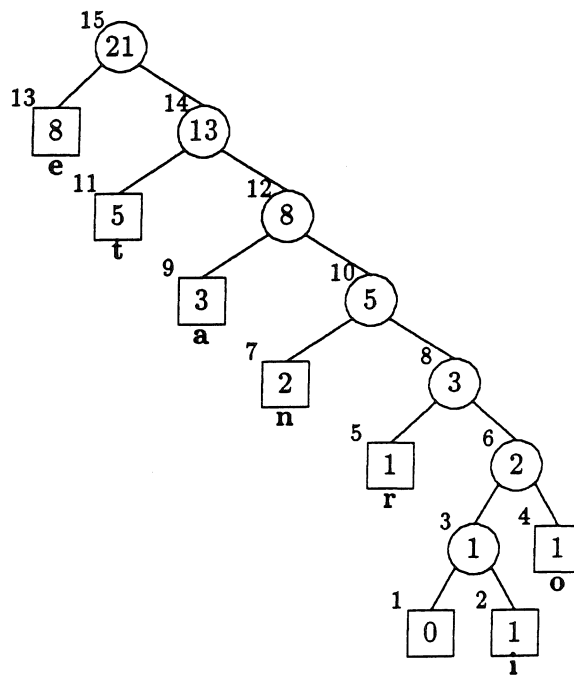


Figure 4.2 Tree formed by algorithm FGK for ensemble "eteaetneteareteaetnoi".

Disregarding overhead, the number of bits transmitted by algorithm FGK for the message ensemble of Figure 4.2 is 47. The static Huffman algorithm would transmit 53 bits in processing the same data. The overhead associated with the adaptive method is actually less than that of the static algorithm. In the adaptive case the only overhead is the $n \lg n$ bits needed to represent each of the n different source messages when they appear for the first time. (This is in fact conservative; rather than transmitting a unique code for each of the n source messages, the sender could transmit the message's position in the list of remaining messages and save a few bits in the average case.) In the static case, the source messages need to be sent as does the shape of the code tree. Vitter has proved that the total number of bits transmitted by algorithm FGK for a message ensemble of length t containing n distinct messages is bounded below by $S - n + 1$, where S is the performance of the static method, and bounded above by $2S + t - 4n + 2$ [Vitter 1985]. So the performance of algorithm FGK is never much worse than twice optimal. Knuth provides a complete implementation of algorithm FGK and a proof that the time required for each encoding or decoding operation is $O(l)$, where l is the current length of the codeword [Knuth 1985]. It should be noted that since the mapping is defined dynamically, during transmission, the encoding and decoding algorithms stand alone; there is no additional algorithm to determine the mapping as in static methods.

4.2 Algorithm V

The adaptive Huffman algorithm of Vitter (algorithm V) incorporates two improvements over algorithm FGK. First, the number of interchanges in which a node is moved upward in the tree during a recomputation is limited to one. This number is bounded in algorithm FGK only by $l/2$ where l is the length of the codeword for a_{t+1} when the recomputation begins. Second, Vitter's method minimizes the values of $\sum l_i$ and $\max\{l_i\}$ subject to the requirement of minimizing $\sum w_i l_i$. These improvements are accomplished through the use of a new system for numbering nodes. The numbering, called an implicit numbering, corresponds to a level ordering of the nodes (from bottom to top and left to right). Figure 4.3 illustrates that the numbering of algorithm FGK is not always a level ordering. The following invariant is maintained in Vitter's algorithm: For each weight w , all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w . Vitter proves that this invariant enforces the desired bound on node promotions [Vitter 1985]. The invariant also implements bottom merging, as discussed in Section 3.2, to minimize $\sum l_i$ and $\max\{l_i\}$. The difference between Vitter's method and algorithm FGK is in the way the tree is updated between transmissions. In order to understand the revised update operation, the following definition

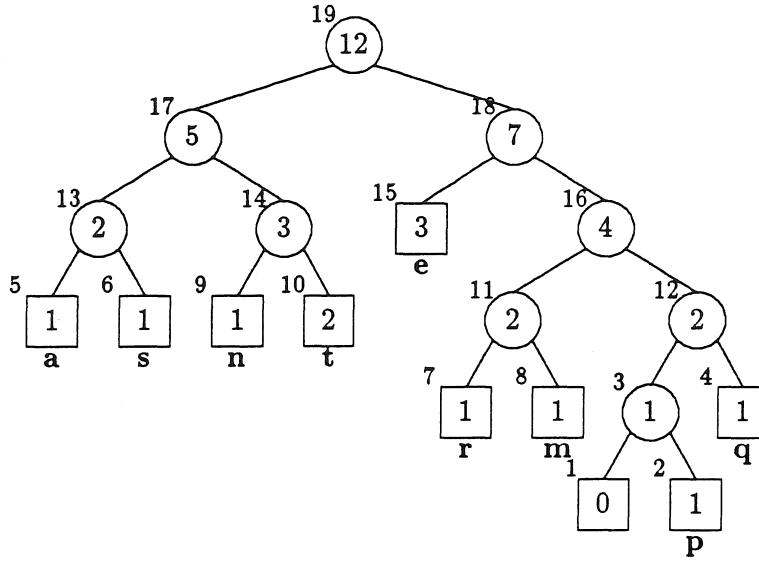


Figure 4.3 FGK tree with non-level order numbering.

of a block of nodes is necessary: Blocks are equivalence classes of nodes defined by $u \equiv v$ iff $weight(u) = weight(v)$ and u and v are either both leaves or both internal nodes. The leader of a block is the highest-numbered (in the implicit numbering) node in the block. Blocks are ordered by increasing weight with the convention that a leaf block always precedes an internal block of the same weight. When an exchange of nodes is required to maintain the sibling property, algorithm V requires that the node being promoted be moved to the position currently occupied by the highest-numbered node in the target block.

Figure 4.4 illustrates the tree built by Vitter's method for the ensemble of Figure 4.2. Both $\sum l_i$ and $\max\{l_i\}$ are smaller in the tree of Figure 4.4. The number of bits transmitted during the processing of the sequence is 47, the same used by algorithm FGK. However, if the transmission continues with n, r, o, i or an unused letter, the cost of algorithm V will be less than that of algorithm FGK. The reason is that while both algorithms have the property that immediately after the t^{th} letter is transmitted the adaptive Huffman tree is a valid Huffman tree with respect to the first t letters, Vitter's method has the additional property that the external path length ($\sum l_i$) and the height ($\max\{l_i\}$) are minimized as well. This is helpful for efficiently coding the $(t + 1)^{st}$ letter. Vitter proves that the performance of his algorithm is bounded by $S - n + 1$ from below and $S + t - 2n + 1$ from above [Vitter 1985]. At worst then, Vitter's adaptive method may transmit one more bit per codeword than the

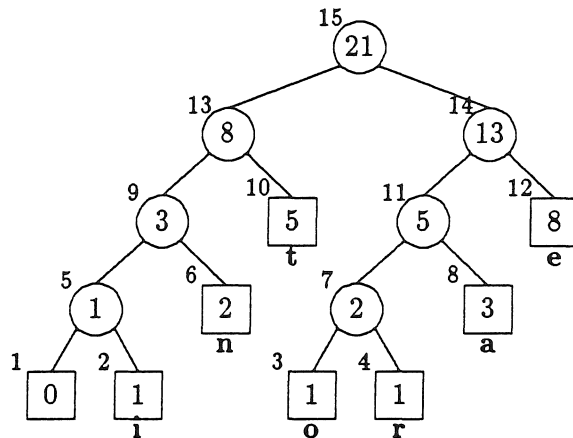


Figure 4.4 Tree formed by algorithm V for the ensemble of Fig. 4.2.

static Huffman method.

The improvements made by Vitter do not change the complexity of the algorithm; algorithm V encodes and decodes in $O(l)$ time as does algorithm FGK.

5. OTHER ADAPTIVE METHODS

Two more adaptive data compression methods, algorithm BSTW and Lempel-Ziv coding, are discussed in this section. These schemes diverge from the fundamental Huffman coding approach to a greater degree than the methods discussed in Section 4. Algorithm BSTW is a defined-word scheme which attempts to exploit locality. Lempel-Ziv coding is a *free-parse* method; that is, the words of the source alphabet are defined dynamically, as the encoding is performed. Algorithm BSTW is a variable-variable scheme, while Lempel-Ziv coding is variable-block.

5.1 Lempel-Ziv Codes

The Lempel-Ziv algorithm consists of a rule for parsing strings of symbols from a finite alphabet into substrings, or words, whose lengths do not exceed a prescribed integer L_1 ; and a coding scheme which maps these substrings sequentially into uniquely decipherable codewords of fixed length L_2 [Ziv and Lempel 1977]. The strings are selected so that they have very nearly equal probability of occurrence. As a result, frequently-occurring symbols

are grouped into longer strings while infrequent symbols appear in short strings. This strategy is effective at exploiting redundancy due to symbol frequency, character repetition, and high-usage patterns. Figure 5.1 shows a small Lempel-Ziv code table. Low-frequency letters such as Z are assigned individually to fixed-length codewords (in this case, 12 bit binary numbers represented in base ten for readability). Frequently-occurring symbols, such as blank (represented by b) and zero, appear in long strings. Effective compression is achieved when a long string is replaced by a single 12-bit code.

Symbol String	Code
A	1
T	2
AN	3
TH	4
THE	5
AND	6
AD	7
b	8
bb	9
bbb	10
0	11
00	12
000	13
0000	14
Z	15
###	4095

Figure 5.1 A Lempel-Ziv code table.

The Lempel-Ziv method is an incremental parsing strategy in which the coding process is interlaced with a learning process for varying source characteristics [Ziv and Lempel 1977]. In Figure 5.1, run-length encoding of zeros and blanks is being learned.

The Lempel-Ziv algorithm parses the source ensemble into a collection of segments of gradually increasing length. At each encoding step, the longest prefix of the remaining

source ensemble which matches an existing table entry (α) is parsed off, along with the character (c) following this prefix in the ensemble. The new source message, αc , is added to the code table. The new table entry is coded as (i, c) where i is the codeword for the existing table entry and c is the appended character. For example, the ensemble 010100010 is parsed into $\{0, 1, 01, 00, 010\}$ and is coded as $\{(0, 0), (0, 1), (1, 1), (1, 0), (3, 0)\}$. The table built for the message ensemble “*ababcbababaaaaaaaa*” is shown in Figure 5.2. The coded ensemble has the form: $\{(0, a), (0, b), (1, b), (0, c), (2, a), (5, b), (1, a), (7, a), (8, a)\}$. The string table is represented in a more efficient manner than in Figure 5.1; the string is represented by its prefix codeword followed by the extension character, so that the table entries have fixed length. The Lempel-Ziv strategy is simple, but greedy. It simply parses off the longest recognized string each time rather than searching for the best way to parse the ensemble.

Message	Codeword
<i>a</i>	1
<i>b</i>	2
<i>1b</i>	3
<i>c</i>	4
<i>2a</i>	5
<i>5b</i>	6
<i>1a</i>	7
<i>7a</i>	8
<i>8a</i>	9

Figure 5.2 Lempel-Ziv table for the message ensemble “*ababcbababaaaaaaaa*”.

The Lempel-Ziv method specifies fixed-length codewords. The size of the table and the maximum source length message are determined by the length of the codewords. It should be clear from the definition of the algorithm that Lempel-Ziv codes tend to be quite inefficient during the initial portion of the message ensemble. If the codeword length is not sufficiently large, Lempel-Ziv codes may also rise slowly to reasonable efficiency, maintain good performance briefly, and fail to make any gains once the table is full and messages can no longer be added. If the ensemble’s characteristics vary over time, the method may be “stuck with” the behavior it has learned and may be unable to continue to adapt.

Lempel-Ziv coding is asymptotically optimal, meaning that the redundancy approaches zero as the length of the source ensemble tends to infinity. However, for particular finite

sequences, the compression achieved may be far from optimal [Storer and Szymanski 1982]. When the method begins, each source symbol is coded individually. In the case of 6- or 8-bit source symbols and 12-bit codewords, the method yields as much as 50% expansion during initial encoding. This initial inefficiency can be mitigated somewhat by initializing the string table to contain all of the source characters. Implementation issues are particularly important in Lempel-Ziv methods. A straightforward implementation takes $O(n^2)$ time to process a string of n symbols; for each encoding operation, the existing table must be scanned for the longest message occurring as a prefix of the remaining ensemble. Rodeh addresses the issue of computational complexity by defining a linear implementation of Lempel-Ziv coding based on suffix trees [Rodeh et al. 1981]. The Rodeh scheme is asymptotically optimal, but an input must be very long in order to allow efficient compression, and the memory requirements of the scheme are large, $O(n)$ where n is the length of the source ensemble. It should also be mentioned that the Rodeh method constructs a variable-variable code; the pair (i, c) is coded using a representation of the integers, such as the Elias codes, for i and for c (a letter c can always be coded as the k^{th} member of the source alphabet for some k).

The other major implementation consideration involves the way in which the string table is stored and accessed. Welch suggests that the table be indexed by the codewords (integers $1 \dots 2^L$ where L is the maximum codeword length) and that the table entries be fixed-length codeword-extension character pairs [Welch 1984]. Hashing is proposed to assist in encoding. Decoding becomes a recursive operation, in which the codeword yields the final character of the substring and another codeword. The decoder must continue to consult the table until the retrieved codeword is 0. Unfortunately, this strategy peels off extension characters in reverse order and some type of stack operation must be used to reorder the source.

Storer and Szymanski present a general model for data compression which encompasses Lempel-Ziv coding [Storer and Szymanski 1982]. Their broad theoretical work compares classes of *macro schemes*, where macro schemes include all methods which factor out duplicate occurrences of data and replace them by references either to the source ensemble or to a code table. They also contribute a linear-time Lempel-Ziv-like algorithm with better performance than the standard Lempel-Ziv method.

Rissanen extends the Lempel-Ziv incremental parsing approach [Rissanen 1983]. Abandoning the requirement that the substrings partition the ensemble, the Rissanen method gathers "contexts" in which each symbol of the string occurs. The contexts are substrings of the previously encoded string (as in Lempel-Ziv), have varying size, and are in general overlapping. The Rissanen method hinges upon the identification of a design parameter

capturing the concept of “relevant” contexts. The problem of finding the best parameter is undecidable, and Rissanen suggests estimating the parameter experimentally.

5.2 Algorithm BSTW

The most recent of the algorithms surveyed here is due to Bentley, Sleator, Tarjan and Wei [Bentley et al. 1986]. This method, algorithm BSTW, possesses the advantage that it requires only one pass over the data to be transmitted yet has performance which compares well to that of the static two-pass method along the dimension of number of bits per word transmitted. It incorporates the additional benefit of taking advantage of locality of reference, the tendency for words to occur frequently for short periods of time then fall into long periods of disuse. The algorithm uses a self-organizing list as an auxiliary data structure and employs shorter encodings for words near the front of this list. There are many strategies for maintaining self-organizing lists (see [Hester and Hirschberg 1985]); algorithm BSTW uses move-to-front.

A simple example serves to outline the method of algorithm BSTW. As in other adaptive schemes, sender and receiver maintain identical representations of the code; in this case message lists which are updated at each transmission, using the move-to-front heuristic. These lists are initially empty. When message a_t is transmitted, if a_t is on the sender’s list, he transmits its current position. He then updates his list by moving a_t to position 1 and shifting each of the other messages down one position. The receiver similarly alters his word list. If a_t is being transmitted for the first time, then $k + 1$ is the “position” transmitted, where k is the number of distinct messages transmitted so far. Some representation of the message itself must be transmitted as well, but just this first time. Again, a_t is moved to position one by both sender and receiver subsequent to its transmission. For the ensemble “abcadeabfd”, the transmission would be (for ease of presentation, the position is represented in base ten):

1 a 2 b 3 c 3 4 d 5 e 3 5 6 f 5

As the example shows, algorithm BSTW transmits each source message once; the rest of its transmission consists of encodings of list positions. Therefore, an essential feature of algorithm BSTW is a reasonable scheme for representation of the integers. The methods discussed by Bentley are the Elias codes presented in Section 3.3. The simple scheme, code γ , involves prefixing the binary representation of the integer i with $\lfloor \lg i \rfloor$ zeros. This yields a prefix code with the length of the codeword for i equal to $2\lfloor \lg i \rfloor + 1$. Greater compression can be gained through use of the more sophisticated scheme, δ , which encodes an integer i in $1 + \lfloor \lg i \rfloor + 2\lfloor \lg(1 + \lfloor \lg i \rfloor) \rfloor$ bits.

A message ensemble on which algorithm BSTW is particularly efficient, described by Bentley, is formed by repeating each of n messages n times, for example $1^n 2^n 3^n \dots n^n$. Disregarding overhead, a static Huffman code uses $n^2 \lg n$ bits (or $\lg n$ bits per message), while algorithm BSTW uses $n^2 + \sum_{i=1}^n \lceil \lg i \rceil$ (which is essentially $n^2 + 2n \lg n$, or $O(1)$ bits per message). The overhead for algorithm BSTW consists of just the $n \lg n$ bits needed to transmit each source letter once. As discussed in Section 3.2, the overhead for static Huffman coding includes an additional $2n$ bits.

Bentley proves that with the simple scheme for encoding integers, the performance of algorithm BSTW is bounded above by $2S + 1$, where S is the cost of the static Huffman coding scheme [Bentley et al. 1986]. Using the more sophisticated integer encoding scheme, the bound is $1 + S + 2 \lg(1 + S)$. A key idea in the proofs given by Bentley is the fact that, using the move-to-front heuristic, the integer transmitted for a message a_t will be one more than the number of different words transmitted since the last occurrence of a_t . Bentley also proves that algorithm BSTW is asymptotically optimal.

An implementation of algorithm BSTW is described in great detail by Bentley [Bentley et al. 1986]. In this implementation, encoding an integer consists of a table lookup; the codewords for the integers from 1 to $n + 1$ are stored in an array indexed from 1 to $n + 1$. A binary trie is used to store the inverse mapping, from codewords to integers. Decoding an Elias codeword to find the corresponding integer involves following a path in the trie. Two interlinked data structures, a binary trie and a binary tree, are used to maintain the word list. The trie is based on the binary encodings of the source words. Mapping a source message a_i to its list position p involves following a path in the trie, following a link to the tree, and then computing the symmetric order position of the tree node. Finding the source message a_i in position p is accomplished by finding the symmetric order position p in the tree and returning the word stored there. Using this implementation, the work done by sender and receiver is $O(\text{length}(a_i) + \text{length}(w))$ where a_i is the message being transmitted and w the codeword representing a_i 's position in the list. If the source alphabet consists of single characters, then the complexity of algorithm BSTW is just $O(\text{length}(w))$.

6. EMPIRICAL RESULTS

Empirical tests of the efficiencies of the algorithms presented here are reported in [Bentley et al. 1986; Knuth 1985; Schwartz and Kallick 1964; Vitter 1985; Welch 1984]. These experiments compare the number of bits per word required and processing time is not reported. While theoretical considerations bound the performance of the various algorithms, experi-

mental data is invaluable in providing additional insight. It is clear that the performance of each of these methods is dependent upon the characteristics of the source ensemble.

Schwartz and Kallick test an implementation of static Huffman coding in which bottom merging is used to determine codeword lengths and all codewords of a given length are sequential binary numbers [Schwartz and Kallick 1964]. The source alphabet in the experiment consists of 5,114 frequently-used English words, 27 geographical names, 10 numerals, 14 symbols, and 43 suffixes. The entropy of the document is 8.884 binary digits per message and the average codeword constructed has length 8.920. The same document is also coded one character at a time. In this case, the entropy of the source is 4.03 and the coded ensemble contains an average of 4.09 bits per letter. The redundancy is low in both cases. However, the *relative redundancy* (i.e., redundancy / entropy) is lower when the document is encoded by words.

Knuth describes algorithm FGK's performance on three types of data: a file containing the text of Grimm's first ten Fairy Tales, text of a technical book, and a file of graphical data [Knuth 1985]. For the first two files, the source messages are individual characters and the alphabet size is 128. The same data is coded using pairs of characters, so that the alphabet size is 1968. For the graphical data, the number of source messages is 343. In the case of the Fairy Tales the performance of FGK is very close to optimum, although performance degrades with increasing file size. Performance on the technical book is not as good, but is still respectable. The graphical data proves harder yet to compress, but again FGK performs reasonably well. In the latter two cases, the trend of performance degradation with file size continues. Defining source messages to consist of character pairs results in slightly better compression, but the difference would not appear to justify the increased memory requirement imposed by the larger alphabet.

n	k	Static	Alg. V	Alg. FGK
100	96	664	569	659
500	96	3320	3233	3339
961	97	6417	6330	6432

Figure 6.1 Simulation results for a small text file [Vitter 1985]; n = file size in bytes, k = number of distinct messages.

Vitter tests the performance of algorithms V and FGK against that of static Huffman coding. Each method is run on data which includes Pascal source code, the T_EX source

of the author's thesis, and electronic mail files [Vitter 1985]. Figure 6.1 summarizes the results of the experiment for a small file of text. The performance of each algorithm is measured by the number of bits in the coded ensemble and overhead costs are not included. Figure 6.2 presents data for Pascal source code. For the T_EX source, the alphabet consists of 128 individual characters; for the other two file types, only 97 characters appear. For each experiment, when the overhead costs are taken into account, algorithm V outperforms static Huffman coding as long as the size of the message ensemble (number of characters) is no more than 10⁴. Algorithm FGK displays slightly higher costs, but never more than 104% of the static algorithm.

n	k	Static	Alg. V	Alg. FGK
100	32	459	450	471
500	49	2460	2486	2522
1000	57	4902	4945	4993
10000	73	47807	47942	48020
12067	78	57592	57737	57822

Figure 6.2 Simulation results for Pascal source code [Vitter 1985]; n = file size in bytes, k = number of distinct messages.

Bentley uses C and Pascal source files, TROFF source files, and a terminal session transcript of several hours for experiments which compare the performance of algorithm BSTW to static Huffman coding. Here the defined words consist of two disjoint classes, sequences of alphanumeric characters and sequences of nonalphanumeric characters. The performance of algorithm BSTW is very close to that of static Huffman coding in all cases. The experiments reported by Bentley are of particular interest in that they incorporate another dimension, the possibility that in the move-to-front scheme one might want to limit the size of the data structure containing the codes to include only the m most recent words, for some m [Bentley et al. 1986]. The tests consider cache sizes of 8, 16, 32, 64, 128 and 256. Although performance tends to increase with cache size, the increase is erratic, with some documents exhibiting nonmonotonicity (performance which increases with cache size to a point and then decreases when cache size is further increased).

Welch reports simulation results for Lempel-Ziv codes in terms of compression ratios [Welch 1984]. His definition of compression ratio is the one given in Section 1.3, $C = (\text{average message length}) / (\text{average codeword length})$. The ratios reported are: 1.8 for English text, 2

to 6 for Cobol data files, 1.0 for floating point arrays, 2.1 for formatted scientific data, 2.6 for system log data, 2.3 for source code, and 1.5 for object code. The tests involving English text files showed that long individual documents did not compress better than groups of short documents. This observation is somewhat surprising, in that it seems to refute the intuition that redundancy is due at least in part to correlation in content. For purposes of comparison, Welch cites results of Pechura and Rubin. Pechura achieved a 1.5 compression ratio using static Huffman coding on files of English text [Pechura 1982]. Rubin reports a 2.4 ratio for English text when employing a complex technique for choosing the source messages to which Huffman coding is applied [Rubin 1976]. These results provide only a very weak basis for comparison, since the characteristics of the files used by the three authors are unknown. It is very likely that a single algorithm may produce compression ratios ranging from 1.5 to 2.4, depending upon the source to which it is applied.

7. SUSCEPTIBILITY TO ERROR

The discrete noiseless channel is, unfortunately, not a very realistic model of a communication system. Actual data transmission systems are prone to two types of error: *phase error*, in which a code symbol is lost or gained; and *amplitude error*, in which a code symbol is corrupted [Neumann 1962]. The degree to which channel errors degrade transmission is an important parameter in the choice of a data compression method. The susceptibility to error of a coding algorithm depends heavily on whether the method is static or adaptive.

7.1 Static Codes

It is generally known that Huffman codes tend to be self-correcting [Standish 1980]. That is, a transmission error tends not to propagate too far. The codeword in which the error occurs is incorrectly received and it is likely that several subsequent codewords are misinterpreted but, before too long, the receiver is back in synchronization with the sender. In a static code, synchronization means simply that both sender and receiver identify the beginnings of the codewords in the same way. In Figure 7.1, an example is used to illustrate the ability of a Huffman code to recover from phase errors. The message ensemble “*BCDAEB*” is encoded using the Huffman code of Figure 3.3 where the source letters $a_1 \dots a_5$ represent *A...E* respectively, yielding the coded ensemble “0110100011000011”. Figure 7.1 demonstrates the impact of loss of the first bit, the second bit, or the fourth bit. The dots show the way in which each line is parsed into codewords. The loss of the first bit results in re-synchronization after the third bit so that only the first source message (*B*) is lost (replaced by *AA*). When the second bit is lost, the first eight bits of the coded ensemble are

misinterpreted and synchronization is regained by bit 9. Dropping the fourth bit causes the same degree of disturbance as dropping the second.

0 1 1 . 0 1 0 . 0 0 1 . 1 . 0 0 0 . 0 1 1	coded ensemble (<i>BCDAEB</i>)
1 . 1 . 0 1 0 . 0 0 1 . 1 . 0 0 0 . 0 1 1	bit 1 is lost, interpreted as <i>AACDAEB</i>
0 1 0 . 1 . 0 0 0 . 1 . 1 . 0 0 0 . 0 1 1	bit 2 is lost, interpreted as <i>CAEAAEB</i>
0 1 1 . 1 . 0 0 0 . 1 . 1 . 0 0 0 . 0 1 1	bit 4 is lost, interpreted as <i>BAEAAEB</i>

Figure 7.1 Recovery from phase errors

The effect of amplitude errors is demonstrated in Figure 7.2. The format of the illustration is the same as that in Figure 7.1. This time bits 1, 2, and 4 are inverted rather than lost. Again synchronization is regained almost immediately. When bit 1 or bit 2 is changed, only the first three bits (the first character of the ensemble) are disturbed. Inversion of bit four causes loss of synchronization through the ninth bit. A very simple explanation of the self-synchronization present in these example can be given. Since many of the codewords end in the same sequence of digits, the decoder is likely to reach a leaf of the Huffman code tree at one of the codeword boundaries of the original coded ensemble. When this happens, the decoder is back in synchronization with the encoder.

0 1 1 . 0 1 0 . 0 0 1 . 1 . 0 0 0 . 0 1 1	coded ensemble (<i>BCDAEB</i>)
1 . 1 . 1 . 0 1 0 . 0 0 1 . 1 . 0 0 0 . 0 1 1	bit 1 is inverted, interpreted as <i>AAACDAEB</i>
0 0 1 . 0 1 0 . 0 0 1 . 1 . 0 0 0 . 0 1 1	bit 2 is inverted, interpreted as <i>DCDAEB</i>
0 1 1 . 1 . 1 . 0 0 0 . 1 . 1 . 0 0 0 . 0 1 1	bit 4 is inverted, interpreted as <i>BAAEAAEB</i>

Figure 7.2 Recovery from amplitude errors

So that self-synchronization may be discussed more carefully, the following definitions are presented. (It should be noted that these definitions hold for arbitrary prefix codes, so that the discussion includes all of the codes described in Section 3.) If s is a suffix of some codeword and there exist sequences of codewords Γ and Δ such that $s\Gamma = \Delta$, then Γ is said to be a *synchronizing sequence* for s . For example, in the Huffman code used above, 1 is a synchronizing sequence for the suffix 01 while both 000001 and 011 are synchronizing sequences for the suffix 10. If every suffix (of every codeword) has a synchronizing sequence, then the code is *completely self-synchronizing*. If some or none of the proper suffixes have synchronizing sequences, then the code is, respectively, *partially-* or *never-self-synchronizing*. Finally, if there exists a sequence Γ which is a synchronizing sequence for every suffix, Γ is defined to be a *universal synchronizing sequence*. The code used in the examples above is completely self-synchronizing, and has universal synchronizing sequence 00000011000. Gilbert and Moore prove that the existence of a universal synchronizing sequence is a necessary as well as a sufficient condition for a code to be completely self-synchronizing [Gilbert and Moore 1959]. They also state that any prefix code which is completely self-synchronizing will synchronize itself with probability 1 if the source ensemble consists of successive messages independently chosen with any given set of probabilities. This is true since the probability of occurrence of the universal synchronizing sequence at any given time is positive.

It is important to realize that the fact that a completely self-synchronizing code will re-synchronize with probability 1 does not guarantee recovery from error with bounded delay. In fact, for every completely self-synchronizing prefix code with more than two codewords, there are errors within one codeword which cause unbounded error propagation [Neumann 1962]. In addition, prefix codes are not always completely self-synchronizing. Bobrow and Hakimi state a necessary condition for a prefix code with codeword lengths $l_1 \dots l_r$ to be completely self-synchronizing: the greatest common divisor of the l_i must be equal to one [Bobrow and Hakimi 1969]. The Huffman code {00, 01, 10, 1100, 1101, 1110, 1111} is not completely self-synchronizing, but is partially self-synchronizing since suffixes 00, 01 and 10 are synchronized by any codeword. The Huffman code {000, 0010, 0011, 01, 100, 1010, 1011, 100, 111} is never-self-synchronizing. Examples of never-self-synchronizing Huffman codes are difficult to construct, and the example above is the only one with fewer than 16 source messages. Stiffler proves that a code is never-self-synchronizing if and only if none of the proper suffixes of the codewords are themselves codewords [Stiffler 1971].

The conclusions which may be drawn from the above discussion are: while it is common for Huffman codes to self-synchronize, this is not guaranteed; and when self-synchronization is assured, there is no bound on the propagation of the error. An additional difficulty is that

self-synchronization provides no indication that an error has occurred.

The problem of error detection and correction in connection with Huffman codes has not received a great deal of attention. Several ideas on the subject are reported here. Rudner states that synchronizing sequences should be as short as possible to minimize re-synchronization delay. In addition, if a synchronizing sequence is used as the codeword for a high probability message, then re-synchronization will be more frequent. A method for constructing a minimum-redundancy code having the shortest possible synchronizing sequence is described by Rudner [Rudner 1971]. Neumann suggests purposely adding some redundancy to Huffman codes in order to permit detection of certain types of errors [Neumann 1962]. Clearly this has to be done carefully, so as not to negate the redundancy reduction provided by Huffman coding. McIntyre and Pechura cite data integrity as an advantage of the codebook approach discussed in Section 3.2 [McIntyre and Pechura 1985]. When the code is stored separately from the coded data, the code may be backed up to protect it from perturbation. However, when the code is stored or transmitted with the data, it is susceptible to errors. An error in the code representation constitutes a drastic loss and therefore extreme measures for protecting this part of the transmission are justified.

7.2 Adaptive Codes

Adaptive codes are far more adversely affected by transmission errors than are static codes. For example, in the case of a adaptive Huffman code, even though the receiver may re-synchronize with the sender in terms of correctly locating the beginning of a codeword, the information lost represents more than a few bits or a few characters of the source ensemble. The fact that sender and receiver are dynamically redefining the code indicates that by the time synchronization is regained, they may have radically different representations of the code. Synchronization as defined in Section 7.1 refers to synchronization of the bit stream, which is not sufficient for adaptive methods. What is needed here is *code synchronization*, that is, synchronization of both the bit stream and the dynamic data structure representing the current code mapping.

There is no evidence that adaptive methods are self-synchronizing. Bentley notes that, in algorithm BSTW, loss of synchronization can be catastrophic, whereas this is not true with static Huffman coding [Bentley et al. 1986]. Ziv and Lempel recognize that the major drawback of their algorithm is its susceptibility to error propagation [Ziv and Lempel 1977]. Welch also considers the problem of error tolerance of Lempel-Ziv codes and suggests that the entire ensemble be embedded in an error-detecting code [Welch 1984].

8. NEW DIRECTIONS

Data compression is still very much an active research area. This section suggests possibilities for further study.

The discussion of Section 7 illustrates the susceptibility to error of the codes presented in this survey. Strategies for increasing the reliability of these codes while incurring only a moderate loss of efficiency would be of great value. This area appears to be largely unexplored. Possible approaches include embedding the entire ensemble in an error-correcting code or reserving one or more codewords to act as error flags. For adaptive methods it may be necessary for receiver and sender to verify the current code mapping periodically.

For adaptive Huffman coding, Gallager suggests an "aging" scheme, whereby recent occurrences of a character contribute more to its frequency count than do earlier occurrences [Gallager 1978]. This strategy introduces the notion of locality into the adaptive Huffman scheme. Cormack and Horspool describe an algorithm for approximating exponential aging [Cormack and Horspool 1984]. However, the effectiveness of this algorithm has not been established.

Both Knuth and Bentley suggest the possibility of using the "cache" concept to exploit locality and minimize the effect of anomalous source messages. Preliminary empirical results indicate that this may be helpful [Knuth 1985; Bentley et al. 1986]. A problem related to the use of a cache is overhead time required for deletion. Strategies for reducing the cost of a deletion could be considered. Another possible extension to algorithm BSTW is to investigate other locality heuristics. Bentley proves that intermittent-move-to-front (move-to-front after every k occurrences) is as effective as move-to-front [Bentley et al. 1986]. It should be noted that there are many other self-organizing methods yet to be considered.

Several aspects of free-parse methods merit further attention. Lempel-Ziv codes appear to be promising, although the absence of a worst-case bound on the redundancy of an individual finite source ensemble is a drawback. The variable-block type Lempel-Ziv codes have been implemented with some success [ARC 1986] and the construction of a variable-variable Lempel-Ziv code has been sketched [Ziv and Lempel 1978]. The efficiency of the variable-variable model should be investigated. In addition, an implementation of Lempel-Ziv coding which combines the time efficiency of Rodeh's method with more efficient use of space is worthy of consideration.

9. SUMMARY

Data compression is a topic of much importance and many applications. Methods of data compression have been studied for almost four decades. This paper has provided an overview of data compression methods of general utility. The algorithms have been evaluated in terms of the amount of compression they provide, algorithm efficiency, and susceptibility to error. While algorithm efficiency and susceptibility to error are relatively independent of the characteristics of the source ensemble, the amount of compression achieved depends upon the characteristics of the source to a great extent.

Semantic dependent data compression techniques, as discussed in Section 2, are special-purpose methods designed to exploit local redundancy or context information. A semantic dependent scheme can usually be viewed as a special case of one or more general-purpose algorithms. It should also be noted that algorithm BSTW is a general-purpose technique which exploits locality of reference, a type of local redundancy.

Susceptibility to error is the main drawback of each of the algorithms presented here. Although channel errors are more devastating to adaptive algorithms than to static ones, it is possible for an error to propagate without limit even in the static case. Methods of limiting the effect of an error on the effectiveness of a data compression algorithm should be investigated.

REFERENCES

- Abrahamson, N. 1963. *Information Theory and Coding*. McGraw-Hill, New York.
- ARC File Archive Utility*. 1986. Version 5.1. System Enhancement Associates. Wayne, N. J.
- Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. 1986. A Locally Adaptive Data Compression Scheme. *Commun. ACM* 29, 4 (Apr.), 320-330.
- Bobrow, L. S., and Hakimi, S. L. 1969. Graph Theoretic Prefix Codes and Their Synchronizing Properties. *Inform. Contr.* 15, 1 (July), 70-94.
- Brent, R., and Kung, H. T. 1978. Fast Algorithms for Manipulating Formal Power Series. *J. ACM* 25, 4 (Oct.), 581-595.
- Cappellini, V., Ed. 1985. *Data Compression and Error Control Techniques with Applications*. Academic Press, London.
- Connell, J. B. 1973. A Huffman-Shannon-Fano Code. *Proc. IEEE* 61, 7 (July), 1046-1047.
- Cormack, G. V. 5. Data Compression on a Database System. *Commun. ACM* 28, 12 (Dec.), 1336-1342.
- Cormack, G. V., and Horspool, R. N. 1984. Algorithms for Adaptive Huffman Codes. *Inform. Process. Lett.* 18, 3 (Mar.), 159-165.
- Cortesi, D. 1982. An Effective Text-Compression Algorithm. *BYTE* 7, 1 (Jan.), 397-403.
- Cot, N. 1977. Characterization and Design of Optimal Prefix Codes. Ph. D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif.
- Elias, P. 1975. Universal Codeword Sets and Representations of the Integers. *IEEE Trans. Inform. Theory* 21, 2 (Mar.), 194-203.

- Faller, N. 1973. An Adaptive System for Data Compression. *Record of the 7th Asilomar Conf. on Circuits, Systems and Computers* (Pacific Grove, Ca., Nov.), 593-597.
- Fano, R. M. 1949. *Transmission of Information*. M. I. T. Press, Cambridge, Mass.
- Fraenkel, A. S., Mor, M., and Perl, Y. 1983. Is Text Compression by Prefixes and Suffixes Practical? *Acta Inf.* 20, 4 (Dec.), 371-375.
- Gallager, R. G. 1968. *Information Theory and Reliable Communication*, Wiley, New York.
- Gallager, R. G. 1978. Variations on a Theme by Huffman. *IEEE Trans. Inform. Theory* 24, 6 (Nov.), 668-674.
- Garey, M. R. 1974. Optimal Binary Search Trees with Restricted Maximal Depth. *SIAM J. Comput.* 3, 2 (June), 101-110.
- Gilbert, E. N. 1971. Codes Based on Inaccurate Source Probabilities. *IEEE Trans. Inform. Theory* 17, 3, (May), 304-314.
- Gilbert, E. N., and Moore, E. F. 1959. Variable-Length Binary Encodings. *Bell System Tech. J.* 38, 4 (July), 933-967.
- Glasse, C. R., and Karp, R. M. 1976. On the Optimality of Huffman Trees. *SIAM J. Appl. Math* 31, 2 (Sept.), 368-378.
- Gonzalez, R. C., and Wintz, P. 1977. *Digital Image Processing*. Addison-Wesley, Reading, Mass.
- Hahn, B. 1974. A New Technique for Compression and Storage of Data. *Commun. ACM* 17, 8 (Aug.), 434-436.
- Hester, J. H., and Hirschberg, D. S. 1985. Self-Organizing Linear Search. *ACM Comput. Surv.* 17, 3 (Sept.), 295-311.
- Hu, T. C., and Tan, K. C. 1972. Path Length of Binary Search Trees. *SIAM J. Appl. Math* 22, 2 (Mar.), 225-234.
- Hu, T. C., and Tucker, A. C. 1971. Optimal Computer Search Trees and Variable-Length Alphabetic Codes. *SIAM J. Appl. Math* 21, 4 (Dec.), 514-532.

- Huffman, D. A. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE* 40, 9 (Sept.), 1098-1101.
- Ingels, F. M. 1971. *Information and Coding Theory*. Intext, Scranton, Penn.
- Itai, A. 1976. Optimal Alphabetic Trees. *SIAM J. Comput.* 5, 1 (Mar.), 9-18.
- Karp, R. M. 1961. Minimum Redundancy Coding for the Discrete Noiseless Channel. *IRE Trans. Inform. Theory* 7, 1 (Jan.), 27-38.
- Knuth, D. E. 1971. Optimum Binary Search Trees. *Acta Inf.* 1, 1 (Jan.), 14-25.
- Knuth, D. E. 1985. Dynamic Huffman Coding. *J. Algorithms* 6, 2 (June), 163-180.
- Krause, R. M. 1962. Channels Which Transmit Letters of Unequal Duration. *Inform. Contr.* 5, 1 (Mar.), 13-24.
- Laeser, R. P., McLaughlin, W. I., and Wolff, D. M. 1986. Engineering Voyager 2's Encounter with Uranus *Scientific American* 255, 5 (Nov.), 5, 36-45.
- McIntyre, D. R., and Pechura, M. A. 1985. Data Compression Using Static Huffman Code-Decode Tables. *Commun. ACM* 28, 6 (June), 612-616.
- Mehlhorn, K. 1980. An Efficient Algorithm for Constructing Nearly Optimal Prefix Codes. *IEEE Trans. Inform. Theory* 26, 5 (Sept.), 513-517.
- Neumann, P. G. 1962. Efficient Error-Limiting Variable-Length Codes. *IRE Trans. Inform. Theory* 8, 4 (July), 292-304.
- Parker, D. S. 1980. Conditions for the Optimality of the Huffman Algorithm. *SIAM J. Comput.* 9, 3 (Aug.), 470-489.
- Pechura, M. 1982. File Archival Techniques Using Data Compression. *Commun. ACM* 25, 9 (Sept.), 605-609.
- Perl, Y., Garey, M. R., and Even, S. 1975. Efficient Generation of Optimal Prefix Code: Equiprobable Words Using Unequal Cost Letters. *J. ACM* 22, 2 (Apr.), 202-214.
- Reghbati, H. K. 1981. An Overview of Data Compression Techniques. *Computer* 14, 4 (Apr.), 71-75.

- Rissanen, J. 1983. A Universal Data Compression System. *IEEE Trans. Inform. Theory* 29, 5 (Sept.), 656-664.
- Rodeh, M., Pratt, V. R., and Even, S. 1981. A Linear Algorithm for Data Compression via String Matching. *J. ACM* 28, 1 (Jan.), 16-24.
- Rubin, F. 1976. Experiments in Text File Compression. *Commun. ACM* 19, 11 (Nov.), 617-623.
- Rudner, B. 1971. Construction of Minimum-Redundancy Codes with Optimal Synchronizing Property. *IEEE Trans. Inform. Theory* 17, 4 (July), 478-487.
- Ruth, S. S., and Kreutzer, P. J. 1972. Data Compression for Large Business Files. *Datamation* 18, 9 (Sept.), 62-66.
- Samet, H. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June), 187-260.
- Schwartz, E. S. 1964. An Optimum Encoding with Minimum Longest Code and Total Number of Digits. *Inform. Contr.* 7, 1 (Mar.), 37-44.
- Schwartz, E. S., and Kallick, B. 1964. Generating a Canonical Prefix Encoding. *Commun. ACM* 7, 3 (Mar.), 166-169.
- Shannon, C. E., and Weaver, W. 1949. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Ill.
- Snyderman, M., and Hunt, B. 1970. The Myriad Virtues of Text Compaction. *Datamation* 16, 12 (Dec.), 36-40.
- Standish, T. A. 1980. *Data Structure Techniques*. Addison-Wesley, Reading, Mass.
- Stiffler, J. J. 1971. *Theory of Synchronous Communications*. Prentice-Hall, Englewood Cliffs, N. J.
- Storer, J. A., and Szymanski, T. G. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (Oct.), 928-951.
- Tropper, R. 1982. Binary-Coded Text, A Text-Compression Method. *BYTE* 7, 4 (Apr.), 398-413.

UNIX User's Manual. 1984. 4.2. Berkeley Software Distribution, Virtual VAX-11 Version, University of California, Berkeley.

Varn, B. 1971. Optimal Variable Length Codes (Arbitrary Symbol Cost and Equal Code Word Probability). *Inform. Contr.* 19, 4 (Nov.), 289–301.

Vitter, J. S. 1985. Design and Analysis of Dynamic Huffman Coding. *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science* (Portland, Oreg., Oct.), IEEE, New York, 293–302.

Wagner, R. A. 1973. Common Phrases and Minimum-Space Text Storage. *Commun. ACM* 16, 3 (Mar.), 148–152.

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

Wilkins, L. C., and Wintz, P. A. 1971. Bibliography on Data Compression, Picture Properties and Picture Coding. *IEEE Trans. Inform. Theory* 17, 2, 180–197.

Zimmerman, S. 1959. An Optimal Search Procedure. *Amer. Math. Monthly* 66, (Oct.), 690–693.

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

Ziv, J., and Lempel, A. 1978. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inform. Theory* 24, 5 (Sept.), 530–536.

