Data Compression

DEBRA A. LELEWER and DANIEL S. HIRSCHBERG

Department of Information and Computer Science, University of California, Irvine, California 92717

This paper surveys a variety of data compression methods spanning almost 40 years of research, from the work of Shannon, Fano, and Huffman in the late 1940s 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.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Adaptive coding, adaptive Huffman codes, coding, coding theory, file compression, Huffman codes, minimum-redundancy codes, optimal codes, prefix codes, text compression

INTRODUCTION

Data compression is often referred to as coding, where coding is a general term encompassing any special representation of data that satisfies a given need. Information theory is defined as 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.

A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (e.g., of bits) that contains the same information but whose length is as small as possible. Data compression has important application in the areas of data transmission and data storage. Many data processing applications require storage of large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of computer communication networks is resulting in massive transfer of data over communication links. Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the effect is that of increasing the capacity of the communication channel. Similarly, compressing a file to half of its original size is equivalent to doubling the capacity of the storage medium. It may then become feasible to store the data at a

CONTENTS

higher, thus faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.

Many of the methods discussed in this paper are implemented in production systems. The UNIX[1] utilities *compact* and *compress* are based on methods discussed in Sections 4 and 5, respectively [UNIX 1984]. Popular file archival systems such as *ARC* and *PKARC* use techniques presented in Sections 3 and 5 [ARC 1986; PKARC 1987]. The savings achieved by data compression can be dramatic; reduction as high as 80% is not uncommon [Reghbati 1981]. Typical values of compression provided by *compact* are text (38%), Pascal source (43%), C source (36%), and binary (19%). *Compress* generally achieves better compression (50–60% for text such as source code and English) and takes less time to compute [UNIX 1984]. Arithmetic coding (Section 3.4) has been reported to reduce a file to anywhere from 12.1 to 73.5% of its original size [Witten et al. 1987]. Cormack reports that data compression programs based on Huffman coding (Section 3.2) reduced the size of a large student-record database by 42.1% when only some of the information was compressed. As a consequence of this size reduction, the number of disk operations required to load the database was reduced by 32.7% [Cormack 1985]. Data compression routines developed with specific applications in mind have achieved compression factors as high as 98% [Severance 1983].

Although coding for purposes of data security (cryptography) and codes that guarantee a certain level of data integrity (error detection/correction) are topics worthy of attention, they 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. That is, we assume a system in which a sequence of symbols chosen from a finite alphabet can be transmitted from one point to another without the possibility of error. 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. As noted earlier, data compression is of value in data storage as well. Although this discussion is framed in the terminology of data transmission, compression and decompression of data files are essentially the same tasks 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 [1985] for a discussion of techniques with natural hardware implementation.

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 that follows.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Although 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 quadtrees, 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. These descriptions are sufficiently detailed to provide an understanding of the techniques. The reader will need to consult the references for implementation details. In most cases only worst-case analyses of the methods are feasible. To provide a more realistic picture of their effectiveness, empirical data are 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. The following string of characters is used to illustrate the concepts defined: *EXAMPLE* = "*aa bbb cccc ddddd eeeeee fffffffgggggggg*".

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). The source messages are the basic units into which the string to be represented is partitioned. These basic units may be single symbols from the source alphabet, or they may be strings of symbols. For string *EXAMPLE*, α = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *space*}. For purposes of explanation, β is taken to be {0, 1}. 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. A block–block code for *EXAMPLE* is shown in Figure 1, and a variable–variable code is given in Figure 2. If the string *EXAMPLE* were coded using the Figure 1 code, the length of the coded message would be 120; using Figure 2 the length would be 30.

Source message	Codeword
a	000
b	001
c	010
d	011
e	100
f	101
g	110
space	111

Figure 1. A block–block code for *EXAMPLE*.

Source message	Codeword
aa	0
bbb	1
cccc	10
ddddd	11
eeeeee	100
fffffff	101
gggggggg	110
space	111

Figure 2. A variable–variable code for *EXAMPLE*.

The oldest and most widely used codes, ASCII and EBCDIC, are examples of block–block codes, mapping an alphabet of 64 (or 256) single characters onto 6-bit (or 8-bit) codewords. These are not discussed, since 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 before 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 nonalphanumeric 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. Most of the known data compression methods are defined-word schemes; the free-parse model differs in a fundamental way from the classical coding paradigm.

A code is *distinct* if each codeword is distinguishable from every other (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. Clearly, each of these features is desirable. The codes of Figures 1 and 2 are both distinct, but the code of Figure 2 is not uniquely decodable. For example, the coded message 11 could be decoded as either "*ddddd*" or "*bbbbbb*". A uniquely decodable code is a *prefix code* (or *prefix-free code*) if it has the prefix property, which requires that no codeword be a proper prefix of any other codeword. All uniquely decodable block–block and variable–block codes are prefix codes. The code with codewords {1, 100000, 00} is an example of a code that is uniquely decodable but that does not have the prefix property. Prefix codes are *instantaneously decodable*; that is, they have the desirable property that the coded message can be parsed into codewords without the need for lookahead. In order to decode a message encoded using the codeword set {1, 100000, 00}, lookahead is required. For example, the first codeword of the message 1000000001 is 1, but this cannot be determined until the last (tenth) symbol of the message is read (if the string of zeros had been of odd length, the first codeword would have been 100000).

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. The set of codewords {00, 01, 10} is an example of a prefix code that is not minimal. The fact that 1 is a proper prefix of the codeword 10 requires that 11 be either a codeword or a proper prefix of a codeword, and it is neither. Intuitively, the minimality constraint prevents the use of codewords that are longer than necessary. In the above example the codeword 10 could be replaced by the codeword 1, yielding a minimal prefix code with shorter codewords. The codes discussed in this paper are all minimal prefix codes.

In this section a *code* has been defined to be a mapping from a source alphabet to a code alphabet; we now define related terms. The process of transforming a source ensemble into a coded message is *coding* or *encoding*. The encoded message may be referred to as an encoding of the source ensemble. The algorithm that constructs the mapping and uses it to transform the source ensemble is called the *encoder*. The *decoder* performs the inverse operation, restoring the coded message to its original form.

1.2 Classification of Methods

Not only are data compression schemes categorized with respect to message and codeword lengths, but they are also 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 that appear frequently are represented by short codewords; messages with smaller probabilities map to longer codewords. These probabilities are determined before transmission begins. A Huffman code for the ensemble *EXAMPLE* is given in Figure 3. If *EXAMPLE* were coded using this Huffman mapping, the length of the coded message would be 117. Static Huffman coding is discussed in Section 3.2;

Source message	Probability	Codeword
a	2/40	1001
b	3/40	1000
c	4/40	011
d	5/40	010
e	6/40	111
f	7/40	110
g	8/40	00
space	5/40	101

Figure 3. A Huffman code for the message *EXAMPLE* (code length = 117).

other static schemes are discussed in Sections 2 and 3.

A code is *dynamic* if the mapping from the set of messages to the set of codewords changes over time. For example, 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. As an illustration, Figure 4 presents a dynamic Huffman code table corresponding to the prefix "*aa bbb*" of *EXAMPLE*. Although the frequency of *space* over the entire message is greater than that of *b*, at this point *b* has higher frequency and therefore is mapped to the shorter 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 is 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], whereas others exploit *locality of reference* [Bentley et al. 1986]. Locality of reference is the tendency,

Source message	Probability	Codeword
a	2/6	10
b	3/6	0
space	1/6	11

Figure 4. A dynamic Huffman code table for the prefix "*aa bbb*" of message *EXAMPLE*.

common in a wide variety of text types, for a particular word to occur frequently for short periods of time and 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. Thus, as long as the encoding and decoding times of an adaptive method are not substantially greater than those of a static method, the fact that an initial scan is not needed implies a speed improvement in the adaptive case. In addition, the mapping determined in the first pass of a static coding scheme must be transmitted by the encoder to the decoder. The mapping may preface each transmission (i.e., each file sent), or a single mapping may be agreed upon and used for multiple transmissions. In one-pass methods the encoder defines and redefines the mapping dynamically during transmission. The decoder must define and redefine the mapping in sympathy, in essence "learning" the mapping as codewords are received. Adaptive methods are discussed in Sections 4 and 5.

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 previously agreed upon codes and inform the receiver of the choice (by transmitting first the "name" or number of the chosen code). Hybrid methods are discussed further in Sections 2 and 3.2.

1.3 A Data Compression Model

In order to discuss the relative merits of data compression techniques, a framework

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.