SPRINGER PROFESSIONAL COMPUTING

A Guide to DATA COMPRESSION METHODS

David Salomon

Springer

CD-ROM INCLUDED

A Guide to

DATA COMPRESSION METHODS

David Salomon

With 92 Illustrations

Includes a CD-ROM

Springer

David Salomon
Department of Computer Science
California State University, Northridge
Northridge, CA 91330-8281
USA
david.salomon@csun.edu

Cover Illustration: "Abstract: Yellow, Blue", 1993, Patricia S. Brown/Superstock.

Library of Congress Cataloging-in-Publication Data
Salomon, D. (David), 1938–
A guide to data compression methods/David Salomon.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-95260-8 (pbk.: alk. paper)
1. Data compression (Computer science). I. Title.
QA76.9D33 S28 2001
005.74'6—dc21 2001-032844

Printed on acid-free paper.

Production managed by Frank McGuckin; manufacturing supervised by Erica Bresler.
Camera-ready copy prepared from the author's TeX files.
Printed and bound by Hamilton Printing Co., Rensselaer, NY.
Printed in the United States of America.

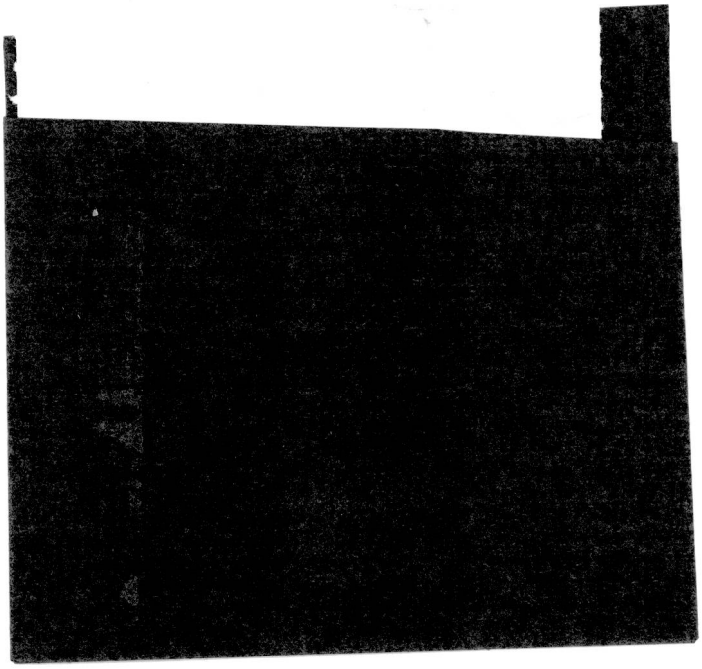
9 8 7 6 5 4 3 2 1

ISBN 0-387-95260-8 SPIN 10796580

Springer-Verlag New York Berlin Heidelberg
A member of BertelsmannSpringer Science+Business Media GmbH

A Guide to

DATA COMPRESSION METHODS



Springer
New York
Berlin
Heidelberg
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

To the data compression community;
visionaries, researchers, and implementors.

Preface

In 1829, Louis Braille, a young organist in a Paris church, blind since age 3, invented the well-known code for the blind, still in common use today all over the world and named after him. Braille himself modified his code in 1834, and there have been several modifications since. However, the basic design of this code, where each character is represented by a group of 3×2 dots, has remained intact. The dots are embossed on thick paper and each can be raised or flat (i.e., present or absent). Each dot is therefore equivalent to one bit of information. As a result, the Braille code (Figure 1) is a 6-bit code and can therefore represent 64 symbols (the code of six flat dots indicates a blank space).

Braille's followers extended the capabilities of his code in several ways. One important extension is contractions. These are letters that, when they stand alone, mean words. For example, the letter "b" standing alone (or with punctuation) means the word "but," the letter "e" standing alone means "every," and "p" means "people." Another extension is short-form words. These are combinations of two or more codes that mean an entire word (short-form words may contain contractions). For example, "ab" means "about," "rcv" means "receive," and "(the)mvs" means "themselves." (The "the" in parentheses is a contraction, dots 2-3-4-6.) Figure 2 shows some examples of these special codes.

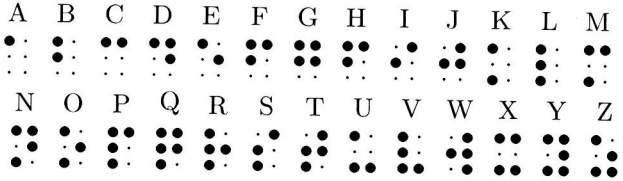


Figure 1: The 26 Braille letters

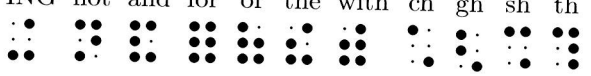


Figure 2: Some contractions and short words in Braille

The contractions, short words, and other extensions of the Braille code are examples of *intuitive data compression*. Those who developed the Braille code further and modified it for various languages realized that certain words and letter combinations are common and should be assigned special, short codes to facilitate rapid reading. The idea that common data items should be assigned short codes is one of the principles of the modern field of data compression.

A Brief History of Braille

Louis Braille was born on 4 January, 1809, at Coupvray, near Paris. An accident at age 3 deprived him of his sight and he remained blind for the rest of his life. At age 10, he was sent to the Paris Blind School where he learned to read in a code of raised dots. This code was originally developed by M. Charles Barbier and later adopted by the military, which called it "night writing" and used it for soldiers to communicate after dark. Night writing was based on a twelve-dot cell, two dots wide by six dots high. Each dot or combination of dots within the cell stood for a letter or a phonetic sound. The problem with the military code was that the human fingertip could not feel all the dots with one touch.

Braille spent nine years developing and refining night writing, eventually ending up with the system of raised dots that today bears his name. His crucial improvement was to reduce the cell size from 6×2 to 3×2 dots. This meant that a fingertip could enclose the entire cell with one impression and advance fast from one cell to the next.

The Braille code was introduced to the United States in about 1860 and was taught with some success at the St. Louis School for the Blind. In 1868, the British and Foreign Blind Associations were founded. They introduced Braille into England and promoted it by printing and disseminating books in Braille.

In North America, the Braille organization is Braille Authority of North America (BANA), located at `http://www.brailleauthority.org/index.html`.

BANA's purpose is to promote and to facilitate the uses, teaching, and production of braille. It publishes rules and interprets and renders opinions pertaining to Braille in all existing and future codes.

The predecessor of this volume, *Data Compression: The Complete Reference*, was published in 1977, with a second edition published in late 2000. It was the immediate and enthusiastic readers' response that encouraged me to write this slim volume. Whereas the original book is large, attempting to cover both the principles of data compression and the details of many specific methods, this book is less ambitious. It aims to guide a lay reader through the field of compression by conveying the general flavor of this field. It does so by presenting the main approaches to compression and describing a few of the important algorithms. The book contains little mathematics, has no exercises, and includes simple examples.

The Introduction explains why data can be compressed, presents simple examples, and discusses the main technical terms of the field.

Chapter 1 discusses the statistical approach to data compression. This approach is based on estimating the probabilities of the elementary symbols in the data to be compressed and assigning them codes of varying sizes according to their probabilities.

The elementary symbols can be bits, ASCII codes, bytes, pixels, audio samples, or anything else. The main concept treated in this chapter is variable-size (prefix) codes. The methods described are Huffman coding, facsimile compression, and arithmetic coding.

The popular technique of dictionary compression is the topic of Chapter 2. A dictionary-based compression method saves bits and pieces of the file being compressed in a data structure called a dictionary. The dictionary is searched for each new fragment of data to be compressed. If that fragment is found, a pointer to the dictionary is written on the compressed file. The following compression methods are described in this chapter: LZ77, LZSS, LZ78, and LZW.

Images are common in computer applications, and image compression is especially important because an image can be large. Chapter 3 is devoted to image compression. Most of the chapter discusses various approaches to this problem, such as run-length encoding, context probability, pixel prediction, and image transforms. The only specific methods described are JPEG and JPEG-LS.

Chapter 4 is devoted to the wavelet transform. This technique is becoming more and more important in image, video, and audio compression. It is mathematically demanding, and a simple, nonmathematical presentation of its principles presents a challenge to both author and reader. The chapter starts with an intuitive technique based on the calculation of averages and differences. It then relates this technique to the Haar wavelet transform. The concept of filter banks is introduced next, followed by the discrete wavelet transform. The only wavelet-based specific compression method illustrated in this chapter is SPIHT.

A movie is, in some sense, a generalization of a single still picture. Movies are quickly becoming popular in computer multimedia applications, a trend that has created a demand for video compression. A movie file tends to be much bigger than a single image, so efficient video compression is a practical necessity. Another factor in video compression is the need for simple, fast decompression, so that a compressed video can be decompressed in real time. Chapter 5 covers the principles of video compression.

The last chapter, Chapter 6, examines the topic of audio compression. Sound is one of the "media" included in computer multimedia applications and is therefore very popular with computer users. Sound has to be digitized before it can be stored and used in a computer, and the resulting audio files tend to be large. The chapter presents the basic operation of the MP3 audio compression method (actually, this is the audio part of MPEG-1) and also includes a short introduction to sound, the properties of the human auditory system, and audio sampling.

The book is intended for those interested in a basic understanding of the important field of data compression but do not have the time or the technical background required to follow the details of the many different compression algorithms. It is my hope that the light use of mathematics will attract the lay reader and open up the "mysteries" of data compression to the nonexpert.

The CD-ROM included with the book is readable by PC and Macintosh computers. For each platform, the CD contains popular compression programs (freeware and shareware) and a catalog file listing the programs. In addition, there is one file with verbatim listings of the various code fragments (in Mathematica and Matlab) found in the book.

Domain name `BooksByDavidSalomon.com` has been registered and will always point to any future location of the book's Web site. The author's present email address is `david.salomon@csun.edu`, but some readers may find it easier to use the redirection address ⟨*anyname*⟩`@BooksByDavidSalomon.com`.

Readers willing to put up with eight seconds of advertisement can be redirected to the book's web site from `http://welcome.to/data.compression`. Email sent to `data.compression@welcome.to` will also be redirected.

Those interested in data compression in general should consult the short section titled "Joining the Data Compression Community" at the end of the book, as well as the useful URLs `http://www.internz.com/compression-pointers.html` and `http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`.

Northridge, California — David Salomon

Math is hard.
—Barbie

Non mi legga chi non e matematico
(Let no one read me who is not a mathematician.)
—Leonardo da Vinci

Colophon

Most of this book was culled from the second edition of *Data Compression: The Complete Reference* during winter 2001. As its predecessors, this book was designed by the author and was typeset by him with the TEX typesetting system developed by D. Knuth. The text and tables were done with Textures, a commercial TEX implementation for the Macintosh. The diagrams were done with Adobe Illustrator, also on the Macintosh. Diagrams that require calculations were done either with *Mathematica* or Matlab, but even those were "polished" in Adobe Illustrator. The following points illustrate the amount of work that went into the book:

- The book contains about 128,100 words, consisting of about 744,500 characters. As is now so common with any technical text, much reference material, including some *Mathematica* and Matlab codes, were obtained from the World Wide Web.
- The text is typeset mainly in font cmr10, but about 30 other fonts were used.
- The raw index file contained about 1350 items.
- There are about 310 cross references in the book.

That which shrinks must first expand.
—Lao-Tzu, verse 36 of *Tao Te Ching*

Contents

Thus a rigidly chronological series of letters would present a patchwork of subjects, each of which would be difficult to follow. The Table of Contents will show in what way I have attempted to avoid this result.

—Charles Darwin, *Life and Letters of Charles Darwin*

Introduction

Those who use compression software are familiar with terms such as "zip," "implode," "stuffit," "diet," and "squeeze." These are names of programs or methods for compressing data, names chosen to imply compression. However, such names do not reflect the true nature of data compression. Compressing data is not done by stuffing or squeezing it, but by removing any *redundancy* that's present in the data. The concept of redundancy is central to data compression. Data with redundancy can be compressed. Data without any redundancy cannot be compressed, period.

We all know what information is. We intuitively understand it but we consider it a qualitative concept. Information seems to be one of those entities that cannot be quantified and dealt with rigorously. There is, however, a mathematical field called *information theory*, where information is handled quantitatively. Among its other achievements, information theory shows how to precisely define redundancy. Here, we try to understand this concept intuitively by pointing out the redundancy in two common types of computer data and trying to understand why redundant data is used in the first place.

The first type of data is text. Text is an important example of computer data. Many computer applications, such as word processing and software compilation, are nonnumeric; they deal with data whose elementary components are characters of text. The computer can store and process only binary information (zeros and ones), so each character of text must be assigned a binary code. Present-day computers use the ASCII code (pronounced "ass-key," short for "American Standard Code for Information Interchange"), although more and more computers use the new Unicode. ASCII is a fixed-size code where each character is assigned an 8-bit code (the code itself occupies seven of the eight bits, and the eighth bit is parity, designed to increase the reliability of the code). A fixed-size code is a natural choice because it makes it easy for software applications to handle characters of text. On the other hand, a fixed-size code is inherently redundant.

In a file of random text, we expect each character to occur approximately the same number of times. However, files used in practice are rarely random. They contain meaningful text, and we know from experience that in typical English text certain letters, such as "E," "T," and "A" are common, whereas other letters, such as "Z" and

"Q," are rare. This explains why the ASCII code is redundant and also points the way to eliminating the redundancy. ASCII is redundant because it assigns to each character, common or rare, the same number (eight) of bits. Removing the redundancy can be done by assigning variable-size codes to the characters, with short codes assigned to the common characters and long codes assigned to the rare ones. This is precisely how Huffman coding (Section 1.4) works.

Imagine two text files A and B with the same text, where A uses ASCII codes and B has variable-size codes. We expect B to be smaller than A and we say that A has been *compressed* to B. It is obvious that the amount of compression depends on the redundancy of the particular text and on the particular variable-size codes used in file B. Text where certain characters are very common while others are very rare has much redundancy and will compress well if the variable-size codes are properly assigned. In such a file, the codes of the common characters should be very short, while those of the rare characters can be long. The long codes would not degrade the compression because they would rarely appear in B. Most of B would consist of the short codes. Random text, on the other hand, does not benefit from replacing ASCII with variable-size codes, because the compression achieved by the short codes is cancelled out by the long codes. This is a special case of a general rule that says that random data cannot be compressed because it has no redundancy.

The second type of common computer data is digital images. A digital image is a rectangular array of colored dots, called *pixels*. Each pixel is represented in the computer by its color code. (In the remainder of this section, the term "pixel" is used for the pixel's color code.) In order to simplify the software applications that handle images, the pixels are all the same size. The size of a pixel depends on the number of colors in the image, and this number is normally a power of 2. If there are 2^k colors in an image, then each pixel is a k-bit number.

There are two types of redundancy in a digital image. The first type is similar to redundancy in text. In any particular image, certain colors may dominate, while others may be infrequent. This redundancy can be removed by assigning variable-size codes to the pixels, as is done with text. The other type of redundancy is much more important and is the result of *pixel correlation*. As our eyes move along the image from pixel to pixel, we find that in most cases, adjacent pixels have similar colors. Imagine an image containing blue sky, white clouds, brown mountains, and green trees. As long as we look at a mountain, adjacent pixels tend to be similar; all or almost all of them are shades of brown. Similarly, adjacent pixels in the sky are shades of blue. It is only on the horizon, where mountain meets sky, that adjacent pixels may have very different colors. The individual pixels are therefore not completely independent, and we say that neighboring pixels in an image tend to be *correlated*. This type of redundancy can be exploited in many ways, as described in Chapter 3.

Regardless of the method used to compress an image, the effectiveness of the compression depends on the amount of redundancy in the image. One extreme case is a uniform image. Such an image has maximum redundancy because adjacent pixels are identical. Obviously, such an image is not interesting and is rarely, if ever, used in practice. However, it will compress very well under any image compression method. The other extreme example is an image with uncorrelated pixels. All adjacent pixels

in such an image are very different, so the image redundancy is zero. Such an image will not compress, regardless of the compression method used. However, such an image tends to look like a random jumble of dots and is therefore uninteresting. We rarely need to keep and manipulate such an image, so we rarely need to compress it. Also, a truly random image features small or zero correlation between pixels.

What with all the ARC war flames going around, and arguments about which program is best, I decided to do something about it and write my OWN.
You've heard of crunching, jamming, squeezing, squashing, packing, crushing, imploding, etc....
Now there's TRASHING.
TRASH compresses a file to the smallest size possible: 0 bytes! NOTHING compresses a file better than TRASH! Date/time stamp are not affected, and since the file is zero bytes long, it doesn't even take up any space on your hard disk!
And TRASH is FAST! Files can be TRASHED in microseconds! In fact, it takes longer to go through the various parameter screens than it does to trash the file!
This prerelease version of TRASH is yours to keep and evaluate. I would recommend backing up any files you intend to TRASH first, though....
The next version of TRASH will have graphics and take wildcards:
`TRASH C:\PAYROLL\*.*`
...and will even work on entire drives:
`TRASH D:`
...or be first on your block to trash your system ON PURPOSE!
`TRASH ALL`
We're even hoping to come up with a way to RECOVER TRASHed files!

From *FIDO News*, 23 April 1990

The following simple argument illustrates the essence of the statement "Data compression is achieved by reducing or removing redundancy in the data." The argument shows that most data files cannot be compressed, no matter what compression method is used. This seems strange at first because we compress our data files all the time. The point is that most files cannot be compressed because they are random or close to random and therefore have no redundancy. The (relatively) few files that can be compressed are the ones that we *want* to compress; they are the files we use all the time. They have redundancy, are nonrandom and therefore useful and interesting.

Given two different files A and B that are compressed to files C and D, respectively, it is clear that C and D must be different. If they were identical, there would be no way to decompress them and get back file A or file B.

Suppose that a file of size n bits is given and we want to compress it efficiently. Any compression method that can compress this file to, say, 10 bits would be welcome. Even compressing it to 11 bits or 12 bits would be great. We therefore (somewhat arbitrarily) assume that compressing such a file to half its size or better is considered good compression. There are 2^n n-bit files and they would have to be compressed into 2^n different files of sizes less than or equal $n/2$. However, the total number of these files

is

$$N = 1 + 2 + 4 + \cdots + 2^{n/2} = 2^{1+n/2} - 1 \approx 2^{1+n/2},$$

so only N of the 2^n original files have a chance of being compressed efficiently. The problem is that N is much smaller than 2^n. Here are two examples of the ratio between these two numbers.

For $n = 100$ (files with just 100 bits), the total number of files is 2^{100} and the number of files that can be compressed efficiently is 2^{51}. The ratio of these numbers is the ridiculously small fraction $2^{-49} \approx 1.78 \cdot 10^{-15}$.

For $n = 1000$ (files with just 1000 bits, about 125 bytes), the total number of files is 2^{1000} and the number of files that can be compressed efficiently is 2^{501}. The ratio of these numbers is the incredibly small fraction $2^{-499} \approx 9.82 \cdot 10^{-91}$.

Most files of interest are at least some thousands of bytes long. For such files, the percentage of files that can be efficiently compressed is so small that it cannot be computed with floating-point numbers even on a supercomputer (the result is zero).

It is therefore clear that no compression method can hope to compress all files or even a significant percentage of them. In order to compress a data file, the compression algorithm has to examine the data, find redundancies in it, and try to remove them. Since the redundancies in data depend on the type of data (text, images, sound, etc.), any compression method has to be developed for a specific type of data and works best on this type. There is no such thing as a universal, efficient data compression algorithm.

The rest of this introduction covers important technical terms used in the field of data compression.

- The *compressor* or *encoder* is the program that compresses the raw data in the input file and creates an output file with compressed (low-redundancy) data. The *decompressor* or *decoder* converts in the opposite direction. Notice that the term *encoding* is very general and has wide meaning, but since we discuss only data compression, we use the name *encoder* to mean data compressor. The term *codec* is sometimes used to describe both the encoder and decoder. Similarly, the term *companding* is short for "compressing/expanding."

- A *nonadaptive* compression method is rigid and does not modify its operations, its parameters, or its tables in response to the particular data being compressed. Such a method is best used to compress data that is all of a single type. Examples are the Group 3 and Group 4 methods for facsimile compression (Section 1.6). They are specifically designed for facsimile compression and would do a poor job compressing any other data. In contrast, an *adaptive* method examines the raw data and modifies its operations and/or its parameters accordingly. An example is the adaptive Huffman method of Section 1.5. Some compression methods use a two-pass algorithm, where the first pass reads the input file to collect statistics on the data to be compressed, and the second pass does the actual compressing using parameters or codes set by the first pass. Such a method may be called *semiadaptive*. A data compression method can also be *locally adaptive*, meaning it adapts itself to local conditions in the input file and varies this adaptation as it moves from area to area in the input. An example is the move-to-front method [Salomon 2000].