LIBRARY OF CONGRESS

Office of Business Enterprises
Duplication Services Section

THIS IS TO CERTIFY that the collections of the Library of Congress contain a bound volume entitled **A GUIDE TO DATA COMPRESSION METHODS,** David Salomon, call number QA 76.9.D33 S28 2002, Copy 2. The attached - Title Page, Copyright Page, Table of Contents Pages and Introduction Pages - are a true and complete representation from that work.

IN WITNESS WHEREOF, the seal of the Library of Congress is affixed hereto on May 18, 2018.

Deirdre Scott
Business Enterprises Officer
Office of Business Enterprises
Library of Congress

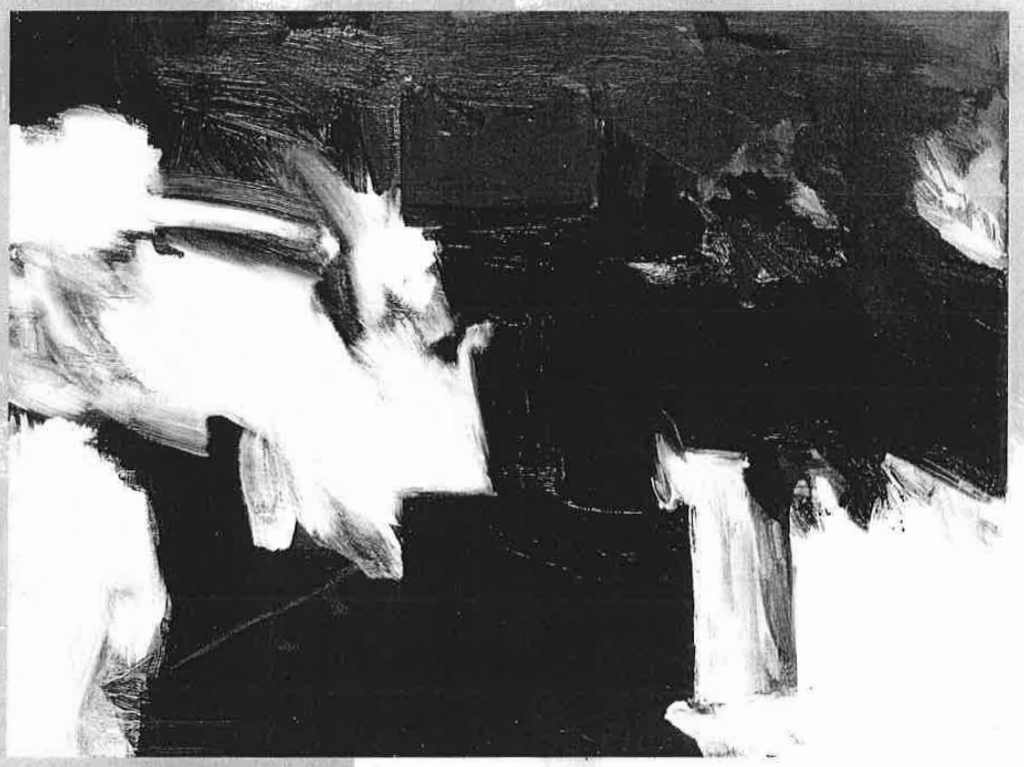
101 Independence Avenue, SE Washington, DC 20540–4917 Tel 202.707.5650 www.loc.gov; duplicationservices@loc.gov

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*

# 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].

- *Lossy/lossless compression:* Certain compression methods are lossy. They achieve better compression by losing some information. When the compressed file is decompressed, the result is not identical to the original data. Such a method makes sense when compressing images, movies, or sounds. If the loss of data is small, we may not be able to tell the difference. In contrast, text files, especially files containing computer programs, may become worthless if even one bit gets modified. Such files should be compressed only by a lossless compression method. [Two points should be mentioned regarding text files: (1) If a text file contains the source code of a program, many blank spaces can normally be eliminated, since they are disregarded by the compiler anyway. (2) When the output of a word processor is saved in a text file, the file may contain information about the different fonts used in the text. Such information may be discarded if the user wants to save just the text.]

- *Symmetric compression* is the case where the compressor and decompressor use the same basic algorithm but work in "opposite" directions. Such a method makes sense for general work, where the same number of files are compressed as are decompressed. In an asymmetric compression method, either the compressor or the decompressor may have to work significantly harder. Such methods have their uses and are not necessarily bad. A compression method where the compressor executes a slow, complex algorithm and the decompressor is simple is a natural choice when files are compressed into an archive, where they will be decompressed and used very often, such as mp3 audio files on a CD. The opposite case is useful in environments where files are updated all the time and backups are made. There is only a small chance that a backup file will be used, so the decompressor isn't used very often.

- *Compression performance:* Several quantities are commonly used to express the performance of a compression method.

1. The *compression ratio* is defined as

$$\text{Compression ratio} = \frac{\text{size of the output file}}{\text{size of the input file}}.$$

A value of 0.6 means that the data occupies 60% of its original size after compression. Values greater than 1 indicate an output file bigger than the input file (negative compression). The compression ratio can also be called bpb (bit per bit), since it equals the number of bits in the compressed file needed, on average, to compress one bit in the input file. In image compression, the similar term bpp stands for "bits per pixel." In modern, efficient text compression methods, it makes sense to talk about bpc (bits per character), the number of bits it takes, on average, to compress one character in the input file.

Two more terms should be mentioned in connection with the compression ratio. The term *bitrate* (or "bit rate") is a general term for bpb and bpc. Thus, the main goal of data compression is to represent any given data at low bit rates. The term *bit budget* refers to the functions of the individual bits in the compressed file. Imagine a compressed file where 90% of the bits are variable-size codes of certain symbols and the remaining 10% are used to encode certain tables that are needed by the decompressor. The bit budget for the tables is 10% in this case.

2. The inverse of the compression ratio is the *compression factor*:

$$\text{Compression factor} = \frac{\text{size of the input file}}{\text{size of the output file}}.$$

In this case values greater than 1 indicate compression, and values less than 1 imply expansion. This measure seems natural to many people, since the bigger the factor, the better the compression. This measure is distantly related to the sparseness ratio, a performance measure discussed in Section 4.1.2.
3. The expression $100 \times (1 - \text{compression ratio})$ is also a reasonable measure of compression performance. A value of 60 means that the output file occupies 40% of its original size (or that the compression has resulted in savings of 60%).
4. In image compression, the quantity bpp (bits per pixel) is commonly used. It equals the number of bits needed, on average, to compress one pixel of the image. This quantity should always be compared with the bpp before compression.

- The *probability model*. This concept is important in statistical data compression methods. Sometimes, a compression algorithm consists of two parts, a probability model and the compressor itself. Before the next data item (bit, byte, pixel, or anything else) can be compressed, the model is invoked and is asked to estimate the probability of the data item. The item and the probability are then sent to the compressor, which uses the estimated probability to compress the item. The better the probability, the better the item is compressed.

  Here is an example of a simple model for a black and white image. Each pixel in such an image is a single bit. Assume that after reading 1000 pixels and compressing them, pixel 1001 is read. What is the probability that this pixel is black? The model can simply count the numbers of black and white pixels read so far. If 350 of the 1000 pixels were black, then the model can assign pixel 1001 a probability of $350/1000 = 0.35$ of being black. The probability and the pixel (which may, of course be either black or white) are then sent to the compressor. The point is that the decompressor can easily calculate the same probabilities before it decodes the 1001st pixel.

- The term *alphabet* refers to the set of symbols in the data being compressed. An alphabet may consist of the two bits 0 and 1, of the 128 ASCII characters, of the 256 possible 8-bit bytes, or of any other symbols.

- The performance of any compression method is limited. No method can compress all data files efficiently. Imagine applying a sophisticated compression algorithm $X$ to a data file $A$ and obtaining a compressed file of just 1 bit! Any method that can compress an entire file to a single bit is excellent. Even if the compressed file is 2 bits long, we still consider it to be highly compressed, and the same is true for compressed files of 3, 4, 5 and even more bits. It seems reasonable to define efficient compression as compressing a file to at most half its original size. Thus, we may be satisfied (even happy) if we discover a method $X$ that compresses any data file $A$ to a file $B$ of size less than or equal half the size of $A$.

  The point is that different files $A$ should be compressed to different files $B$, since otherwise decompression is impossible. If method $X$ compresses two files $C$ and $D$ to the

same file $E$, then the $X$ decompressor cannot decompress $E$. Once this is clear, it is easy to show that method $X$ cannot compress all files efficiently. If file $A$ is $n$ bits long, then there are $2^n$ different files $A$. At the same time, the total number of all files $B$ whose size is less than or equal half the size of $A$ is $2^{1+n/2}-2$. For $n=100$, the number of $A$ files is $N_A = 2^{100} \approx 1.27 \cdot 10^{30}$ but the number of $B$ files is only $N_B = 2^{1+50}-2 \approx 2.25 \cdot 10^{15}$. The ratio $N_B/N_A$ is the incredibly small fraction $1.77 \cdot 10^{-15}$. For larger values of $n$ this ratio is much smaller.

The conclusion is that method $X$ can compress efficiently just a small fraction of all the files $A$. The great majority of files $A$ cannot be compressed efficiently since they are random or close to random.

This sad conclusion, however, should not cause any reader to close the book with a sigh and turn to more promising pursuits. The interesting fact is that out of the $2^n$ files $A$, the ones we actually *want* to compress are normally the ones that compress well. The ones that compress badly are normally random or close to random and are therefore uninteresting, unimportant, and not candidates for compression, transmission, or storage.

The days just prior to marriage are like a
snappy introduction to a tedious book.
—Wilson Mizner