Data Compression

Data compression is any process by which a digital (e.g. electronic) file may be transformed to another ("compressed") file, such that the original file may be fully recovered from the original file with no loss of actual information. This process may be useful if one wants to save storage space: for example if one wants to store a 3MB file, it may be preferable to first compress it to a smaller size. Also compressed files are much more easily exchanged over the internet since they upload and download much faster. We require, however, the ability to reconstitute the original file from the compressed version at any time.

This process is different than what often happens in 'compressing' a digital photograph, say, where significant reduction in file size is possible by sacrificing image resolution: a high-resolution 2MB digital image may be transformed to a 50KB image which is more appropriate to put on your website (since 2MB files take a long time for visitors to your website to download) but the 50KB version is coarser or grainier than the original image, and the original version cannot be recovered from the 50KB version. We are not considering this type of 'lossy compression'; it is understood that here we are not tolerating any degradation of data.

It must also be observed that *not every* file can be compressed to a smaller size (without loss of actual information). Otherwise we could compress a 3MB file to 1.2MB (say), then compress it again to 350KB (say), then compress it again to 70KB (say), etc., repeating this process until the resulting file is only 1 byte in size. Clearly we cannot expect every 3MB file to be compressible in this way since there are only $2^8=256$ different possible 1-byte files, but a much larger number of 3MB files (actually $2^{1073741824}$ such files, a number of more-than-astronomical proportions!). Typically text files are highly compressible; binary executable files are somewhat compressible; and audio files or digital images are rather incompressible since they are already compressed.

If a 3MB file can be compressed to a 1.2MB file with no information loss, why would we have any further need for the 3MB file? If the 3MB file is plain text (for example your email correspondence for the year 2003) then it would not be readable in compressed form. Or if the 3MB file were a binary executable file, it would not be recognizable in compressed form by your computer's operating system. So even if a 1.2MB file contains, *in principle*, all the information of your original 3MB file, it will usually be necessary to recover the original file (at least temporarily) from the compressed version, before the information can be accessed or used. And it will be necessary to have efficient algorithms for both the compression and decompression of data.

Having read my argument above (about why it is ludicrous to expect *every* file to be compressible), you might well wonder how it is that file compression is possible at all. Upon close inspection, however, you will see that my argument really only shows why *not all* files are compressible. It is still reasonable to expect that many everyday files are compressible using a compression utility such as WinZip®, and others are not. The files that are more successfully compressible are those with a high degree of repetition or other observable pattern. If, on the other hand, WinZip is applied to a binary file having a highly random appearance with no evident pattern, the utility may actually return a "compressed" file that is larger than the original.

Consider, for example, a 1MB binary file consisting of a string of zeroes. Since there are $2^3=8$ bits in every byte, such a file may arise as a bitstring '000…0' of length $2^{20}\times 2^3=2^{23}=8388608$. The same information may be summarized in a much smaller file that simply tells WinZip "This is a file consisting of 8388608 bits, all of which are zeroes". Similarly, a 1MB binary file consisting of the bitstring '010101…01' can be summarized in a much smaller file that tells WinZip "This is a file consisting of 8388608 bits formed by the bitstring '01' repeated 4194304 times". Most large files will not be as highly compressible as these extreme examples, but all occurrences of pattern provide at least some opportunity for compression; for example, if the character string 'the', or some other sequence of symbols, is represented frequently in a file, this can be represented in an abbreviated form in the compressed file.

Ideally, we might hope for a 'perfect' compression algorithm that compresses 'most' everyday files as much as possible (recognizing that any such algorithm will actually enlarge some files, those files with no exploitable repetitions or patterns). Such an algorithm, we expect, would never be outperformed by any other compression algorithm. Surprisingly, *no such optimal algorithm exists!* What do I mean by this? Do I mean that no one has figured out yet how to design *the best* compression algorithm? *No!* I am telling you that no such algorithm is possible, even in principle. Worse than that, there cannot even exist an algorithm telling the smallest size to which a given file can be compressed. (Note that here we are presumably asking for less than the optimally compressed file, if we are asking only for the *size* of the optimally compressed file.) It is a mathematical theorem that no such algorithm can possibly exist. Just as trisecting an angle with a straightedge and compass is impossible (there is also a theorem to this effect).

The good news is that there are algorithms that do a reasonably good job of compressing many of our everyday computer files, especially text files. No such algorithm is optimal, but some are better than others, and those with better compression ratios tend to require more execution time.

We now proceed to explain how data compression is possible using Huffman encoding works. This will not only explain how practical data compression is possible, but also provide a foundation for understanding entropy as a measure of the actual 'information content' of an information source.

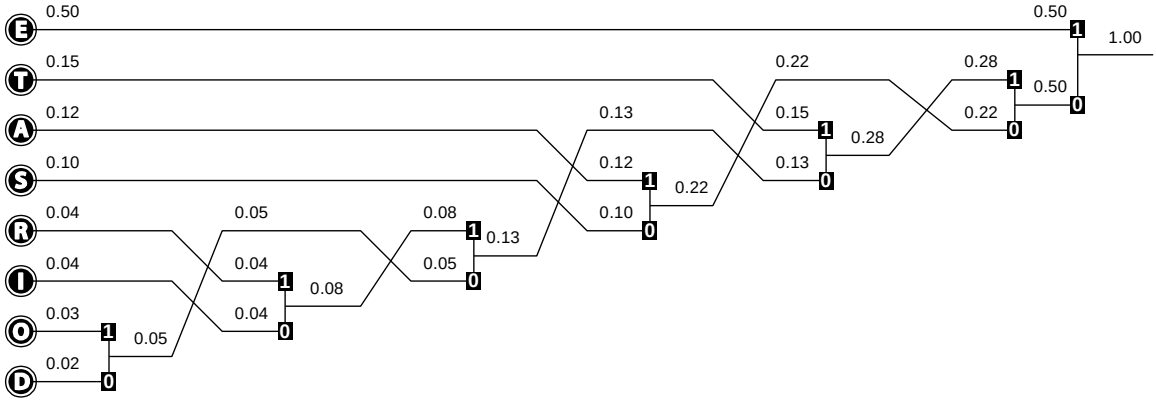
Huffman Encoding: Example 1

A file containing English language text, is a stream of characters (upper and lower case Roman letters, digits, punctuation marks, and other symbols such as '$', '%', etc.) with varying frequencies. Typically the blank symbol ' ' will be most frequent, followed by the letter 'e', etc. with symbols 'q', '#', appearing infrequently. Moreover certain combinations of characters will occur more frequently than others.

We will model this situation by adopting a simplified alphabet with only eight possible characters E, T, A, S, R, I, O, D occurring with frequencies as given by the following table:

character	E	T	A	S	R	I	O	D
frequency	0.50	0.15	0.12	0.10	0.04	0.04	0.03	0.02

In practice our file will typically be given in binary form; if the file is *n* characters in length, we may assume it is originally given as a binary file of size 3*n* bits, in which the letters E, T, …, D are represented by the eight bitstrings 000, 001, …, 111 respectively. We now show how to take advantage of the dominance of the letter 'E' to compress a large file with this character distribution, to a file of length about 2.8333*n* bits on average (6.6% shorter than the original file). This compression is achieved using an encoding based on the following '*tree*' or '*trellis*':



The trellis is constructed as follows: We first list the eight characters on the left, in decreasing order of frequency from top to bottom. A horizontal line extends to the right of each character, labeled by the corresponding frequency. We check that these frequencies add up to 1.00. We then join the bottom two characters, adding together their frequencies: 0.02 + 0.03 = 0.05. Since this is no longer the lowest frequency in the list, it 'bubbles up' above the lines corresponding to 'R' and 'I' (having frequencies 0.04 and 0.04). We now have only seven horizontal lines instead of eight, and the frequencies labeling these lines are again listed in decreasing order from top to bottom. Again merge the bottom two lines to obtain a single line with combined frequency 0.04 + 0.04 = 0.08, and 'bubble' this up above the previous line to restore the decreasing order of

frequencies. Repeat until all horizontal lines have merged to one line. At each stage, moving from left to right, we check that the frequencies add up to 1.00; in particular the last horizontal line on the right is labeled by the total frequency 1.00.

Every time two horizontal lines merge, moving from left to right, we label the upper branch '1' and the lower branch '0'. For each of the eight characters in turn, the path from left to right starting at that character passes through a sequence of 0's and 1's, and we *reverse* this string to obtain the Huffman codeword for that character. This gives the following table of Huffman codewords for each of the eight characters:

character	Standard codeword	Huffman codeword
E	000	1
T	001	011
A	010	001
S	011	000
R	100	01011
I	101	01010
O	110	01001
D	111	01000

Thus for example, the message text 'STEER' would be encoded as the 13-bit string 0000111101011 using Huffman encoding, compared with the 15-bit string 011001000000100 using the standard encoding. The encoding algorithm assigns shorter codewords to the more frequent characters, in order to reduce the average length of the encoded file. This feature explains why a Huffman encoded file is usually shorter than the original.

Decoding of the compressed text involves reading the trellis from right to left; each time is encountered, the next bit in the encoded text tells us which branch to follow. For example to decode the compressed text '0000111101011', start from the right side of the trellis and follow the branches labeled 0, 0, 0 to arrive at 'S'. Then start again from the right side of the trellis and follow the branches labeled 0, 1, 1 to arrive at 'T' on the left. Continue in this way until the original text 'STEER' is recovered.

Note that Huffman encoding is only successful in the absence of bit errors. For example if the compressed text '0000111101011' is altered to '1000111101011' as a result of only the first bit being switched, this will be misinterpreted as 'ESEEEER'. A *single* bit error results in *multiple* errors in the decoding process! Even the *length* of the decoded message comes out wrong in this case.

Note that the most frequent character 'E' is encoded as a single bit (fewer than the three bits required by the standard encoding), whereas the least frequent characters require 5 bits each (more than the three bits required by the standard encoding). The average number of bits required by a single character is

$$0.50\times1 + 0.15\times3 + 0.12\times3 + 0.10\times3 + 0.04\times5 + 0.04\times5 + 0.03\times5 + 0.02\times5 = 2.26 \text{ bits},$$

compared with 3 bits per character used by the standard encoding. Thus a typical binary file of length 3*n* bits will compress to a file about 2.26*n* bits long, on average. Thus the compressed file is, on average, 2.26/3.00 = 75.3% as large as the original file. As expected, in the worst case the "compressed" file will be longer than the original (possibly as long as 5*n* bits); but this happens only for files consisting of characters 'R', 'I', 'O', 'D'; and such files are quite rare, according to our table of frequencies. Most files have lots of E's, and these reduce the size of the compressed file considerably.

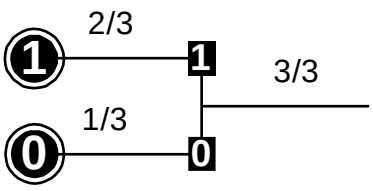
Morse code uses a similar principle of assigning shorter codewords to the more frequent letters, as the following table shows:

A	. —	**H**	. . . .	**O**	— — —	**V**	. . . —
B	— . . .	**I**	. .	**P**	. — — .	**W**	. — —
C	— . — .	**J**	. — — —	**Q**	— — . —	**X**	— . . —
D	— . .	**K**	— . —	**R**	. — .	**Y**	— . — —
E	.	**L**	. — . .	**S**	. . .	**Z**	— — . .
F	. . — .	**M**	— —	**T**	—		
G	— — .	**N**	— .	**U**	. . —		

Huffman Encoding: Example 2

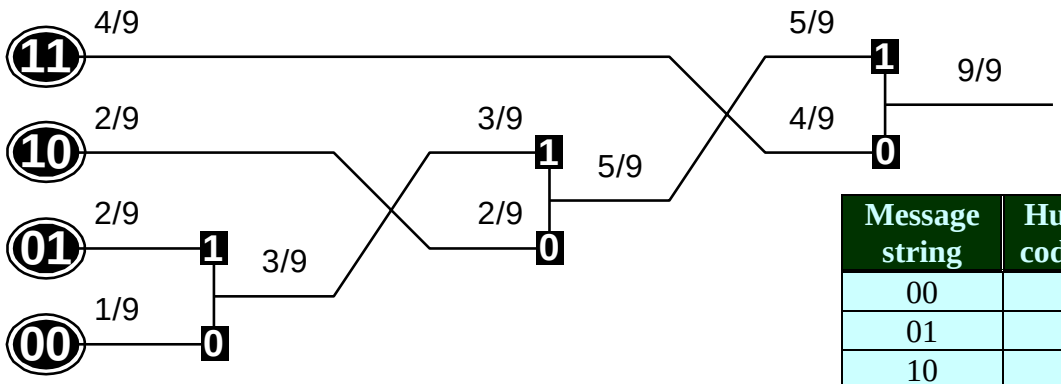
In Example 1 we considered an encoding of text, one character at a time. We now wish to demonstrate how a better compression ratio is typically achievable by encoding strings of characters. For this example we will simplify our alphabet even further. Let us now assume that the binary alphabet {0,1} is used, with twice as many 1's as 0's. We also assume that bits are independent of each other. That is, each position in the text is either 0 or 1, and the bit '1' occurs with probability 2/3, independently of the other bits in the text. (Later we will see how the assumption that bits are independent, affects the encoding algorithm and the compression ratio.

If characters (i.e. bits) are considered only one at a time as in Example 1, then no compression occurs: the "compressed" file is identical to the original, as we see from the following trellis and table of codewords:



Message string	Huffman codeword
0	0
1	1

The assumption that bits are independent random variables means that the bitstrings 00, 01, 10 and 11 occur in the original file with frequencies 1/9, 2/9, 2/9 and 4/9 respectively. Therefore if we divide the original file into bitstrings of length 2, and apply Huffman encoding to each of these pairs of bits, we obtain the trellis and table of codewords:



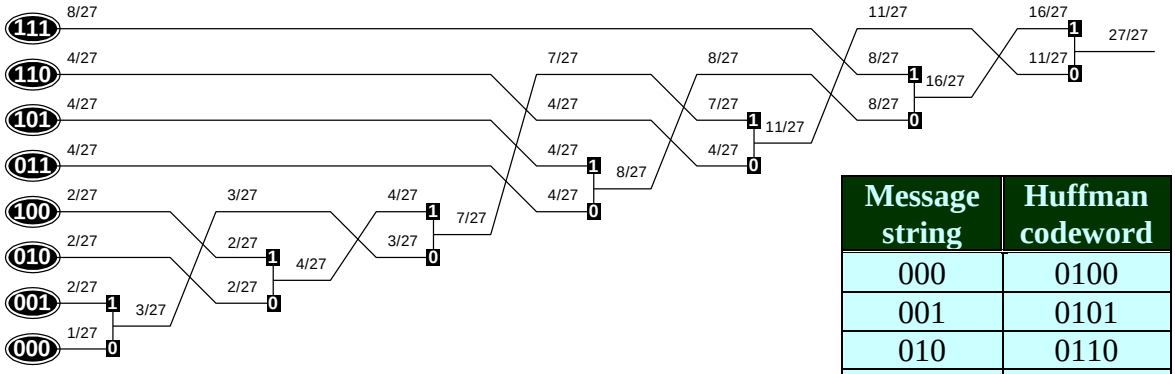
Message string	Huffman codeword
00	110
01	111
10	01
11	0

In this case the average number of bits required to encode each pair of message bits is

$$(1/9)\times 3 + (2/9)\times 3 + (2/9)\times 2 + (4/9)\times 1 = 17/9 \text{ bits.}$$

This algorithm will compress a file of n bits to a file of $(17/18)\times n = 0.9444n$ bits, on average. For example the input string '10111011' of length 8 will be compressed to the string '010010' of length six. Note that if the original file does not contain an even number of bits, then last bit must be treated differently than the previous bits; but this can be done without reducing the average compression ratio for large files.

We can do even better by grouping message bits together three at a time. This leads to the following trellis and table of codeword equivalents:



Message string	Huffman codeword
000	0100
001	0101
010	0110
011	100
100	0111
101	101
110	00
111	11

In this case every triple of input bits is encoded as

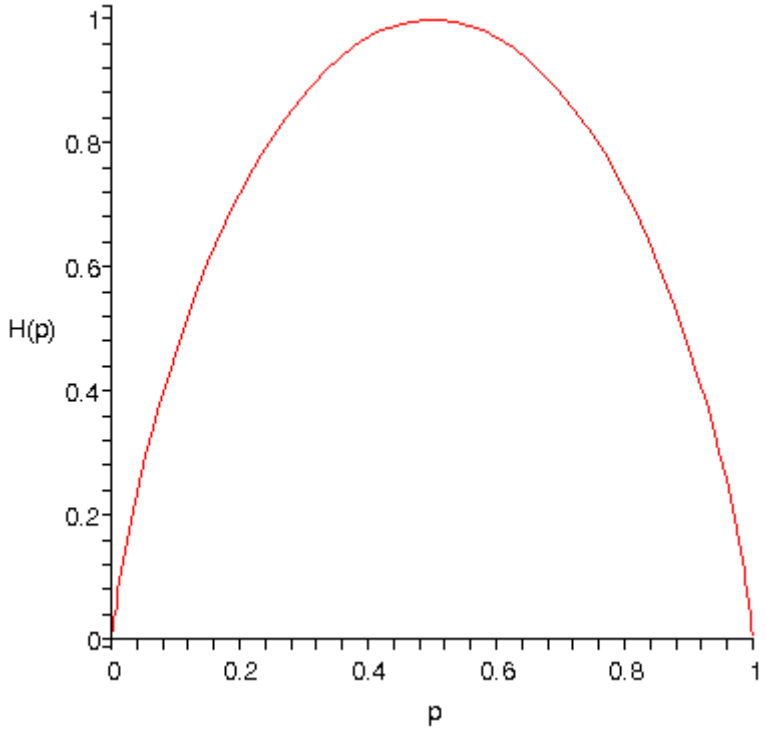
$$(1/27) \times 4 + (2/27) \times 4 + (2/27) \times 4 + (2/27) \times 4 + (4/27) \times 3 + (4/27) \times 3 + (4/27) \times 2 + (8/27) \times 2$$

= 76/27 bits on average. This will compress an input file of n bits to a file of $(76/81) \times n = 0.9383n$ bits on average, a slight improvement over the compression rate available by considering the input bits two at a time.

We can achieve even better compression rates if we are willing to group together the input bits 4 at a time, 5 at a time, etc.; but there is a limit to how much better we can do. This limit (for data containing random 0's and 1's with frequencies 1/3 and 2/3 respectively, with bits in different positions independent of each other) means that an input file of n bits cannot be compressed to a file any smaller than about $0.9183n$ bits on average. The value 0.9183 to which I refer here is actually $H(1/3)$ where $H(p)$ is the *binary entropy function* defined by

$$H(p) = -p \log_2(p) - (1-p) \log_2(1-p)$$

whose graph is shown here:



The base 2 logarithm function appearing in this formula is defined as follows: we say that $\log_2(N) = k$ is the value of k such that $2^k = N$. The reason that base 2 is used in this context is that we are using two characters (0 and 1) to represent all information. There are 2^k different bitstrings of length k, so it is possible to send 2^k different messages using bitstrings of length k. Now suppose we are given a number N and a list of N different messages that we might want to send. We want to encode each possible message as a bitstring of some length k. How large must k be? We must solve $2^k = N$ for k. For example, suppose we want to send one of the 16 possible hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F encoded as a bitstring. How long must each bitstring be? In

this case $k = \log_2(16) = 4$ since $2^4 = 16$. Thus each hexadecimal digit can be encoded as a bitstring of length 4 (0=0000, 1=0001, …, F=1111).

One interpretation of the function $H(p)$ is as follows: consider a file consisting of n random bits (with bits in different positions independent of each other), with 0's and 1's occurring with frequency p and $1–p$, respectively. The smallest file to which we may compress this data will be a file of size about $H(p)\times n$ bits, on average. Accordingly, we view the actual information content of an n-bit file as being $H(p)\times n$ bits.

Let's think carefully what this means in the special cases $p = 0, 0.5, 1$. If $p = 0$ then the frequency of the bit '0' is zero, so the input file consists of a long string of 1's. As mentioned earlier, such data is highly compressible; it carries essentially no information. This is what we expect from the graph, where we clearly see that $H(p)$ goes to zero when p goes to zero. The same reasoning applies when p = 1: in this case the bit '0' occurs with frequency 1, so the input file consists of a long string of 0's. Once again the data is highly compressible; it carries essentially no information. Again, the graph shows $H(p)$ going to zero when p approaches 1. The case $p = 0.5$ means that the data consists of 0's and 1's occurring in equal numbers. This type of data has the most random appearance of all, and is incompressible. From the graph we see that $H(0.5) = 1$, so for an input file of length n bits, the "compressed" file will also have $H(0.5)\times n = n$ bits.

Notice that the graph of $H(p)$ is symmetric about the line $p = 0.5$. This means that $H(1–p) = H(p)$. We interpret this as saying that interchanging 0's and 1's in the data (i.e. replacing every 0 by 1, and every 1 by 0) does not change the actual information content of the data. It does mean that 0's and 1's will occur with frequency $1–p$ and p (respectively), instead of p and $1–p$ as before. But clearly the choice of two-letter alphabet is arbitrary: {0,1} is no better than {+,–} or {N,S} or {a,b} or {1,0}.

The entropy function $H(p)$ is also used in physics as a measure of randomness. We highlight here the fact that *information is measured in the same way as randomness*. Evidently data with a high degree of pattern or repetition (i.e. very little randomness), being the most compressible, carries very little information. Conversely, data with very little evidence of pattern or repetition (i.e. having the appearance of randomness), being the least compressible, carries a high degree of information. It was Claude Shannon whose insight, during the 1940's and 1950's, led to the quantification of information in this way: the possibility of measuring information much as we measure temperature or volume or electrical current. Of course we are not quantifying the subjective value of information: this measure will not distinguish between text from a Dr. Seuss book and a presidential State of the Union address, even though the former may contain much more information than the latter.