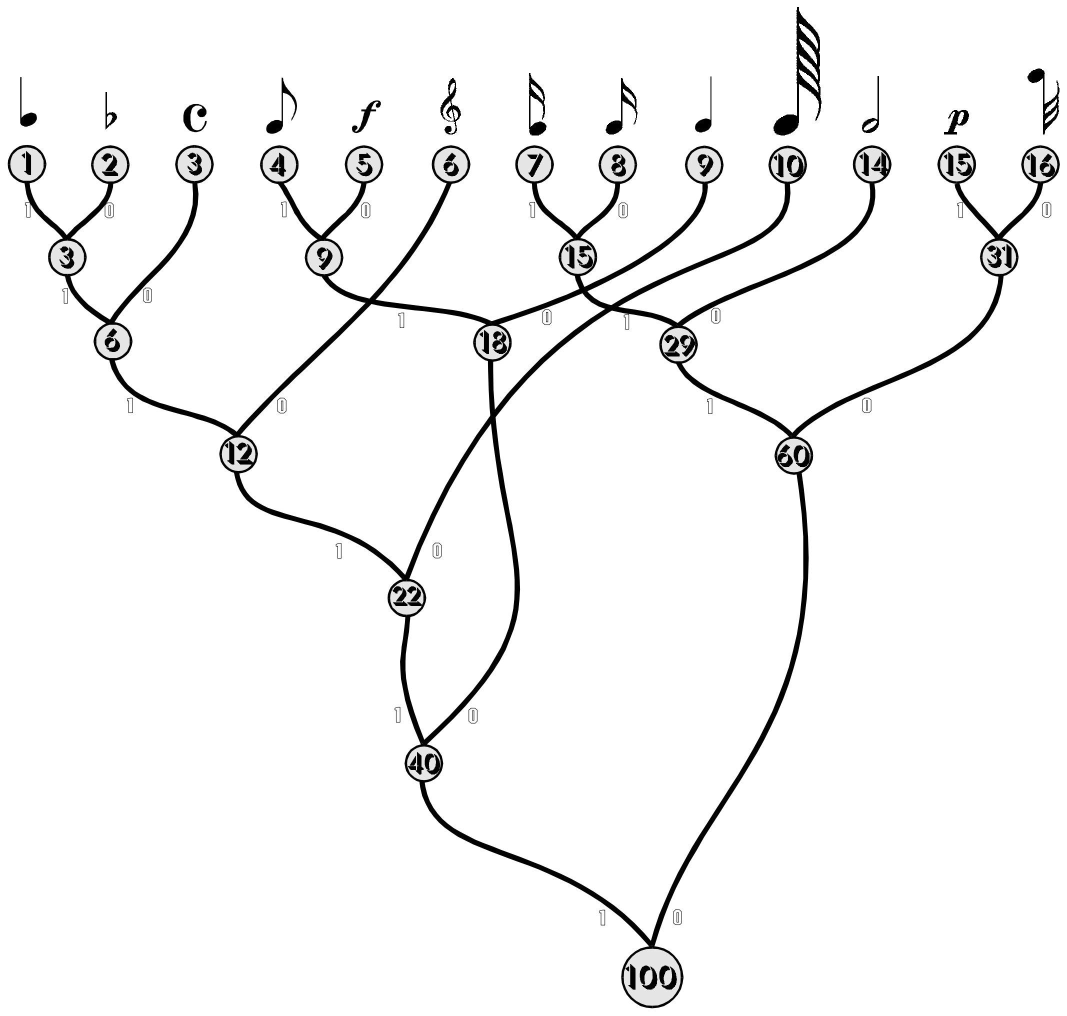
Word Vices



Information Theory, Memory, and Minimalism

Information Theory poses elegant riddles: how best code messages so that they are as short as possible? Also, how transmit them so that errors caused by noise are detected and corrected swiftly and accurately? In these days of increasing data transmission, there are dollars to be made with elegant answers to such questions. The shorter the phone call, the smaller the bill. Also, some compression programs allow users to store approximately twice as much data on their computers as they would otherwise be able to. This saves having to buy more memory. The utility, then, of data compression algorithms is two-fold: monetary savings that result from less time and space in transmitting and storing data.

But beyond the utility of compression algorithms lies the inherent attraction and elegance of interesting ways to concentrate messages. The minimalist drive to zero has informed much twentieth century art. Borges once wrote a story in which an ancient poet, asked by his king for poetry, wrote an epic. The king approved, had a special understanding of his work (unusual for kings), and asked for more. The poet obliged. The result was shorter but not less amazing. Of course, the king urged him on to more. The poet returned aged beyond the few years that had passed. Yes, he had composed something, he told the king, but did not think it wise to speak it aloud to one and all. He took the king into confidence and told him the one line he had composed. The king had rewarded the poet handsomely for the previous poems and was no less generous on this occasion. Too bad it was a golden dagger: the poet killed himself shortly thereafter and, as the story ends, the king is a beggar wandering the streets. Having heard an ultimate one-liner.

The ultimate (and otherworldly) data compression algorithm reduces every file to no size because the algorithm is so omnisciently predictive that there is absolutely no new or unexpected thing under the sun. God does not require compression algorithms: s/he/you/me/we/they can reconstruct the entire message from necessity; events convey no information to God; the amount of information in a message is proportional to its unpredictableness. If you know that a file consists of a string of zeroes and ones that record the outcomes of a million consecutive flips of a two-headed coin and 'heads' means '1', you don't need the file to reproduce it. The file itself is long but the information content is low.

However, as we know, we are unknowing; the sun is new with each day. So even the most ingenious compression algorithms currently shrink the size of files typically by only a factor of 1.5 to 5. We will look at two noteworthy compression algorithms and see how they work. The algorithms themselves are elegant in a manner that befits the task of concentration. Before examining the algorithms, it is useful to understand why contemporary computers understand only zeroes and ones. Knowing this explains why we will be coding messages into zeroes and ones.

Apparently neuro-physiology has not yet determined what the brain's basic data structures are. In computers, the basic unit of memory is currently a transistor-like device that can be charged with electricity or uncharged. It can also be inspected for the presence or the absence of a charge. If it's charged, then it represents a '1', let us say; if it is uncharged, it represents a '0'. These transistor-like devices are the basic units of memory in modern computers. We do not currently understand the (possibly) analogous basic units of memory in the brain.

Transistors can hold various levels of charge. However, the more levels of charge that must be measured, the more likely the measurement is to be in error. In the absence of measuring devices fine enough to reliably detect minute changes in charge in modern micro-chip transistors, computer architects have settled for just on or off. That is reliably measured. Obviously computer memory must be accurate or we get oranges and idiots. So computers operate in binary, i.e., a single cell of memory is capable of storing one bit of information, a '0' or a '1'.

Consequently, the text that I am typing into the computer is stored in a sequence of cells in the computer's electronic memory and each cell is either charged or not charged, a zero or a one, depending on whether we choose to view the situation physically or conceptually. Every bit of information in the memory of a computer is coded in zeroes and ones. It is important to appreciate this to begin to appreciate how computers work and how information is stored and transmitted over telecommunications lines.

Strings, Binary Encoding, and Huffman's Brillig

A string is a concatenation of symbols. There are two 1-bit strings: 0 and 1. There are four 2-bit strings: 00, 01, 10, and 11. There are eight 3-bit strings: 000, 001, 010, 011, 100, 101, 110, and 111. There are 2^n n-bit strings. The English alphabet consists of 52 characters (upper and lower case), so 6-bit strings would be required to encode the entire alphabet in binary. The computer keyboard has about 250 different characters in its alphabet. Two hundred fifty-six different characters can be represented in 8-bit strings. And that is partly why a byte, quite a crucial measure of information, is 8 bits long. Computers are language machines, and a byte is just right for a letter.

Suppose the keyboard had only five buttons on it. None of them are Shift or Ctrl or Alt keys. Boring keyboard. It only has five characters on it. Suppose the keys had the following symbols on them:

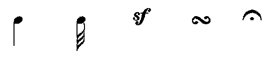
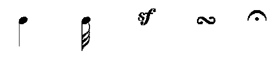


Figure 1

Suppose also that the language you were typing had the following probability distribution:



.01 .02 .03 .03 .91

Figure 2

So happens 1% of the time, happens 2% of the time, and so on. Though the probability distributions of characters in many types of files are relatively stable (such as in English files) there are obviously different types of computer files besides English ones.

One way to code the letters in binary is as follows:

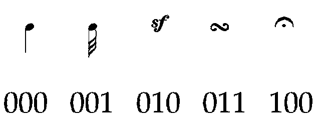


Figure 3

Each time you tap a key, three bits are stored in a file in your computer or sent over the line somewhere else. Consequently, the message 𝄾𝆍𝆗𝄐𝆗 is coded as 000010011100011. In general, if a message consists of n symbols, it will be coded in 3n bits. Since every character is coded in 3 bits, the average code length is 3 bits. The average code length of a coding into binary is the average number of bits per letter involved in the coding. One of the main concerns of Information Theory is reducing the average code length. It is somewhat surprising that we can do better. But that is part of the appeal of Information Theory.

When David A. Huffman was studying electrical engineering at MIT in 1951, Robert M. Fano, one of the pioneers of Information Theory, gave the students a choice between writing a final exam or a term paper. The term paper's subject was to be a description of the most efficient way to code any message into binary. Fano did not tell his students the problem was unsolved. Into the end of the term Huffman had a flash of insight that he has called "the most singular moment of my life."[1] There was the absolute lightning of sudden realization. He devised an elegant means of coding that minimizes the average code length when messages are coded into their single characters. The method is known as the Huffman algorithm.

Morse code is compressed coding. Some characters are coded to be shorter than other characters. The code for 'e' is shorter than the code for 'z'. 'e' is more probable than is 'z'. If symbol *a* is more probable than symbol *b*, then the code for *a* should be no longer than the code for *b*. And shorter, if possible.

Shown in Figure 4 are the steps involved in the Huffman encoding of strokes on our keyboard that are tapped upon in a bizarre language. Its heavy on the 𝄐. The particular musical symbols were chosen because they are five interesting symbols in a row in the character map of the Musical Symbols typeface on my computer.

What Huffman realized in his flash of brillig was the method for constructing codes pictured in Figure 4 and that this method minimizes the average code length.

The Huffman Algorithm

The Huffman algorithm begins by placing each probability in a node (a circle; see step 1 of Figure 4). Probabilities in the diagrams are given as percentages. Nodes that have no nodes attached below them are called root nodes. So all five nodes in step 1 of Figure 4 are root nodes. Then we select two minimal probabilities contained in root nodes. We then create a new node that contains the sum of the two minimal root nodes. Then we label the left edge with a 1 and the right edge with a 0. We continue in this fashion until the Huffman tree is constructed.

[1] Stix, Gary, "Encoding the "Neatness" of Ones and Zeroes, PROFILE: DAVID A. HUFFMAN," *Scientific American*, September, 1991, p. 54.

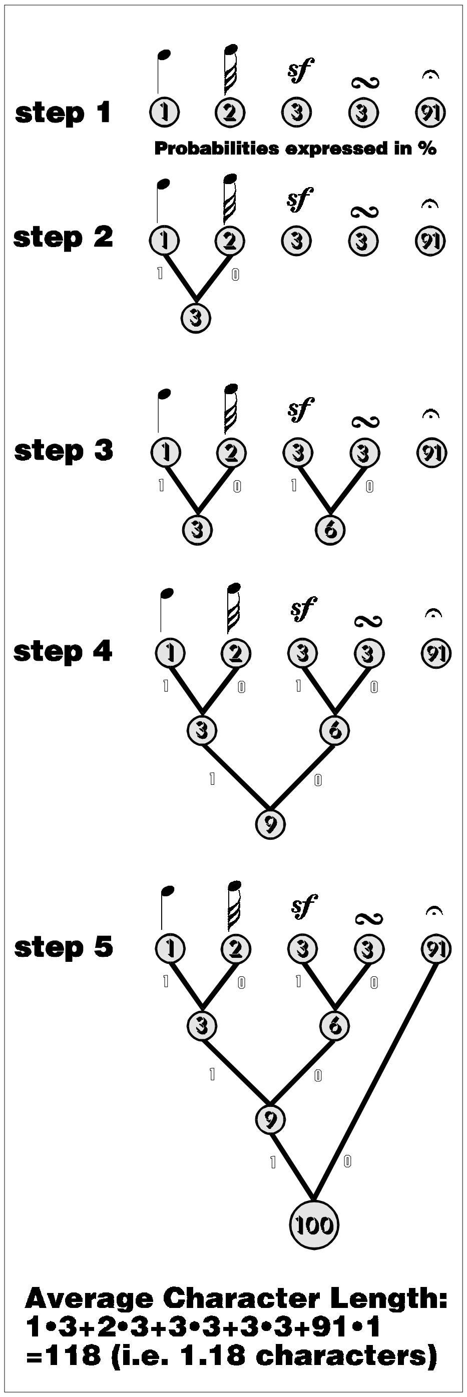

step 1
Probabilities expressed in %
step 2
step 3
step 4
step 5
Average Character Length:
1•3+2•3+3•3+3•3+91•1
=118 (i.e. 1.18 characters)

Figure 4

Notice that in step 2 there are three minimal root node probabilities. Each of the three have a value of 3%. It turns out that we are free to choose whichever two among the three we like. A hint as to why later. Figure 6 shows two Huffman trees that result from different choices being made at step 2. Different encodings of our musical alphabet result from different choices in the breaking of ties, but the average code length remains the same.

Having constructed a Huffman tree, we can then code a message. The binary code for a particular character may be read by traversing the Huffman tree from the root node to the node containing the character's probability. Figure 5 gives the Huffman codes that result from the construction in Figure 4.

Symbol:	𝅘𝅥𝅮	𝅘𝅥𝅱	*sf*	∾	𝄐
Probability:	.01	.02	.03	.03	.91
Huffman code:	111	110	101	100	0
Code Length:	3	3	3	3	1

Figure 5

The message *sf* ∾ 𝄐 would be coded as 1011000 according to Figure 5. This is slightly better than the encoding for *sf*∾𝄐 that results from applying the coding in Figure 3. That coding is 010011100.

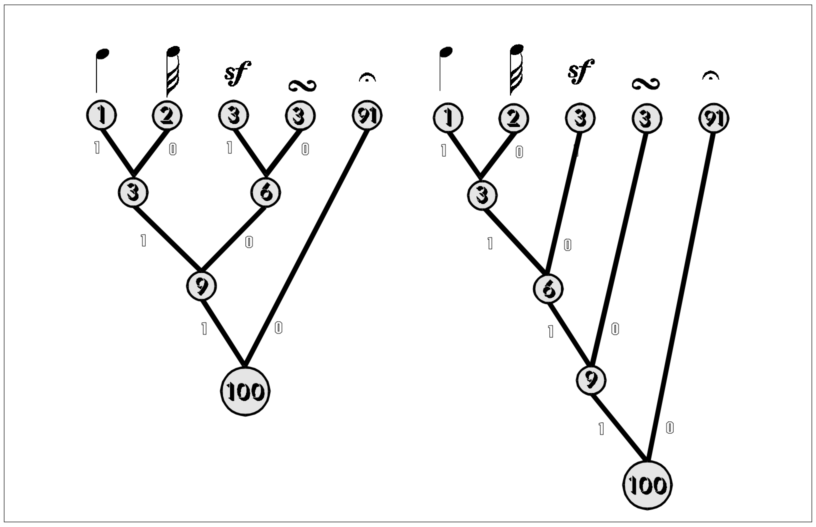


Figure 6
Two different, valid Huffman trees.

Weighted Averages

How do we calculate the average code length of the encoding described in Figures 4 and 5? We use a weighted average. To illustrate the idea of a weighted average, consider a game flipping coins. We flip one coin. If it's a fair coin, then the probability of heads is .5 and the probability of tails is .5. If it's heads, then you lose 50 cents—or gain -50 cents. If it's tails, you gain 25 cents. So in the long run, expect to gain 5 (-50) + .5 (25) = -25 cents per turn.

In general, suppose there are n possible events $e_1, e_2, e_3, \ldots, e_n$ and probabilities $p_1, p_2, p_3, \ldots, p_n$ such that the probability of event e_1 is equal to p_1 and the probability of e_2 is equal to p_2, and so on. Mathematicians say this by stipulating that the probability of event e_i is equal to p_i when i is equal to or between 1 and n. Even more concisely, '$P(e_i)=p_i$ when $1 \leq i \leq n$.' '$P(e_i)$' is read as 'the probability of event e_i.' Suppose also that $p_1+p_2+p_3+\ldots+p_n=1$ and that there is a weight (or payoff or code-length) associated with each event. In the case of the game, the events are whether the coin lands heads or tails. Each of these two events has a probability of .5. The weights assigned to the events are the pay-offs associated with the events. Let us denote the weights as $w_1, w_2, w_3, \ldots, w_n$. Then the average weight (or payoff or code-length) is given by

$$P(e_1)\, w_1 + P(e_2)\, w_2 + P(e_3)\, w_3 + \ldots + P(e_n)\, w_n$$

Figure 7

This quantity is sometimes called the expected weight. If we were to play a game flipping three coins at once and I win 50 cents if at least two heads come up and you win 40 cents otherwise, who expects to win? Figure 8 shows the possibilities. HTT, for instance, is the event that coin 1 turns up heads and coins 2 and 3 turn up tails.

8 possible outcomes:	HHH	HHT	HTT	HTH	TTH	TTT	THT	THH
Probability:	.125	.125	.125	.125	.125	.125	.125	.125
You win:	-50	-50	40	-50	40	40	40	-50
Weighted average:	.125(-50)+	.125(-50)+	.125(40)+	.125(-50)+	.125(40)+	.125(40)+	.125(40)+	.125(-50)=-5

Figure 8

So you would expect, in the long run, to lose about 5 cents per throw of the coins. The weighted average is a useful concept used widely in mathematics, computer science, gambling, and other activities.

Now that you understand the idea of a weighted average, you will understand that the average code length of the Huffman encoding of Figure 4 is given by

$$(.01)3 + (.02)3 + (.03)3 + (.03)3 + (.91)1 = 1.18$$

Figure 9

This average code length is a big improvement on the coding in Figure 2. It can be proved that you can do no better than 1.18 if you code only the individual characters rather than strings of characters. We will see later how the Ziv-Lempel algorithm codes increasingly long strings of characters.

Huffman Trees and Their Properties

Step 1 of Figure 4 contains 5 trees. Step 2 contains 4 trees, step 3 contains 3 trees, and step 5 contains 1 tree. The end of each branch contains 1 leaf node, and each branch extends from the root node to the leaf node. Nodes that are not leaf nodes are called internal nodes.

An interesting property of Huffman trees is that the average code length is the sum of the internal nodes. To see why this is so, remember that the weights in Figure 7 are the lengths of the binary codes for letters. Consider the number of times a probability is counted in the internal nodes of a Huffman tree.

Understanding that the average code length is the sum of the internal nodes is necessary toward understanding that during the construction of a Huffman tree we are free to choose any two minimal roots.

Another property of Huffman trees is that the code for a particular letter may be read by traversing the tree from the root to the appropriate leaf. The code for a particular letter never terminates at an internal node. Consequently, Huffman codes have the prefix property: the code for a particular letter is never a prefix for the code of another letter. This ensures that any message is both uniquely decodable and will require relatively small memory resources for the decoding. Uniquely decodable messages are not ambiguous as long as the decoder knows the Huffman code, which is sent to the decoder before the message is sent.

Codings that do not have the prefix property are not necessarily ambiguous. For instance, if we consider a three letter alphabet coded as 1, 10, and 00, then the coding does not have the prefix property because 1 is a prefix of 10. Nonetheless, any message coded in 1, 10, and 00 is uniquely decodable. However, the decoding device will have to read to the end of 100000 and remember the entire message before it can uniquely decode 100000. So we can construct messages coded in 1, 10, and 00 that require an arbitrarily large amount of memory for their decoding. Codings that have the prefix property, on the other hand, require a relatively small amount of memory that is independent of the length of the string to be decoded.

A Ziv-Lempel Algorithm

There are several drawbacks to Huffman encoding. The algorithm requires two readings of the file: one to calculate the probability distribution and one to encode the file. The Ziv-Lempel algorithm, named for the people who invented it, reads the file only once. Another advantage of the Ziv-Lempel algorithm is that, unlike the Huffman algorithm, it is capable of using strings longer than one letter in its coding. For instance, it is possible that the Ziv-Lempel algorithm would code *the* simply as *1* whereas the Huffman algorithm would have a separate code for *t*, for *h*, and for *e*.

Figure 11 shows the encoding of the string *abcdabdc*. Λ (see Figure 11) is not a symbol from the file's alphabet but, rather, denotes the empty string. The idea of Λ is similar to the idea of $\varnothing$, the empty set. Λ and $\varnothing$ are not the same idea, however, because sets and strings are different types of entities. The string *ab* is different from the string *ba*, but $\{a, b\} = \{b, a\}$.

In the Ziv-Lempel algorithm, we begin with a row of memory cells numbered from 0. Each memory cell can hold two things: an integer and a letter/symbol from the alphabet or Λ, which is not a letter/symbol from the file's alphabet. The integer is always held at the top of a cell, and the letter or Λ is always held at the bottom of the cell. The cells are initially all empty except the 0th cell. The 0th cell, regardless of the string to be encoded, always holds 0 and Λ at the beginning of the Ziv-Lempel algorithm.

Then we read to the end of the longest prefix of the string/message/file that matches a previously read sub-string of the string/message/file. Call that sub-string α. The letter that follows α is also read. Call that letter/symbol β. The string $\alpha\beta$ is coded in the cells, as shown in Figure 10, where i is the number of the cell that stores α.

$$\begin{array}{|c|}\hline \mathrm{i} \\ \beta \\ \hline\end{array}$$

Figure 10

Let us see the algorithm code *abcdabdc*. See Figure 11.

The first *a* is read and compared with previously read strings. No strings have previously been read (stored in the cells) except Λ, so Λ is the longest prefix that has already been read. Λ is stored in the 0th cell. So 0 is stored in the top of the 1st cell. The top of a given cell is an integer that points to another cell. The arrows in Figure 11 are included simply as a reminder of this.

The last character read was an *a*. So it is stored in the bottom of the first cell.

Lemple–Ziv encoding of *abcdabdc*

Λ is the empty string

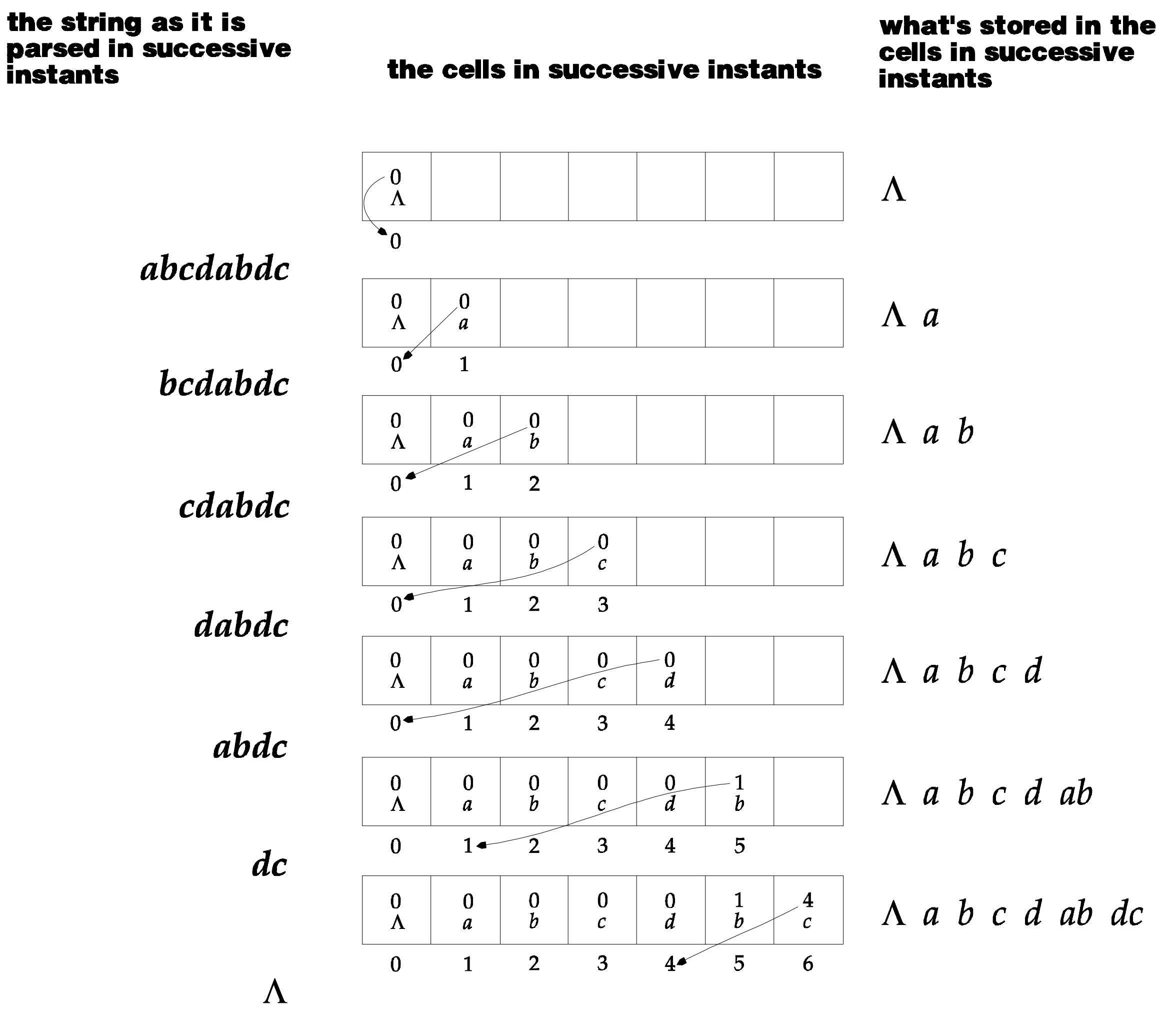

the string as it is parsed in successive instants
the cells in successive instants
what's stored in the cells in successive instants
abcdabdc
bcdabdc
cdabdc
dabdc
abdc
dc
Λ
Λ
Λ a
Λ a b
Λ a b c
Λ a b c d
Λ a b c d ab
Λ a b c d ab dc
0 1 2 3 4 5 6

Figure 11

What remains of the string to be coded is *bcdabdc*. So, again, the longest string, reading from *bcdabdc*, that matches a string already stored in the cells is *Λ*. So *b* is stored in the bottom of the 2nd cell and a pointer to the 0th cell is written into the top of the 2nd cell. This process is continued until the entire string has been read.

Comparing the Algorithms

Messages can be constructed that are better coded with the Ziv-Lempel algorithm. Messages can be constructed that are better coded with the Huffman algorithm. The advantage of Lempel-Ziv coding over Huffman coding is that Ziv-Lempel doesn't just code single letters/symbols, but larger strings. For instance, in Figure 11, *ab* and *dc* are basically treated as a single letter. This ability of the Ziv-Lempel algorithm to encode longer strings makes the coding very concise when the file that is being coded contains many multiple-symbol repetitions. Also, the file only need be read once from beginning to end, whereas the Huffman algorithm demands two readings of the file. So the Ziv-Lempel algorithm is usually faster. However, if the file being encoded is large enough to exceed the available number of random access memory (RAM) cells available to the compression program in the computer, then the Ziv-Lempel algorithm will have to use hard-drive memory also. This brings us to one of the disadvantages of the algorithm.

When coding a string, the algorithm needs to look up the codes for strings it has coded previously. This makes the language into which the algorithm codes messages *context sensitive*. When reading such languages, we (or machines) may have to remember the text that surrounds or precedes or goes after what we are reading in order that we might interpret the syntax correctly or acceptably. English and other human languages are context sensitive. Unless we are dealing with writing such as some of Joyce's or Faulkner's, however, where sentences can continue for sixty pages, we have only to remember as much information as is usually contained in sentences to read with syntactical comprehension. We are not generally required to remember each and every word in a piece of writing as we go along.

Computers used to do Ziv-Lempel encoding must have enough memory to store the entire coding as it proceeds (which becomes less likely as the size of the file increases). This is true for both the machine doing the encoding and the machine receiving the encoding. Of course, both machines must also have enough memory to store the decoded message. These days, with random access memory being cheap and relatively plentiful and hard-drive memory being even cheaper and more plentiful, memory shortages are of relatively little concern (sure, tell that to the judge). Individual files are rarely more than 4 million bytes long. And when they are often larger, the computing environment is usually rich enough to afford the appropriate machines that have enough random access memory to do the job.

Reading from RAM is quick because RAM is electronic circuitry. Reading from hard-drives, however, is about 1000 times slower because the computer must find the information on a disc. The speed is not speed-of-light stuff as it is when reading from RAM. Since the Ziv-Lempel algorithm requires that the cells in Figure 11 be available throughout the encoding (and decoding), the larger the file to be compressed, the more likely it is that some of the cells will have to be stored on the hard drive. Consequently, since the Ziv-Lempel algorithm is such that the cells are consulted often during coding and decoding, the compression will slow dramatically if some of the cells must be stored on the hard-drive.

The Huffman algorithm, on the other hand, does not make such demands on memory. A Huffman tree does not require much more memory than the alphabet requires. Notice that however big the file to be compressed is, the Huffman algorithm creates a Huffman tree that contains each symbol only once. So the Huffman algorithm demands an amount of working memory that is independent of the size of the file to be compressed.

Another potential disadvantage of the Ziv-Lempel algorithm arises out of the need to limit the longest string it can treat as a single letter. This must be done, again, because computers have limited memory. If the longest string that can be parsed off is not long enough then the algorithm may end up with a basic alphabet (composed of strings of various length—none of which exceed this limit) that is not useful later on in the file if the latter part of the file is totally different from the first part of the file where the algorithm trains itself.

However, Ziv-Lempel is the basis of programs such as *compress* and PKUNZIP and PKZIP, all three of which are used widely both commercially and by the general public. Huffman encoding, on the other hand, is not used much anymore at all. In situations where working memory is a crucial problem, such as when extremely large image-files are to be compressed (maps and satellite imagery) using computers located on satellites (you want as little hardware on a satellite as possible), Huffman encoding might be advantageous over Ziv-Lempel—if the time factor is not as crucial as the memory factor. It should be added, however, that neither is used in the above situation, but another coding method specific to visual image data encoding. It should also be mentioned that there is a variation on the Huffman algorithm that is dynamic: like the Ziv-Lempel algorithm, this algorithm only needs to read the file once. Its memory requirements are larger than the Huffman algorithms. It should also be mentioned that there are many variations on the basic Ziv-Lempel algorithm.

So the encoding method used depends on three factors: the speed demanded of the performance, the memory resources available, and the type of files one can (or can't) anticipate having to work with. These are considerations a software engineer will have to think about in a given situation. There is very rarely a one-and-only best way to write a program. A particular situation may have its one-and-only, but general situations like encoding any file are too general to supply a one-and-only solution. Usually the speed that's required and the memory resources available must be taken into account. As in the engineering of most things, the optimal solution doesn't drop out of a textbook, but requires trade-offs of various sorts between the materials at hand and the performance demanded of the thing to be engineered. Nonetheless, the Huffman and Ziv-Lempel algorithms are pretty solutions and serve to introduce the issues and techniques of data compression.

Down the Digital Road

Our electronic transmissions—increasingly—will be in binary over the next years. Data ranging from bills to telephone (and other) books to television images to music to business records of all sorts will be digitized and transmitted in compressed forms.

The more tightly compressed the file, the less time it will take to transmit, and so the less money it will cost to send it. There are many people dreaming up better compression techniques suited to all sorts of situations. Also, the better the compression, the less space a file takes up on the hard-drive. There are commercially available software packages that compress all your files. This is much cheaper than buying a new hard-drive that has twice the capacity of the old hard-drive.

This text was composed on a computer. I then generated a (Postscript) file that the printer at the University can read. I then compressed that file. Usually PKZIP shrinks Postscript files by a factor of about 4.5 if the original file is sufficiently long. So if the file is a million bytes big, PKZIP will shrink it to about 222,222 bytes. The file will then be transmitted over the phone line to the University, UNZIPped there, and sent to the printer. I often send files to the University that, when zipped, take an hour and a half to modem to the University. They would therefore take about six hours and forty-five minutes if they were not zipped up. The numbers will change over the years. But the basic lesson will remain the same.

My modem's transmission rate is 2400 baud, which means 2400 bits per second, i.e. 300 letters per second. The Internet, which is a world-wide computer network, does not use the telephone lines but, rather, fibre optics and microwave connections. I've compared the transmission rates. I often connect to the Internet and thence to Indiana or other sites around the world to get software and typefaces for my computer. Though I'm almost fonted out at this point. But not quite. If I find the software useful, I am honour-bound to send the authors the very reasonable amount of money they ask satisfied customers for. In any case, the files are transmitted along the Internet nodes from Indiana (or wherever) to the University in Victoria. Then I have to use my modem connection to get them from the University (which is local) to my house. The difference in speed is surprising. The fibre optics lines vary in speed depending on the volume of traffic travelling the data highways and the route your data travels, but I've observed that the files go from Indiana to Victoria anywhere from 16 to 300 times faster than they do from the University to my house.

The ingenuity involved in compression algorithms is a small portion of the thought that has gone into making the data highways that begin to resemble a nervous system, of sorts. Yet for all our ingenuity, we don't know the basic architecture of the brain. Given the vast amount of information stored in our heads, there are obviously better ways to code and compress information than we currently know about but perform routinely by virtue of our mysterious making.

Bibliography

Bell, Timothy C., Cleary, John G., Witten, Ian H., *Text Compression* (Englewood Cliffs, New Jersey, Prentice Hall, 1990).

Knuth, Donald E., *The Art of Computer Programming, Volume 1* (Reading, Massachusetts, Addison-Wesley Publishing Company, 1968).

Huffman, D.A., "A method for the construction of minimum-redundancy codes," *Proc. IRE 40*, 1952, pp.1098-1101.

Parker, D. Stott Jr., "Conditions for the optimality of the Huffman Algorithm," *SIAM J. of Computing* 9:3, 1980, pp. 470-489.

Cappellini, V., *Data Compression and Error Control Techniques With Applications* (London: Academic Press, 1985).

Lelewer, D.A., and Hirschberg, D.S., "Data Compression," *ACM Computing Surveys, Vol. 19, No. 3*, September 1987, pp 261-296.

Apiki, Steve, "Lossless Data Compression," *BYTE*, March 1991, p. 309.

Stix, Gary, "Encoding the "Neatness" of Ones and Zeroes, *Profile: David A. Huffman*," *Scientific American*, September 1991, p. 54.