Data Compression

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV is compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand.

There are many situations where you have redundancy in a body of information (image file, text file, video, etc.). We can adopt some form of coding that exploits the redundancy in order to reduce the space which the information occupies. This is the basic idea underlying approaches to data compression.

Suppose we have a message that consists of only four letters, A, B, C, D. To measure the information content of such a message, it is convenient to code the letters as binary digits, the process known as encoding, and count the total number of digits to give an estimate of the information content in bits.

If there is no redundancy in the sequence, so that the letters occur at random, then we can use two binary digits to code each letter. For example, 00 for A, 01 for B, 10 for C and 11 for D. The sequence such as:

BADDACDCADCCABBC

will be coded as:

01001111001011100011101000010110.

The information content will be two bits per letter.

Suppose instead that the occurrence of letters is not completely random, but is constrained in some way. For example, assume that A is followed by B or C with equal probability (but never by D), B is followed by C or D with equal probability, C is followed by D or A with equal probability and D is followed by A or B with equal probability. We can encode a sequence that satisfies these rules by using two binary digits to encode the first letter as above, and then using one binary digit to encode each successive letter. If the preceding letter is A, we can use 0 to encode the following letter if it is B and 1 to encode it if it is C, and so on. Then a sequence such as:

BDBCACDBCDBDBDACDA

will be coded as:

011101101011110100.

The information content will be n + 1 bits if there are n letters. If we had used the encoding that we used for a random sequence, it would be coded as:

0111011000101110111011101110011000101100

which is approximately twice as long. The redundancy in the second sequence has enabled us to reduce the number of binary digits used to represent it by half.
We distinguish between **lossless** data compression algorithms, which can reconstruct the original message exactly from the compressed message, and **lossy** algorithms, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters. Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but the “meaning” and clarity of the message might be fully maintained, or even improved.

Is there a lossless algorithm that can compress all messages? There has been at least one patent application that claimed to be able to compress all files (messages) – Patent 5,533,051 titled “Methods for Data Compression.” The patent application claimed that if it was applied recursively, a file could be reduced to almost nothing. With a little thought you should be able to convince yourself that this is not possible, at least if the source messages can contain any bit-sequence. Unfortunately, this patent was granted.

Simple coding techniques can be devised based on the frequency distribution of the characters. For example, one such coding technique is as follows: frequently occurring characters are assigned short codes and infrequently occurring characters are assigned long codes so that the average number of code symbols per character is minimized.

**Run-length code**

**Run-length coding** is a simple and effective way of compressing data in which it is frequently the case that the same character occurs many times in succession.

Run-length encoding is probably the simplest method of compression. It can be used to compress data made of any combination of symbols. It does not need to know the frequency of occurrence of symbols and can be very efficient if data is represented as 0s and 1s.

The general idea behind this method is to replace consecutive repeating occurrences of a symbol by one occurrence of the symbol followed by the number of occurrences. For example, the sequence **AABBBCCCCDCDDDBBBBBAAA** could be replaced by **A2B4C3D1C2D3B5A4**. This reduces the number of characters from 24 to 16. Protocols then need to be established to distinguish between characters and the counts in the compressed data. Below is another example of run-length encoding.
The method can be even more efficient if the data uses only two symbols (for example 0 and 1) in its bit pattern and one symbol is more frequent than the other. Below is the run-length encoding for two symbols, where 0 is more frequent than 1 and it alternates between 0 and 1.

Original:
0000000000110001111000000000000

Compressed:
1100 0010 0011 0100 1100

Run-length encoding performs lossless data compression and is well suited to palette-based images. It does not work very well on continuous-tone images such as photos, although JPEG uses it quite effectively on the coefficients that remain after transforming image blocks. Common applications of run-length encoded data include Truevision TGA, PackBits, PCX and ILBM. Run-length encoding is used in fax machines (combined with other techniques – modified Huffman coding). It is quite efficient because most faxed documents are mostly white space, with occasional interruptions of black.

Variable-length code

For the cases when there are no (or very few) repetitions, we need to use variable-length encoding. We use a proven fact that we need encoding where no character is a prefix of another character. A code generated in such manner is called a prefix-free code. We can represent our encoding using a binary tree.

For example, assume we have a data that uses a set of five different characters, and characters’ frequencies are given below:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
</tr>
</tbody>
</table>

Then, we can construct a binary tree for this data, where only leaves contain labels (characters), and we label each left edge with 0 and each right edge with 1 as shown in Figure 1. This is a binary tree generated by the Huffman algorithm that we discuss below.
Huffman codes

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm. David Huffman developed the algorithm as a student in a class on information theory at MIT in 1950. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities. The Huffman algorithm is very simple and is most easily described in terms of how it generates the prefix-code tree. The algorithm is as follows:

- Start with a forest of trees, one for each message. Each tree contains a single vertex with frequency $f_i$ (character and its frequency in the message)
- Repeat until only a single tree remains
  - Select two trees with the lowest frequency roots ($f_1$ and $f_2$)
  - Combine them into a single tree by adding a new root with frequency $f_1 + f_2$, and making the two trees its children. It does not matter which is the left or right child, but we can use a convention of putting the lower frequency root on the left if $f_1 \neq f_2$.

For example, the Huffman code for the example and the tree constructed in Figure 1 would be:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
<th>Code</th>
<th>Code length</th>
<th>Total length</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
<td>0</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>100</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>101</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>110</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>111</td>
<td>3</td>
<td>24</td>
</tr>
</tbody>
</table>

For a code of size $n$ this algorithm will require $n1$ steps since every complete binary tree with $n$ leaves has $n - 1$ internal nodes, and each step creates one internal node. If we use a priority queue with $\sim \log n$ time insertions and find-mins (e.g., a heap) the algorithm will run in $\sim n \log n$ time.

The key property of Huffman code algorithm is that it generates an optimal prefix code.