Maximum data compression

Understanding Huffman's encoding will enable more efficient programs

In this issue...

▶ WHAT'S COVERED

When we think of data, we usually just think of the information conveyed by that data: a customer list, an audio CD, a letter and so on. We don't usually think too much about the physical representation of that data itself: the program that manipulates that data takes care of that. But in reality we'd like to be able to store the same information in a smaller space and Huffman encoding was one of the first algorithms to do such a thing.

Let's consider this split in the nature of data: its information content versus its physical representation. Claude Shannon in the 1950s laid the foundations of information theory, including the notion that data can be represented by some minimal number of bits, called its entropy (a term taken from thermodynamics). He also realised that data is usually represented physically by more bits than its entropy suggests.

For a simple example, consider a probability experiment with a coin. We would like to toss the coin many times, build up a large table of results, and then do some statistical analysis on this large dataset to posit or prove some theorem. To build the dataset, we could record the results of each coin toss in several different ways: we could write down the word 'heads' or 'tails', we could write down the letter 'H' or 'T' or we could record a single bit (on or off, with on meaning tails, for example). Information theory would say that the result of each coin toss could be encoded in a single bit, so the final possibility I gave would be the most efficient in terms of space needed to encode the results. The first possibility is the most wasteful of space, taking five characters to record a single coin toss.

Think of it another way though: from the first example of recording the data to the last, we are storing the same results – the same information – in less and less space. In other words, we are compressing the data.

So, data compression is an algorithm for encoding information in a different way so that it occupies less space than before. We are removing redundancy; that is, getting rid of the bits from the physical representation of the data that aren't really required, to get at just the right number of bits that entropy would predict we should need for the information.

For the coin toss experiment, it was fairly easy to determine the best way of storing the dataset, but for other data, it gets more difficult and there are several algorithmic approaches we can take. One of the first approaches (Shannon also devised an algorithm for it, now known as the Shannon-Fano algorithm) was known as minimum redundancy coding. This is a method of encoding bytes (or, more formally, symbols) that occur more often with fewer bits than those that occur less often. For example, for text in the English language the letters E, T and A occur much more often than the letters Q, X and Z and so, in hand-waving terms, if we were able to code E, T and A with less than eight bits (assuming we're using ASCII as our encoding mechanism), and Q, X and Z with more, we would probably be able to store English text in fewer bits than plain ASCII would have done.

Binary trees

The most famous and successful minimum redundancy coding algorithm was devised by graduate student David Huffman in 1952 in a paper called *A Method for the Construction of Minimum-Redundancy Codes*. The reason for this fame is that Huffman's algorithm is mathematically guaranteed to produce the smallest encoding for each character in the original data.

Huffman's algorithm works by building a special kind of binary tree, called a prefix tree, where all the data (that is, byte values) are found on the leaves of the tree. First, we make a pass through the input data counting the number of times each byte value appears. We'll end up with a table of character counts. Figure 1 shows the character count table for the common phrase "she sells seashells by the seashore".

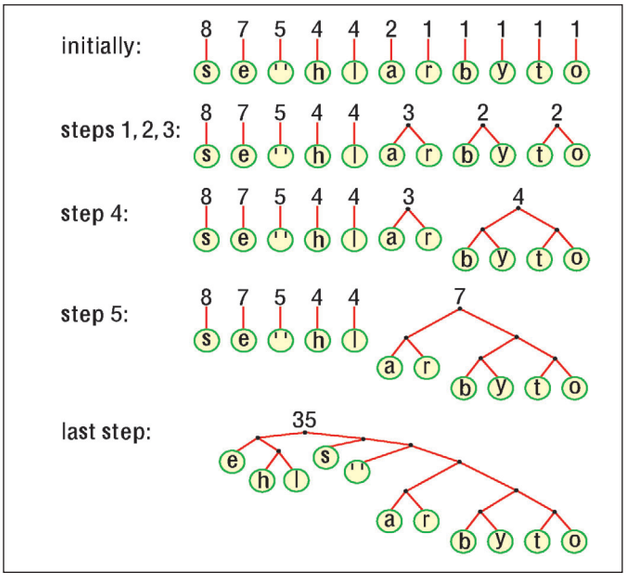
Space	5
a	2
b	1
e	7
h	4
l	4
o	1
r	1
s	8
t	1
y	1

▲ **Figure 1: Showing character counts for a sample phrase.**

Consider these character-and-count pairs as a 'pool' of nodes for the eventual Huffman tree. Remove the two nodes that have the smallest count from this pool. Join the two nodes to a new parent node, and set the parent's

Nugget

From the description of the algorithm for building a Huffman tree, you see that one of the processes we have to solve is the ability to quickly find the two nodes with the smallest count. For something other than the sample phrase there may be hundreds of nodes. The data structure that best solves this situation is the priority queue, a specialised efficient queue that releases items in priority order, highest first. We merely alter the queue so that it releases items in reverse order.



▲ **Figure 2: Building a Huffman tree as a family of nodes.**

count to the sum of its two children. Add the parent node back to the pool. We continue this process of removing two nodes, and adding back a single parent node, until the pool has only one node in it. At this point, we can remove the one node: it's the root of the Huffman tree.

Different nodes

This is a little hard to see, so let's create the Huffman tree for our sample phrase. Select two nodes that have the smallest count. We have ample nodes to choose from with count 1, but we'll select the b and y nodes. We create a parent node of count 2, and attach these two nodes as children. The parent goes back into the pool. Round the loop again, this time we'll select the t and o nodes, combine them as a mini-tree and put the parent node (of count 2 again) back into the pool. Round the loop again. This time we have a single node of count 1 (the r node), and a choice of three nodes of count 2 (the a node and the two parent nodes we've added previously). We choose the a node, join it to the r node, and add the new parent of count 3 to the pool again. We now select the two parent nodes of count 2, join them to a new parent of count 4, and add that back into the pool. Figure 2 shows the first few steps I did in building the Huffman tree and also the final tree. Note that many times there are several choices that could be made in selecting a couple of nodes since they have the same count. Don't worry, it doesn't matter since all possible Huffman trees that could be created are equal in encoding power.

All very well, but how does this Huffman tree help with encoding each character and compressing the phrase? Well, to encode the first s of the phrase, we find it in the tree, and trace its path from the root through to the leave where it's found. Every time we go left we encode a zero bit, every time we go right we encode a one bit. To encode another character, we start off at the root again and do the same thing. Figure 3 shows the table of encodings for each character, and using this table we can now calculate the complete encoding for the phrase. It starts with 10010001101000011011110 and there are 108 bits in all. If we assume that the original phrase was encoded in ASCII, one character per byte, the original phrase was 280 bits in length, and so our compression ratio is about 36 per cent.

To decode a Huffman-encoded bit stream, we use the same tree we built up in the compression phase. We start at the root and pick off a bit at a time from the compressed bit stream. If the bit is clear, we go left, if set, right. We continue like this until we reach a leaf, that is, a character and we output the character to the stream for the uncompressed data. At that point we start off from the root of the tree again with the next bit from the encoded data.

Notice that characters are only found on the leaves and hence the encoding for one character does not form the first part of the encoding for another. Because of this, we can't decode the compressed data incorrectly. (This property is why the tree is known as a prefix tree.)

Space	110
a	11100
b	111100
e	00
h	010
l	011
o	111111
r	11101
s	10
t	111110
y	111101

▲ **Figure 3: Character encodings for a sample phrase.**

Spotlight on... What about the tree?

The problem with Huffman encoding that we have managed to gloss over is the tree. Usually we wish to compress data to save space or to save on transmission times. The time and place we decode the data is usually far removed from the time and place we encoded it. However, the decoding algorithm requires the tree and so we must pass it on somehow.

We have two alternatives. The first one is to make the tree static, in other words, the same Huffman tree will be used to encode all data that we may see. This option is not optimal in the general case since our data tends to be dissimilar.

The second alternative is to attach the tree itself to the encoded bit stream, in some form or other. Of course, attaching the tree to the encoded data is bound to make the compression ratio worse, but there's nothing we can do about that. ■

Nugget

You may think that an algorithm devised in 1952 would be very outdated by now and supplanted by something better. In fact, the Huffman encoding algorithm is still in use in many different compression scenarios. For example, although the Deflate algorithm used in ZIP files uses a technique called dictionary compression to compress most data, it compresses the dictionary tables and so on with Huffman encoding (usually with static trees). Fax machines also use an algorithm with a static Huffman tree to compress the data prior to transmission. ■

We have a slight problem though: how do we recognise the end of the bit stream? When do we stop decoding? After all, underneath the covers, we will be packing bits 8 to a byte and writing the bytes out. It's unlikely that we will have an exact multiple of 8 bits in our bit stream and so there will be some bits 'left over'. There are two standard solutions to this dilemma, the first being to encode a special character not found in the original data and call it an end-of-file character (it will only appear at the end and so will have a count of 1), the second to record the length of the unencoded data to the compressed stream prior to compressing the data itself .

The first option seems bizarre; after all, if we're encoding a binary file, say, then the data might well consist of every single byte value from 0 to 255. For an ASCII text file, certainly we can imagine that there won't be any binary zeros in the text and so we can choose that as an end-of-file marker, but we can make no such assumptions about a binary file. The answer of course is to use 2-byte words instead of bytes to hold the byte values we see in the file and 'add' a special value, 256, to the list of values we'll be encoding. This value will only be used just once for a file, and so will be found in the longest path in the Huffman tree, but it's probable that its encoding will be still less than the 4 bytes you'd need to encode a length value. ■

Julian M Bucknall has worked for several IT companies and is now CTO for Developer Express.
feedback@pcplus.co.uk