Static Huffman Compression

Huffman Coding

What is most important in variable-length coding using an order-0 compression model are the *higher-frequency symbols* and we must have a way to maximize the assignment of smaller bits for their codes. And because some symbols are not that frequent, it is acceptable to represent them in longer bit codes.

Fortunately for us, a method already exists that does exactly this job. An *optimal* algorithm in assigning variable-length codewords for symbol probabilities (or *weights*) is the so-called **Huffman Coding**, named after the scientist who invented it, D. A. Huffman, in 1951. Huffman coding is guaranteed to produce "minimum redundancy codes" for *all* symbols using their frequency counts. It is used as a second-stage algorithm in the **ZIP** compression format as well as in the **MP3** codec.

The Huffman coding algorithm [1] is described as follows :

1. create a binary treenode for each symbol, and sort them according to frequency;
2. select the two *smallest-frequency* symbol nodes and create a new node and make the two nodes the "children" of this new node (after this, ignore the two nodes and only consider the new one when sorting);
3. sort the remaining nodes;
4. repeat steps 2-3 until there is only one node left, which is the top or the *root* of the binary (Huffman) tree.

When the Huffman tree is completed, you now have a distinct path to each symbol starting from the root of the tree. This path, indicated by the left and right pointers, will be distinct for each symbol. It is interesting to note that the left and right "children" pointers can now be easily substituted by the binary digits 0 and 1, respectively. Thus, the path that you follow from the top of the tree down to the symbol is exactly the bit code for that symbol.

Also, notice that since the low-frequency symbols are processed first by the algorithm, they tend to be at the bottom of the Huffman tree, and the high-frequency symbols are cleverly positioned at the top of the tree. Hence, the higher-frequency symbols, already located at the top of the Huffman tree, will have *smaller* codelengths.

Suppose we have the following set of symbols from the data source and the distribution of their frequency values.

Symbol	Frequency
'A'	17
'B'	4
'C'	8
'D'	3

To build the Huffman tree, create nodes for all symbols and sort them according to their frequency counts, from the smallest to the highest, and store them in a list:

'D'	'B'	'C'	'A'
3	4	8	17

Next, you get the two least-frequent symbols and create a new node to become their parent node (denoted here as P1), as in *Figure 1*:

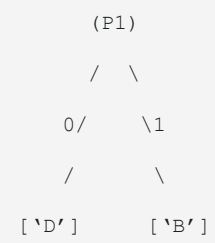

Figure 1. The New Parent Node (P1)

After establishing the right links between the nodes, assign to the parent node's frequency count the sum of its children nodes' frequency counts (3 and 4), which is 7. We then insert this new parent node (P1) into the list; it will be at the start of the list, being smaller than symbol C's frequency count which is 8:

P1	'C'	'A'
7	8	17

Again, we fetch the two least-frequent nodes from the list (P1 and 'C') , and create another parent node (P2). Thus, we arrive at the following tree (remember that P1 has its own children too):

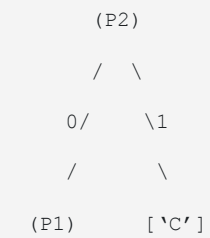

Figure 2. The New Parent Node (P2)

Next, after having created the connections between P2 and its children, assign the sum of the children's frequency counts to P2, and insert P2 into the list. The list will then look like the following, now with only two node entries:

P2	'A'
15	17

Finally, we get the two least-frequency nodes from the list (the very last nodes in this example) and create a new parent node, P3. This last parent node will then be connected to its children (nodes 'A' and P2), as shown below:

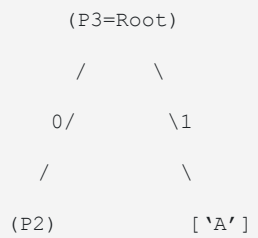

Figure 3. The New Parent Node (P3)

Again, bear in mind that P2 has already the right connections set in place. The new parent node P3 will then become the *root* of the tree, since there are no more nodes left in the list. After this, the tree creation function ends, completely creating

distinct *variable-length* Huffman codes for each symbol of the data source.

The *complete* Huffman coding Tree will then look like the following:

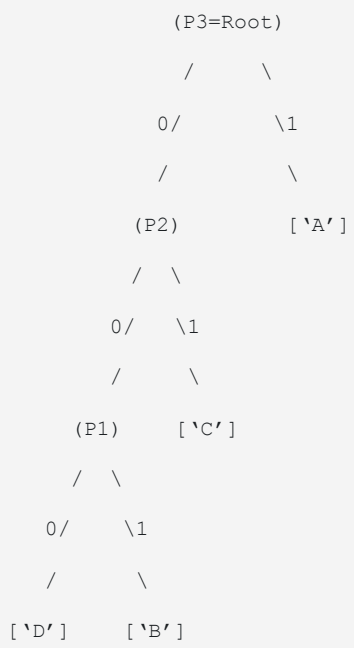

Figure 4. The Huffman Encoding Tree

As you can see in the above Huffman tree, the least-frequent symbols (namely symbols 'D' and 'B') can be found at the bottom of the tree, requiring three bits for their codes, while the *highest-frequency* symbol ('A') has only one bit as its Huffman code and symbol 'C' only needs two bits to encode. Clearly, with this arrangement, the resulting Huffman codes ensure very good compression performance for any data source.

Canonical Huffman Coding

The Huffman tree can be represented more compactly such that only the *length* of the individual codewords is stored with the compressed file. This is called *canonical Huffman coding*. As a simple example, the codelengths can be stored in a look-up table where the indices are their associated symbols. The decoder can then simply re-create the Huffman tree based on the lengths of the codewords.

For a good discussion of canonical Huffman coding, see Michael Schindler's page on "Practical Huffman coding".

For an implemented variant of canonical Huffman coding, see Michael Dipperstein's site, which contains discussions and implementations of various data compression algorithms.

by Gerald R. Tamayo

References

1.] David A. Huffman, "*A Method for the Construction of Minimum Redundancy Codes,*" Proceedings of the I.R.E., September 1952, pp. 1098-1101.

Source Code:

HUFFMAN.ZIP - includes a fast and straightforward implementation of static Huffman coding.