Huffman coding

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code is Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".[1]

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.[2] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods – it is replaced with arithmetic coding[3] or asymmetric numeral systems[4] if a better compression ratio is required.

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols.

A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain a weight, links to two child nodes and an optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The process begins with the leaf nodes containing the probabilities of the symbol they represent. Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children. We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

The simplest construction algorithm uses a priority queue where the node with lowest probability is given highest priority:

Create a leaf node for each symbol and add it to the priority queue.

While there is more than one node in the queue:

Remove the two nodes of highest priority (lowest probability) from the queue

Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.

Add the new node to the queue.

The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log n) time per insertion, and a tree with n leaves has 2n−1 nodes, this algorithm operates in O(n log n) time, where n is the number of symbols.

If the symbols are sorted by probability, there is a linear-time (O(n)) method to create a Huffman tree using two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

Start with as many leaves as there are symbols.

Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).

While there is more than one node in the queues:

Dequeue the two nodes with the lowest weight by examining the fronts of both queues.

Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.

Enqueue the new node into the rear of the second queue.

The remaining node is the root node; the tree has now been generated.

Once the Huffman tree has been generated, it is traversed to generate a dictionary which maps the symbols to binary codes as follows:

Start with current node set to the root.

If node is not a leaf node, label the edge to the left child as 0 and the edge to the right child as 1. Repeat the process at both the left child and the right child.

The final encoding of any symbol is then read by a concatenation of the labels on the edges along the path from the root node to the symbol.

In many cases, time complexity is not very important in the choice of algorithm here, since n here is the number of symbols in the alphabet, which is typically a very small number (compared to the length of the message to be encoded); whereas complexity analysis concerns the behavior when n grows to be very large.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

Arithmetic coding and Huffman coding produce equivalent results — achieving entropy — when every symbol has a probability of the form 1/2k. In other circumstances, arithmetic coding can offer better compression than Huffman coding because — intuitively — its "code words" can have effectively non-integer bit lengths, whereas code words in prefix codes such as Huffman codes can only have an integer number of bits. Therefore, a code word of length k only optimally matches a symbol of probability 1/2k and other probabilities are not represented optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol. This difference is especially striking for small alphabet sizes.