

## Some remarks on General Compression Algorithms

The **Huffman algorithm** is just about as easy as possible to implement.

You create a list of code word lengths for your words.

You first sort the words by order of frequency, smallest first.

Then you take the first two words on the list, add 1 to the length of each, and replace them by a merged word whose frequency is the sum of theirs and whose name is the union of theirs.

You repeat this procedure, increasing the length of each word-element of a merged word by 1 each time you merge. When you do this you must reinsert the merged word with its new frequency into the sorted list.

You stop when you have merged all words down to one, and you then have a list of word code lengths, one for each of your words.

You may then construct a binary tree whose leaves are as far from the root as the lengths on the list, and assign the code words to the words by assigning 0 to one branch and 1 to the other throughout the binary tree, and reading off a code word for a leaf according to the 0's and 1's on its path from the root.

### The Hu Tucker Algorithm

This is the algorithm to use if the original words are completely ordered, and you want the order of the code words (using ordinary numerical order with a decimal point in front of each code word) to agree with that of the original order, but want to compress as much as possible.

It works the same way as Huffman's algorithm, except that you only merge words that are "compatible" with one another. Two words are compatible if there is no original (as opposed to already merged at least once) word in between them whose frequency is greater than both of theirs.

Again you merge and increase word code lengths but now you can merge any two words, say A and B if A has the smallest frequency among words compatible to B and vice versa.

Again you continue until all words are merged into one word and again at each merge you update all the member words in each word that merges.

From this point of view only the last step is different in this algorithm, and that is the step of producing the code from the code word lengths.

Here is how you do this:

Suppose your original words were ordered from left to right.

Then assign the leftmost word the codeword that is all 0's, of the length you have found for it.

Then for the next word, throw away any least significant 1~Rs from the code word of the immediate predecessor, convert the least significant 0 to a 1, and add additional 0's if necessary to increase the word length to what you have computed it to be (The first time you do this, of course, there are no least significant 1's to throw away.).

You repeat this step until each word is assigned a code word. The last word on the right should get a code word that is all 1's.

Here is an example with 8 words, having names and frequencies, respectively, as follows:

1-2, 2-3, 3-7, 4-23, 5-4, 6-3, 7-3, 8-4

Initially 1 and 2 and 6 and 7 can be merged which gives  $L(1) = L(2) = L(6) = L(7) = 1$  and we have the new name frequency list:

12-5, 3-7, 4-23, 5-4, 67-6, 8-4

Now 12 and 3 can merge, as can 5 and 8, and we get

$L(1) = L(2) = 2$ , and  $L(3) = L(5) = L(6) = L(7) = L(8) = 1$ ,

with name-frequency list

123-12, 4-23, 58-8, 67-6.

At this point we can merge 58 with 67, and then 4 with 123, leaving frequency list

1234-35, 5678-14

and length list

$L(1) = L(2) = 3$ ,  $L(3) = L(5) = L(6) = L(7) = L(8) = 2$ ,  $L(4) = 1$ .

Finally, we make the final merge, which raises the length of each word by 1. Our frequencies total 49, and our final length list is

$L(1) = L(2) = 4$ ,  $L(3) = L(5) = L(6) = L(7) = L(8) = 3$ ,  $L(4) = 2$ .

We now produce the code words  $C(j)$  for source symbol  $j$ . We set  $C(1) = 0000$ , and get  $C(2) = 0001$  (last 0 became a 1), then  $C(3) = 001$  (last 1 goes away then and last 0 becomes a 1) Then  $C(4) = 01$  (same way) and  $C(5) = 100$ ,  $C(6) = 101$ ,  $C(7) = 110$ . and  $C(8) = 111$ .

The total message length is  $(2+3)*4 + (7+5+3+3+4)*3 + 23*2$  or 132, compared to 147 if every word had a 3-length code word. By comparison, the Huffman tree will be 17 better or 110. The Shannon bound (which is a bit off because of our approximations) is 119 here.