Finding Efficient Compressions; Huffman and Hu-Tucker Algorithms

We now address the question:

How do we find a code that uses the frequency information about k length patterns efficiently, to shorten our message?

We have, for each of our blocks, say block q, a number of occurrences, f(q), and we want to assign a code word c(q). Let l(q) be the length of c(q). We want to minimize the sum over all q of the product f(q)*l(q).

The way to do this is remarkably simple. It is based upon the following two statements.

1. We should, for efficiency, always give the least frequently occurring blocks the longest code words that we have.

2. There are, in any efficient code, at least two code words with maximum l(q).

We will first notice how these two statements provide us with an algorithm for producing a code from our frequency information. Then we will see where they come from.

We first note how a code can be represented by a tree.

We will associate a binary tree to a code, by assigning a set of code words to each node (or vertex) of the tree.

I like to look at our trees upside down. Then the top node is called the "root" of the tree, and to the root we assign the entire set of c(q)'s for all q's.

We give each vertex either two children or none. Each child will be assigned those of its code words that have a fixed value (either 0 and 1) to the first digit not fixed in the parent. We might as well assign the code words for which this digit is 0 to the left-hand child, though we needn’t do so.

We also assign labels to the vertices; each child receives the label of its parent followed by 0 if it corresponds to code words whose next bit is 0, or 1 otherwise.

Thus, the left child of the root will correspond to code words that have 0 as their first bit, and will have label 0, its right child will have label 01, and so on, if we identify left with 0 and right with 1.
The four vertices at the second level correspond to the sets of blocks that, in order, have first two bits given by 00, 01, 10, and 11 and we can label the vertices as such.

While we are at it, we might as well assign to each vertex the number of occurrences of the blocks that get code words that begin with its bits.

As we go down the tree, the blocks associated with each vertex become fewer and fewer, and eventually there is a single word associated with the vertex. We make that vertex a leaf of our code tree, so that it has no children.

If we try to assign a code word to an interior vertex of our tree, so that there is some other code word that begins with it, we may have some trouble decoding our code. When all our code words are leaves of the tree, then we can immediately recognize when a codeword ends, since there is no other code word that begins at it does and continues. This allows us to recognize the end of every word unambiguously and hence decode unambiguously.

**So we only use code words that are leaves of our tree.**

Any code will correspond to a tree of this kind. And we can make the following statements about that tree without even knowing what it is.

Let us look at the longest code words. They will correspond to leaves at the very bottom of our tree. Each has a parent, and each parent has two children, so that each word at the bottom of the tree has a sibling there as well, which was our second statement.

To find our optimal code, we will work backwards from the bottom of the tree as follows.

Since the least frequent two blocks say, blocks x and y should be at the bottom level of the tree, and each leaf there has a sibling with the same parent, we might as well choose x and y as siblings. Which means we assign the last bit of x to be 0 and of y to be 1, and require that their other bits be identical.

This simple act reduces the problem of creating an optimal code with N blocks, to one of creating one with N-1 blocks, where these two least frequent blocks are replaced by a new artificial block (xy) which occurs whenever either of them occurred, that is, with frequency that is the sum of theirs.

This is all we need to create the entire code. We repeat this step until there is only one artificial block left. In doing so we will assign combination names to merged blocks which will consists of a parenthesis surrounding the juxtaposed names of the two blocks, in any order.
Consider this example: suppose we had 8 blocks with frequencies 20,8,4,3,2,2,1,1, which we call a,b,c,d,e,f,g, and h respectively.

First we apply this step to the two blocks with frequency 1. They are assigned (0,1) and their joint frequency is 2. The remaining problem has frequencies 20,8,4,3,2,2,2 and names a,b,c,d,e,f,g and (hi).

We now combine g and (hi) and the problem reduces to frequency list 20,8,4,4,4,3.2 and names a,b,c,d,(g(hi)),e,f. Next we combine e and f and get new problem: 20,8,5,4,4,4 with names a,b,(ef), c,d,(g(hi))

We will just list the problems that arise the rest of the way here.

20,8,8,5,4 names a, b, (d(g(hi))), (ef), c
20,9,8,8 names a, (c(ef)), b, (d(g(hi)))
20,16,9 names a, (b(d(g(hi)))), (c(ef))
25,20 names (b(d(g(hi)))(c(ef))), a
45 name (a((b(d(g(hi))))(c(ef))))

The last of these is the answer, not a problem at all.

And what answer is it? From the name on the right here you can read off how many times each block got merged in the tree, which is the length of the code word assigned to it and is here the number of parentheses which surround it. Thus a is merged once b and c three times, d e and f 4 times g 5 times and h and i 6 times. This tells us that our encoding tree can have two leaves at level 6, one at level 5, 3 at level 4, 2 at level 2 and 1 at level 1, and tells us further which blocks should be assigned to which levels.

This information is all we need to construct an optimal tree.

Notice that 0 and 1 could be interchanged at each vertex without changing optimality, and it does not really matter here which block gets which code word as long as the code words have the same length.

Notice also that we often had ties for which two blocks had smallest frequency and we could have gotten different optimal trees by making different choices.

This procedure is called Huffman's algorithm after its discoverer.

It is so easy, that we try a harder problem just to keep you awake.

Exercise 1: find a small example of frequencies for which there are two differently shaped optimal trees for this problem.
The Hu-Tucker Algorithm: Shortest Alphabetical Codes.

Our block indices have a natural ordering. They are bit strings of length k, and can be ordered as numbers.

We now ask, how can we construct a code that has minimum total message length as before, but the ordering of the code words (where word x is smaller than word y when at the most significant bit where they differ x is smaller.) is required to be the same as the ordering of their corresponding block indices.

Thus, block with index A must get a code word that is "lexicographically smaller" than the code word for the block with index B, when A<B.

Of course blocks also have frequencies as before and we still want to minimize the length of the total message, subject to this restriction.

There is a neat algorithm for this problem, though it is somewhat strange, and not at all easy to prove that it works. (The first few published proofs were wrong.)

Here is how it works: There are three steps:

First, starting with a frequency list, you merge blocks together, just as in the Huffman algorithm, except that the rules are slightly different.

Second, you use the resulting merge pattern to determine the length of each code word.

Third you construct the final coding tree from these lengths.

How does the merging step differ here?

In the previous problem we merged the two blocks that have the smallest frequencies together.

Now we introduce a notion of compatibility among blocks. Again, we will have original unmerged blocks and artificial merged blocks.

The compatibility rule is: you can only merge two blocks if there are no unmerged blocks left between them. Thus if you have three blocks in order with frequencies 2,4,3 you cannot merge the 2 and 3 frequency blocks together unless the 4 frequency block has already merged at least once.

And here is the rule for merging: if x is the lowest frequency block compatible to y and y is the lowest frequency block compatible to x, you can merge them.
So here is how the algorithm goes: you merge blocks together according to this rule until they all merge into one; you keep track of the number of merges of each block, in order, which will be the lengths of the code words of the blocks.

Then you construct an alphabetic tree having these lengths. There will be only one possible way to do this.

We have yet to describe how to perform this last step. But before doing so let us look at an example.

Suppose our frequencies are, in the order that we want to preserve:

1, 2, 23, 4, 3, 3, 5, 19.

At this stage each block is compatible only with its immediate neighbor. The only pairs that obey our condition that x is the lowest compatible with y and vice versa are the first two and the 3, 3. we can merge each of these adjacent pairs, getting frequencies

3*, 23, 4, 6*, 5, 19,

where the star means that the block has been merged.

If we give the blocks names, a, b, c, d, e, f, g, and h in order here, we can denote the first intermediate problem that of 3*, 23, 4, 6*, 5, 19 with names (ab), c, d, (ef), g, h

Now we can merge the 3 and 23, and also the 4 and 5, which we can locate where the old 4 was: to get frequencies 26*, 9*, 6*, 19 with names ((ab)c), (ef), (dg), h. The remaining problems are

26*, 15*, 19 with names ((ab)c), ((ef)(dg)), h
26*, 34* with names ((ab)c), (((ef)(dg))h)
60* named (((ab)c)((ef)(dg))h))

Again we can read off the word lengths from the depths of the names in parentheses in this last step. We get here a-3, b-3, c-2, d-4, e-4, f-4, g-4, h-2.

To find the actual code words we can start from the leftmost word, which we assign all 0's. (here 000 because it has length 3)

Then we proceed along and construct each succeeding word from the previous word by the following rule:

1. throw away any final 1's
2. convert the last 0 to a 1
3. add additional 0's if necessary to make the length of the word right.

In this case, the words would be:
Comments:

There are proofs that this method gives the best possible order preserving (prefix free) code, but they are surprisingly hard to find.

There are other things that one can show: if you have a given set of frequencies, the ordering of the blocks that makes this code have the longest length occurs if you put the rarest block first, then the most common block, then the second rarest, then second most common, then third, etc.

And doing it that way can require at most one more bit per block than the Huffman non-order preserving code solution has.

Exercises:

2. Find the best order preserving and non-order preserving code for the following block frequencies, in order. Also compute the Shannon theorem (H(\{p(q)\})) bound for these frequencies.

\[1, 21, 3, 4, 5, 35, 5, 4, 3, 5, 98, 21, 14, 17, 32\]

3. The Shannon bound will be exact if the frequencies of the two children of each node are exactly equal. For 3, and 4 blocks, find an arrangement of \(p(q)\) values that force you to deviate the most or nearly the most from the Shannon bound. (For example, for two blocks the Shannon bound approaches 0 as \(p(1)\) approaches 0. Yet any code for which \(p(1)\) is not 0 requires codeword length 1 for each block. What similar statement can be made for more blocks?)