

6. Finding Efficient Compressions; Huffman and Hu-Tucker Algorithms

6.1 What the Huffman Algorithm Solves

The main problem of efficient compression lies in finding a way to turn k -length patterns of information into a shorter coded message.

An example of this would be the frequencies of words in a paragraph. To rephrase the problem, “how can we describe a shorter way of expressing the number of times k different words appear in the paragraph of text?” This is only one example of information of some sort being compressed. Others include compressing the information on the location of pixels in a photo into a smaller file, or data files. The Huffman algorithm is an easy algorithm that helps us to compress information.

Named after David Huffman, a former student at MIT, the Huffman algorithm was proved to be the most efficient method of sorting. Huffman had the option of doing a final term paper or taking a final exam. Huffman, being an intelligent student much like yourself, made his life easier by writing about his algorithm to get out of taking a final.

Now let's express what we want to do in function notation so that we may understand the rest of the notes in a mathematical context. Continuing with the previous example say for each block of words “ q ,” the number of occurrences or “frequency” of that word is $f(q)$, for which we want to assign some code $c(q)$. Let $l(q)$ be the length of the codeword. In order to efficiently compress the information we want to minimize the sum over all q for the product $f(q) \cdot l(q)$. In other words, we want the least number of codewords all having the shortest length possible!

There is a remarkable way of doing this based on the two following rules.

Rule Number 1: In order to be efficient we always give the least frequently occurring blocks the longest codewords that we have. This makes sense because we want the more frequent words to have a shorter coded length and the least frequently used word to have the longest codeword.

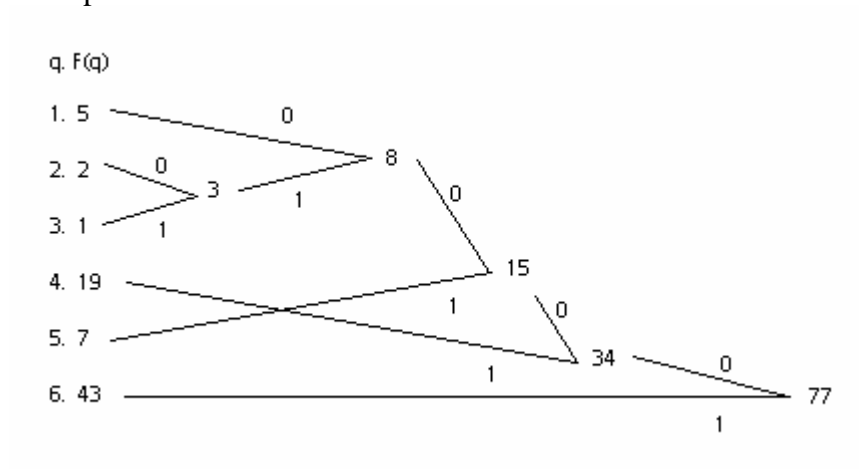
Rule Number 2: There are, in any efficient code, at least two longest codewords. This will be proven later.

We use these two statements to produce a code from frequency information, and to further understand the two statements. The most useful way of representing the code is through “tree diagramming.”

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

The rightmost node is the "root" of the tree, and to it we associate the entire set of blocks which occur at all in our message. The way that the tree is setup is by listing the blocks and their frequencies on the left from top to bottom in any order you please. You then connect the blocks using the methodology that will be outlined below.

Example of a tree-



In the above diagram the "root" would be 77 with the corresponding blocks 1-6 having frequencies of 5, 2, 1, 19, 7, 43 respectively. The Huffman algorithm consists of merging the two shortest frequency blocks in an iterative process. In the first combination 2 and 1 are combined to make 3. The next step would be to combine 3 and 5.

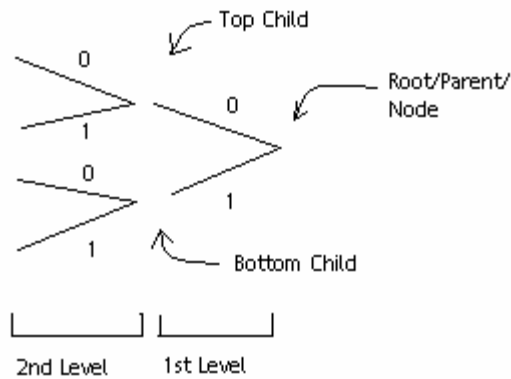
6.2 Picturing the Binary Tree

The purpose of this section is to sketch out in your mind what binary trees look like and how they operate.

In a binary tree, each vertex/node will have two connecting lines labeled either 0 or 1. Each node except for the block nodes should have two "children" nodes. The convention we will be using puts the 0's on the "top" line and 1's on the "bottom" line. We label our vertices with the label of its parent plus an additional bit of 0 if it is the top child or 1 if it is the bottom child of the parent. Looking at the example tree 77 would be a parent with children 34 and 43.

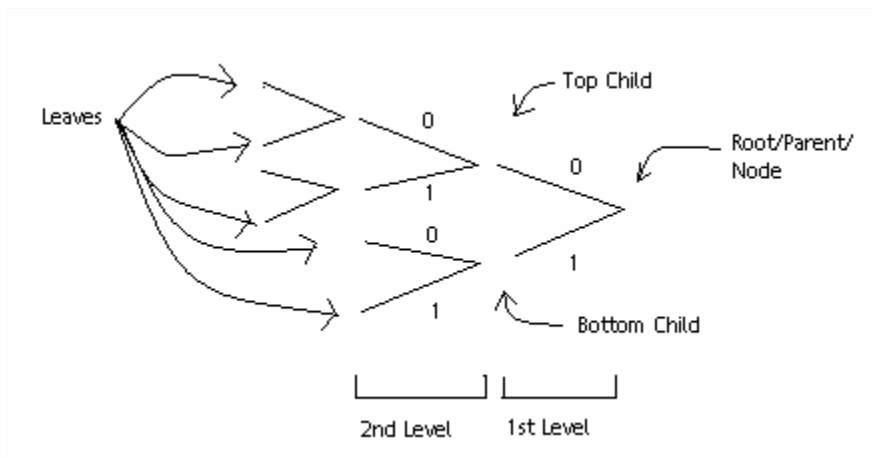
Thus, the top child of the root corresponds to the set of the blocks that have 0 as their first bit, and the bottom child corresponds to those having 1 as a first bit, and so on down the tree. The top child has label 0, the bottom child label 1. Looking at the root 77 we have a top child of 34 and a bottom child of 43.

The four vertices at the second level of a binary tree 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.



We will also assign to each vertex the number of occurrences of the blocks that get codewords 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 block associated with the word. We make that vertex a leaf of our code tree, so that it has no children.



If we assign a codeword to an interior vertex of our tree, so that there is some other codeword that begins with it, we may have some trouble decoding our code. If we have all our codewords be leaves of the tree, then we can immediately recognize when a codeword ends, so that we can unambiguously decode any message. The code for the topmost leaf to the bottom most leaf is 000, 100, 010, 110, 01, 11.

So we only use codewords that are at the leaves of our tree.

Now let us look at the longest codewords. They will correspond to leaves at the left 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 partner there as well, which was our second rule.

6.3 The Huffman Code Algorithm (Non Order Preserving)

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

Since the least frequent two blocks should be at the bottom level of the tree, and each leaf there has a sibling with the same parent, we choose them as siblings. We assign the last bit of one of them to be 0 and of the other to be 1, and require their other bits to be identical. These two least frequent blocks are replaced by a new “artificial” block. This artificial block is the sub-root/root of two least frequent blocks and whose frequency is $f(\text{least frequent block}) + f(\text{next least frequent block})$, that is, with frequency that is the sum of theirs. We end up with a new artificial block with a joint frequency that is the sum of the previous two frequencies. This simple act reduces the problem of creating an optimal code with N blocks, to one of creating one with $N-1$ blocks. We repeat this step until there is only one artificial block left. The blocks that we started off with are called the “original” blocks.

Consider this example: suppose we had 9 blocks with frequencies 20,8,4,4,3,2,2,1,1,

First we apply the artificial block step to the two blocks with frequency 1. The new artificial block is assigned (0,1) and their joint frequency is 2. We combine them next with the one of the other blocks with frequency 2 to produce one with frequency 4 whose codeword end with (0,10,11)

Next we merge the other frequency 2 block with the frequency 3 block to get a frequency 5 block that internally is (0,1).

At this point our frequency sequence looks like: 20, 8,5(0,1), 4,4,4(0,01,11)
Applying our step to the last two and then the next two produces the sequence

20, 9(0,10,11), 8, 8 (0,10,110,111)

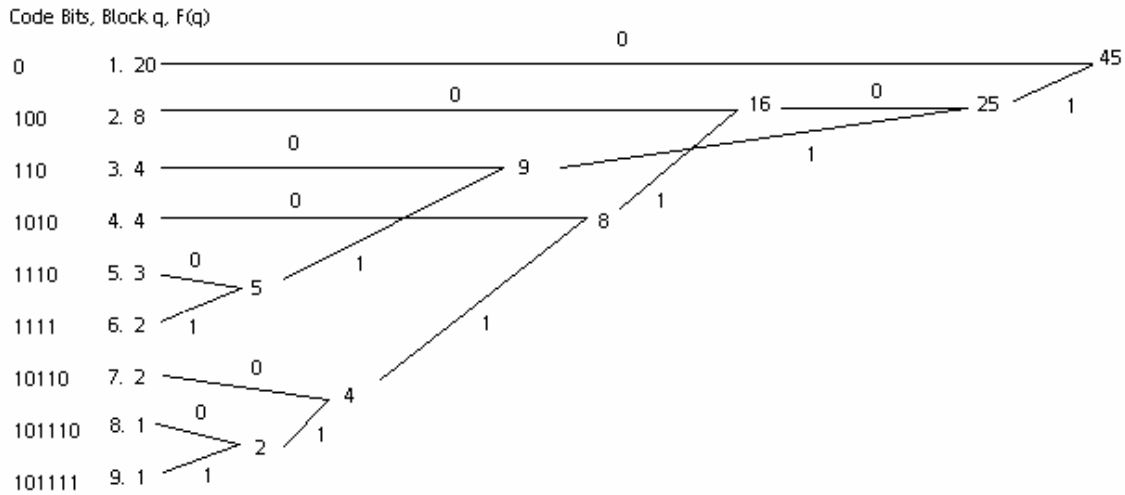
Next the two 8's can be combined, and then the resulting 16 and 9 and we have

20, 25 (00,010,011,10,110,1110,11110,11111)

And finally we merge these, and we have our codewords

(0, 100,1010, 1011,110,1110,11110,111110,111111)

Example



We can assign these to blocks by ordering them by their lengths, shortest first, and ordering the blocks by their frequencies, largest first, and pairing them.

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 codeword as long as the codewords have the same length. We also do not have to keep track of the codeword suffixes at each node as we did above; it is sufficient to keep track of the lengths of these suffixes, and we can easily recreate a code with codewords having the given lengths from that information

6.4 The Hu-Tucker Algorithm: Shortest Alphabetical Codes

The blocks we have been discussing have a natural ordering. They are bit strings of length k , and are ordered in binary number notation.

Our new question is, “how do we construct a code that minimizes total message length, but the ordering of the codewords is preserved from the original ordering of their corresponding blocks. This means that if the codeword for x is smaller than codeword y at the most significant bit where they differ, then x is smaller.

Thus, the block that is smaller in appearance in the given block ordering gets the "lexicographically smaller" codeword. The alphabetization is synonymous to ordering.

Block still 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.)

6.5 Overview of 3-Step Method for the Hu-Tucker Algorithm

First you merge vertices 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 codeword. Third you construct the final tree from these lengths. We will talk about how the two algorithms differ and then walk you through an example.

In the Huffman Algorithm we merged the two blocks that have the smallest frequencies together. We do the same thing in the Hu-Tucker algorithm but we must take into account the “compatibility” of the blocks. We will also still keep concepts of original blocks and artificially merged blocks from the Huffman algorithm.

6.6 Rules

The compatibility rule is: “you can only merge two blocks if there are no original 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 is artificial.

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 should merge them.

6.7 Hu-Tucker Algorithm (Order Preserving)

1. You merge blocks together according to the rules until you get one block
2. Keep track of the number of merges of each block (in the order we wish to preserve). The number of merges will be the lengths of the codewords of the blocks
3. Construct an alphabetic tree having these lengths. There will be only one possible way to do this. In other words if you and your buddy had the same original blocks you should end up with the same Hu-Tucker tree.

We have yet to describe how to perform this last step. Let us do an example and then show how the last step is performed.

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 neighbors. The only pairs that obey our condition that x is the lowest compatible with y and vice versa are the 1 and the 2 and the 3,3. We can merge each of these, getting

3(1,1), 23, 4, 6(1,1) 5, 19.

The numbers in the parentheses are the lengths of the suffixes of the words merged into the artificial block indicated by the preceding number. So 3 (1,1) means in the artificial block 3. There are two smaller blocks connected with the artificial block with respective path lengths of 1 and 1.

Now we can merge 3 and 23, and also 4 and 5, which we can relocate where the old 4 was. We will end up with

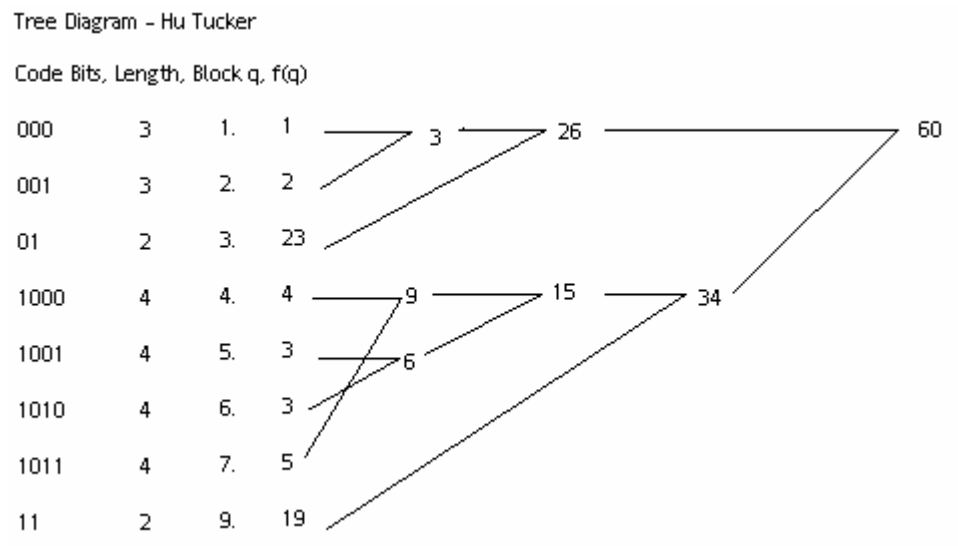
$$26(2,2,1), 9(1,1), 6(1,1), 19.$$

At this point the compatibility restriction is no longer required and merging is like it was in the Huffman algorithm.

Thus, next we can merge the 9 and 6 which gives $(26(2,2,1), 15(2,2,2,2), 19)$, then the 15 and 19, and finally the remaining blocks together, getting

$$26(2,2,1), 34(3,3,3,3,1) \text{ then } 60(3,3,2,4,4,4,2).$$

Here is the final tree diagram for the example.



“Alphabetizing” Our Code for the Hu-Tucker Algorithm (The 3rd Step)

To find the actual codewords we can start from the topmost block’s codeword, which we assign all 0’s. In our example it is “000.” The number of 0s you need corresponds to the length. The length is 3 because it takes three lines to get from 1 to 60 in the above diagram.

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

1. Throw away any 1’s at the very end of the previous codeword.

2. convert the last 0 to a 1
3. Add additional 0's to the end of the codeword if necessary to make the length of the word equal to the length of the block.

Here is a quick example of the process. Turning the codeword for block 3 into block 4 we have 01 becoming 0 because we get rid of the last 1. The 0 turns into a 1 because of rule number 2. And then since block 4 has a length of 4 we add three 0s after it to get 1000.

If you follow the directions above you should receive the code bits in the diagram above.

Comments:

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

You can also show that 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. By ordering these blocks in this fashion, we will require at most one more bit for each block's codeword than in the Huffman non-order preserving code.

The creators of the Hu-Tucker Algorithm are T.C. Hu, a professor of computer science at UCSD and A. C. Tucker.

Exercises

Exercise 1 Find the best order and non-order preserving code for the following block frequencies, in order. In addition, 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

Exercise 2 Same as exercise 1 only with the following frequencies...

3, 3, 82, 4, 5, 10, 12, 23, 52, 2, 1, 5, 1, 8

Exercise 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?)