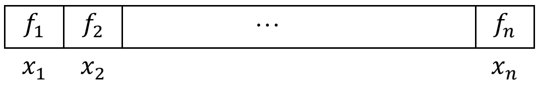
Implementation of Huffman Encoding by Abusing Data-Structures

Greg DePaul

Abstract
Create an implementation of Huffman Encoding for an arbitrary subset of ASCII characters. This implementation makes explicit use of data structures to ease processing.

Implementation Approach

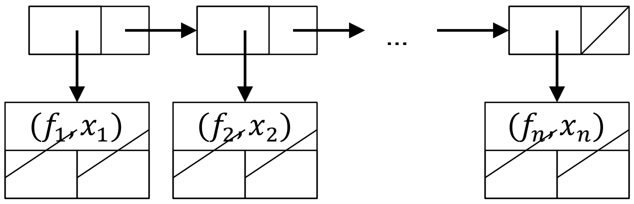
Consider an alphabet $X = \{x_1, x_2, \ldots, x_n\}$. We want to read a text document for which its language is a subset of X^*. It is clear X is finite. We want to optimize access to frequency time while reading a text. Therefore, to do this, we simply pre-compute a finite array of size n. So we initialize an array with a one-to-one relationship to our alphabet with the absolute frequencies of the symbols in our document. Graphically:



Now we run into the issue of jumbling frequencies while generating the Huffman tree. This is where abusing a data structure becomes effective. Consider the objectives we would potentially want in a data structure:

a) $O(1)$ access time to retrieve minimum root.
b) Remove already placed nodal elements from consideration (to avoid reassigning Huffman codes)

Clearly an array cannot satisfy these since the array cannot be assure sorted elements nor the ability to easily remove elements from the array. These two objectives imply a linked list structure, but a simple linked list in which the data component consisting of the ordered pair (symbol, frequency) doesn't facilitate constructing a tree. Instead, we'll consider the a mixed data structure, in which we have a linked list of potential roots which then carry the trees of already placed nodes. Graphically, we initialize this list in order of frequency for all our symbols:



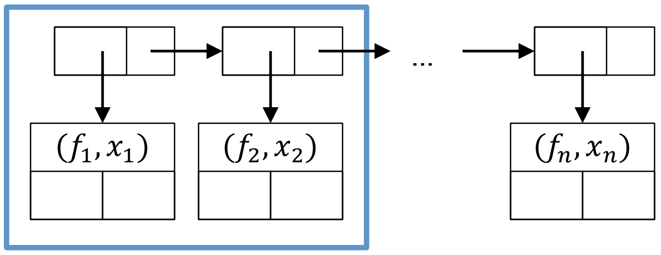
So what if we repeatedly *relax* this data structure? That is, by a greedy approach, take the two least frequent symbol roots and then join the two together with a new root. For two roots (f_i, x_i) and (f_j, x_j), the new root can be denoted by $(f_i + f_j, x_i x_j)$. For the sake of implementation, $x_i x_j$ doesn't possess any useful meaning, so I take the liberty of setting the value to -1. Then the new root $(f_i + f_j, x_i x_j)$ is reinserted back into the list, which automatically inserts it in order. The code specific to this is detailed below in *Algorithm 1*.

Alg. 1. – Huffman relaxation process using a mixed data structure.

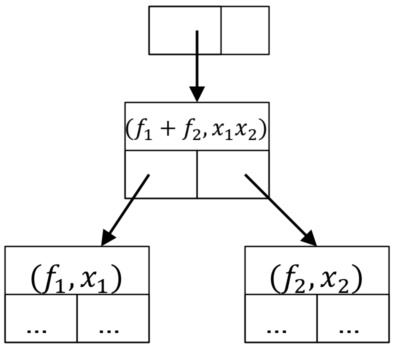
```
while(myList->next != NULL) {
        LinkedRoot* newJointRoot = newLinkedRoot(newBinaryNode(-1, -1));
        newJointRoot->data->left = myList->data;
        myList = removeLeastRoot(myList);
        newJointRoot->data->right = myList->data;
        myList = removeLeastRoot(myList);
        newJointRoot->data->frequency = newJointRoot->data->left->frequency + newJointRoot->data->right->frequency;
        myList = addRoot(myList, newJointRoot);
    }
```

Let us consider a sample iteration of the relaxation:

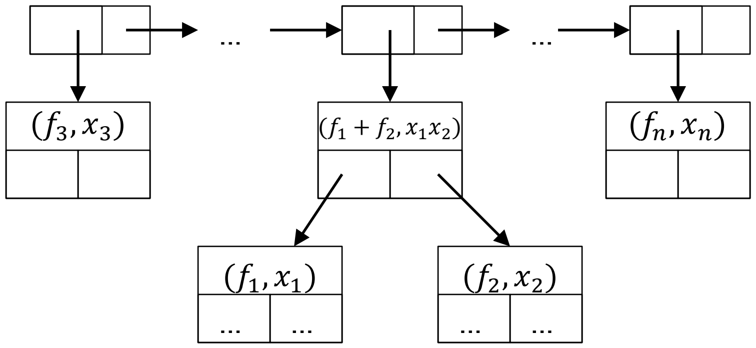
1.) Select the two least roots and pop them from the list:



2.) Generate a joint root node:



3.) Reinsert the joint root node into the list:



And continue performing this relaxation until there is only one root, *specifically the root of the Huffman Tree*. Upon reaching the stop condition, you will have a binary node $(f_1 + f_2 + \cdots + f_n, x_1x_2 \ldots x_n)$. So you can also, without having to iterate over the character array that we constructed in preprocessing, retrieve to total number of characters within the document being compressed.

Results

In order to test my implementation, I used four different classic texts encoded under the same 128 ASCII character set. Clearly to sustain 128 selectable characters, ASCII characters are encoded with a length of 8 bits. Huffman coding, on the other, enables us to benefit from the fact that rarely do we use all 128 characters. If we in fact used every character with the same frequency, then the entropy of that textual information would be 8 bits. Since this is not the case, we should see the trend that written language converges to a different level of entropy, which my results clearly show in Table 1. Specifically, the four classic English texts tend to have entropy of 4.489 bits. Also notice that for each of the columns, my calculated average bit length is between the calculated entropy and one greater than the calculated entropy. Therefore, my results satisfy Shannon's Source Coding Theorem.

Table 1 – Average Bit Lengths and Compression Ratios for Selected English Classics.

		sawyer.txt	mohicans.txt	mobydick.txt	musketeer.txt
Word Count	[words]	6760	15787	19932	31172
Entropy	[bits]	4.478	4.451	4.460	4.568
Average Bit Length (ABL)	[bits]	4.523	4.479	4.492	4.606
ABL / 8		**0.565**	**0.560**	**0.562**	**0.576**
Original Size	[bytes]	391534	881785	1220927	1354239
Compressed Size	[bytes]	221376	493706	685556	773863
Compressed / Original		**0.565**	**0.560**	**0.562**	**0.571**

One of the interesting things of this data is that you may notice that for three of the above columns, the bolded numbers are the same. But the last column, which is the result of calculating the Huffman coding on a copy of the text for Three Musketeers, you'll notice the two ratios aren't the same. Why is that you may ask? This is a result of the spike in entropy, which you will notice is nearly a tenth larger than all the other entropy values. This new entropy will then affect the compression size, often over inflating the average bits per length for characters that have a frequency that borders on never occurring. My algorithm was designed to accept nonvisible characters in the case of their occurrence, and Musketeers has the ASCII character 13 pretty frequently in the text, which is the carriage return often used for windows based texts. I hope this serves as an explanation as to why the two ratios, which generally equal, are in this case different.

Conclusion

I was able to successfully implement a Huffman Encoder. My specific implementation grants me spares me the pain of detailing any rigorous lengths of code because my data structure takes care of the majority of my processing needs on my structure. An unintended result of this project is that written (English) language has a very specific level of entropy, approximately ~4.5 bits. So for the majority of English files, we can compress them around to 60% their original size, assuming each character is independent of all others. A potential idea to further this project would be to include entropies in which entire words have probabilities in order to further compress a file.