Aaron Partridge

Dr. Richard C. Tillquist

CSCI 411 - Advanced Algorithms

04 May 2025

Huffman Coding

Introduction:

In 1951, David A. Huffman was a Ph.D. student at MIT. For a class taught by Professor Robert M. Fano, students were given the choice between taking a final exam or writing a term paper to develop a more efficient method for creating binary codes. Huffman opted for the term paper, but initially struggled to prove that any method could outperform his professor's algorithm. On the verge of giving up, he had a breakthrough, using a frequency-sorted binary tree to generate codes. This bottom-up approach produced an optimal prefix code, ensuring that no code was a prefix of another. Huffman's method turned out to be not only more efficient but also simpler than Fano's top-down algorithm, now known as the Shannon-Fano algorithm.

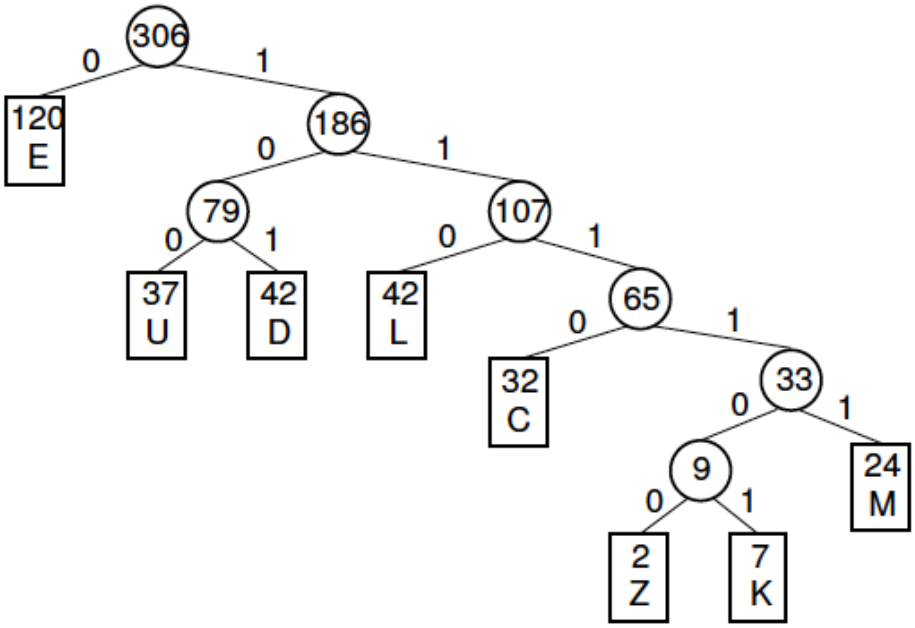
Without compression algorithms, companies like YouTube, Netflix, Amazon, and even Google would struggle to function efficiently. These platforms rely heavily on compression to transmit data quickly and store it at scale. Imagine if YouTube had to store its estimated 5.1 billion videos in their original file sizes, or if Netflix had to stream uncompressed, full-length movies to over 300 million users. The result would be incredibly slow performance and astronomical storage costs. Compression is essential because it reduces the space needed to store data and makes transmitting it far more efficient. This is where Huffman coding comes into play;

it is a core component of many compression algorithms today, such as .zip, .gzip, and multimedia formats like .jpeg, .png, and .mp3. Thanks to Huffman coding and other compression algorithms, companies can store more data for less and serve more customers in a fast and efficient manner.

Intuition:

The flow of Huffman coding is fairly simple. First, you have to traverse your data and count the number of occurrences of each character. Then, you can order this data by its frequency from greatest to least. The next step is a little more complicated because this is where you build the Huffman tree. The idea is that you start assigning parent nodes that hold the value of the two child node frequencies. Then continue in this fashion until you have one root node at the top. Once your tree is built, it is time to assign edge weights. The common practice is to assign a “0” if going left down the tree and a “1” if moving right. Left indicates a higher frequency, and right means a lower frequency. Now, for each character, traverse the tree until that character is reached and record the edge values that were traversed. When this is complete for every character, assemble the binary values for each character, and this is the encoded version of the original data.

Here is an image to help illustrate the encoding process:



Source: http://homes.sice.indiana.edu/yye/lab/teaching/spring2014-C343/huffman.php

Finally, for decompressing, you can simply use the compressed code to traverse the tree by following the 0's and 1's until a character is reached. Then assemble the final message when the process has been completed. This will be the original message before it was compressed.

Pseudocode:

This is the overall flow of the algorithm:

getFrequency()

makeHuffmanTree()

getHuffmanCodes()

encode()

decode()

To help with coherence, I will be going over the runtime of each function after the description of the pseudocode.

```
getFrequency(originalMessage, nodes)
    For every c in originalMessage
        found = false
        for each node in nodes
            if node.ch == c
                node.frq++
                found = true
        if !found
            nodes.append(new Node(c, 1))
```

Here in getFrequency, we iterate over each character in the original message. If it hasn't been found in our list of nodes, then a node is created; if the character has been found, then its frequency is incremented.

We can see that the initial for-loop is iterating over the size of the input (n), then loops over the list of nodes (m) to search for a match. These are the two primary components of this function, creating a final runtime of **O(n * m)**.

```
makeHuffmanTree(nodes)
        PQ[] //min-heap priority queue
        for each node in nodes
                PQ.push(node)
        while PQ.size > 1
                temp1 = PQ.top
                PQ.pop
                temp2 = PQ.top
                PQ.pop
                tempNode = new Node('\0', temp1.frq + temp2.frq)
                tempNode.left = temp1
                tempNode.right = temp2
                PQ.push(tempNode)
        root = PQ.top
        return root
```

For makeHuffmantree, we use a priority queue with a min heap. This starts at the lower frequency end of the characters and creates a parent node with the sum of the two child node frequencies. Then it continues until there is only one root node at the top.

Since we are using an out-of-the-box C++ implementation of a min-heap priority queue, we can use the runtimes associated with that. Most of the operations performed in the queue take O(log n) time. So when we initially loop over each node (m) and push it to the queue, this takes

O(m log m) time. Next, we enter into a while-loop that runs for m - 1 times. Within the loop, we have operations like PQ.pop and PQ.push that take O(log m) time. So the end result of this loop is (m -1) * (log m) = O(m log m). Combined with the earlier loop, we get 2(m log m). The 2 has no effect as m gets really big, so the final runtime is **O(m log m),** where m is the number of nodes.

```
getHuffmanCodes(node, currentCode)
    if !node
        return
    if node.ch != '\0'
        node.code = currentCode

    getHuffmanCodes(node.left, currentCode + "0")
    getHuffmanCodes(node.right, currentCode + "1")
```

In getHuffmanCodes we start by setting up base cases for if there is no node and if the tree hasn't reached a leaf with a character. This repeats recursively and appends either a "0" or "1" respectively until each character is reached. Every character's code is stored on its own node.

Here we are recursively traversing a binary search tree. So we can imagine a worst-case scenario where it is a one-sided tree where all the nodes are in order, descending down one side of the tree. This would result in **O(m)** time, where m is the number of nodes (or unique characters).

```
encode(originalMessage, nodes)
    compressedMessage = ""
    for each c in originalMessage
        for each node in nodes
            if node.ch == c
                compressedMessage += node.code
                break
    return compressedMessage
```

Encode loops through the original message and searches for a matching node. If a node is found, the compressedMessage variable is updated with that character's code. Then the full compressed binary code is returned.

Here we have another function similar to getFrequency, where there is a for-loop based on input (n) and a nested for-loop according to the number of nodes (m). This runtime is **O(n * m)**.

```
decode(compressedMessage, root)
        result = ""
        currNode = root
        for each c in compressedMessage
                if c == '0'
                        currNode = currNode.left
                else if c == '1'
                        currNode = currNode.right
                if currNode.ch != '\0'
                        result += currNode.ch
                        currNode = root
        return result
```

Decode traverses the tree and follows edges according to the 0's and 1's stored in the compressed message. Once a character node is reached, the result is updated and eventually returned at the end.

During decode, we have a singular for-loop that iterates over the input (n) in compressedMessage. Then it performs O(1) operations within. This results in a final runtime of **O(n)**.

Run Time Analysis Summary:

getFrequency() - **O(n * m)**

makeHuffmanTree() - **O(m log(m))**

getHuffmanCodes() - **O(m)**

encode() - **O(n * m)**

decode() - **O(n)**

For n = total number of characters in the original message

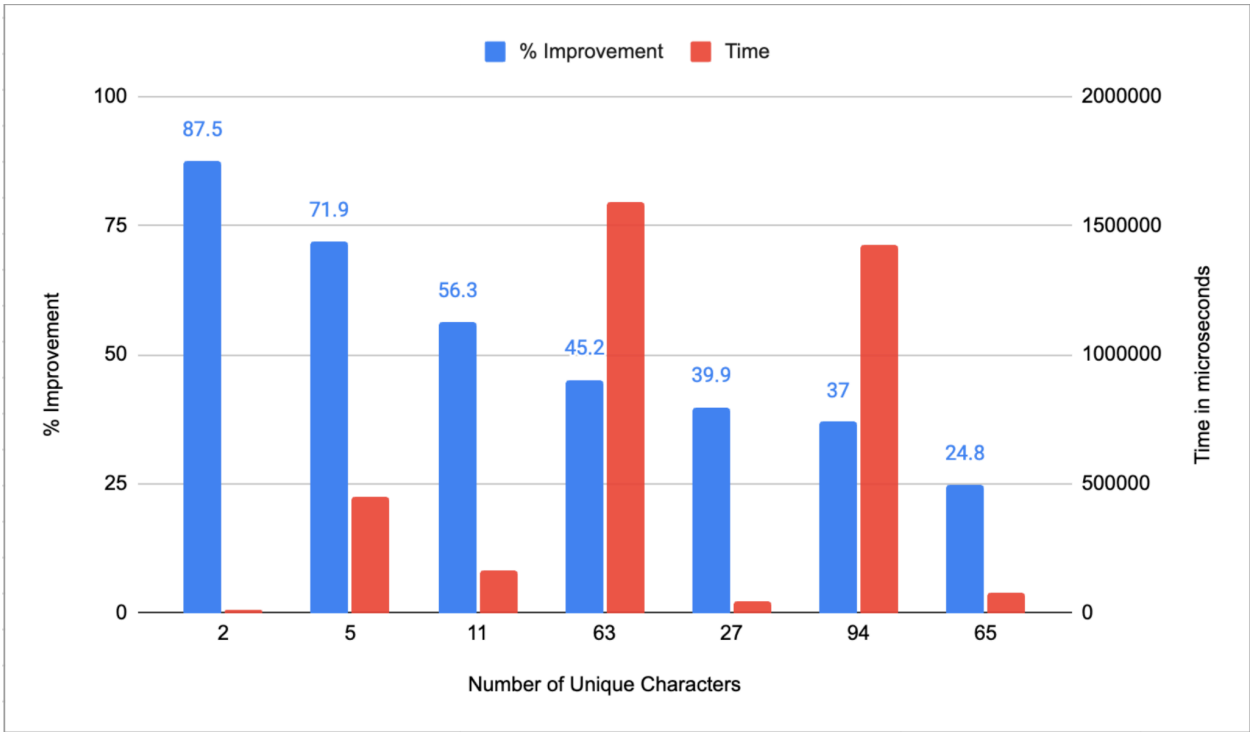
And m = number of nodes (unique ASCII characters)

Therefore, the overall runtime of the algorithm is **O(m log(m))** because this will dominate the rest of the algorithm.

Results:

In conclusion, I found some interesting results after running the algorithm on several different test cases. I downloaded a few different files from https://corpus.canterbury.ac.nz/ to test the compression capability on large files of varying characters and lengths.

From Left to Right	Number of Unique Characters	% Improvement	Time
The letter 'a', repeated 100,000 times.	2	87.5	13738
Complete genome of the E. Coli bacterium	5	71.9	449331
The first million digits of pi	11	56.3	167133
The King James version of the bible	63	45.2	1591809
Enough repetitions of the alphabet to fill 100,000 characters	27	39.9	46780
The CIA world fact book	94	37	1423895
100,000 characters, randomly selected from [a-z|A-Z|0-9|!|] (alphabet size 64)	65	24.8	75871



The data above is organized based on the percent efficiency which it was able to compress. The first test has incredible efficiency because it is the single character ‘a’ repeated 100,000 times with a new line character at the end. Next, we have the entire genetic sequence of E. Coli. This again has high efficiency because it is only four different characters and a new line. In third, we have the first million digits of pi. This also didn’t seem to be much of an issue when

put in terms of efficiency or time, because it is only dealing with digits 0 - 9 plus a new line character. The next few test cases start to get more interesting because now we are dealing with a wide assortment of unique characters in a very random order. Test case four has decent efficiency, but it seemed to take a large amount of time. This could be because the King James Bible is limited to the ASCII characters of the English language, but they appear in such a random or infrequent order that the time it takes to search for that node adds up. The next test case is the alphabet repeated enough times to fill 100,000 characters, indicated by 27 unique characters and 39.9% efficiency, which surprised me. I would have expected that to appear earlier in the chart because of its low amount of unique characters. But it seems that again, unique character count isn't the driving factor in efficiency. This test probably does play closer to a real-world scenario like the King James Bible because the input is so large and there are repeated alphabetical characters. The difference is probably that the Bible has far more *runs* of characters than just the alphabet repeated over and over. Next is the CIA World Factbook. This has a very similar look to the King James Bible. This is most likely due to the same reasons for searching and matching nodes, and being able to find certain runs of the same character to compress. After analyzing the rest of the data and realizing the efficiency is not based solely on the number of unique characters, it is easy to see why 100,000 randomly selected ASCII characters is in last place. There are most likely few amounts of matching characters because there aren't repeated words like in the previous book examples, and therefore, the compressed amount of bits is similar to the original amount of bits.