Huffman Coding Report

Stephen Dumore

May 4, 2025

1 Introduction

A binary prefix code is a sequence of ones and zeros that can be used to represent a particular piece of data (usually a byte). This is useful for lossless compression algorithms because it allows frequently occurring bytes to be represented by codes that are shorter than eight bits. The "prefix" part of the name comes from the fact that a code cannot be the prefix of another code. For instance, if the code for the space character in the table below was '1' instead of '111', it would cause ambiguity when trying to decompress the file. In this scenario, there would be no way of knowing whether a '1' was the code for a space character or the beginning of a code that starts with '1' such as the code for the character 'i' when reading the compressed file as a bitstream.

Char ⬍	Freq ▾	Code ⬍
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Figure 1: A table of binary prefix codes [1]

In 1949 MIT professor, Robert M. Fano proposed a method of generating binary prefix codes. Fano's method can be described as such: Take a list of symbols sorted by frequency and bisect it such that the sum of left frequencies and the sum of right frequencies are as close as possible to equal. Then replace

the list with a root node that has the two sublists as children. Recursively do this to both children until each sublist has a length of one. When complete, symbols will only be stored on leaf nodes and all leaf nodes will have symbols. The binary code for a symbol is the path from the root to the the symbol where left children are represented by zeros and right children by ones. For instance, the code for the character 'D' in the tree below would be '110'. While functional, this method does not guarantee shortest possible code length. [3]

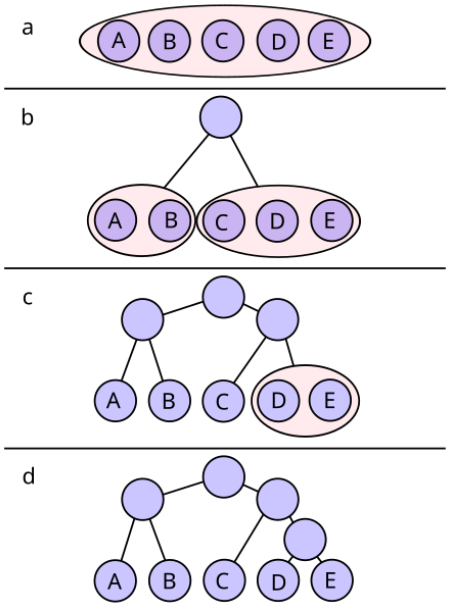

Figure 2: A tree constructed using Fano's method [2]

In 1951 Fano gave the students of his information theory class the choice between a final exam or a term paper on finding the shortest possible code length. Just before giving up, his student David A. Huffman discovered it was possible to generate optimal prefix codes using a combination of a frequency sorted minimum priority queue and a binary tree. The Huffman coding algorithm was born. [4]

Unlike Fano's method which builds a binary tree from the top down, the Huffman coding algorithm builds one from the bottom up. It starts by placing all symbols on a minimum priority queue that sorts based on the frequency of the symbols. Then, two elements are removed from the priority queue. A new element is created, which has a frequency value equal to the sum of the frequencies of the two removed elements. The new element also has the two removed elements as children. The parent and both children are now part of the Huffman tree. The new element is placed back on the priority queue; however, it's children are no longer part of the priority queue. This is repeated while the

priority queue has more than one element. The last element remaining in the priority queue is the root of the completed Huffman tree. Once the Huffman tree is constructed, codes can be found by using the same traversal processes as Fano's method.

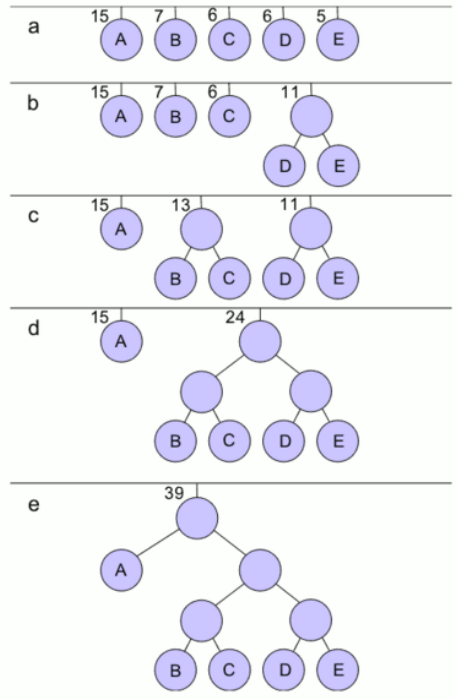

Figure 3: A Huffman tree [1]

The Huffman coding algorithm went on to become the most widely used compression algorithm. It excels at compressing strings since strings tend to have relatively few symbols and often contain frequently repeated symbols. Because of this, HPACK (the algorithm used to compress HTTP headers) uses Huffman coding. Variations of .zip compression are based on Phil Katz's Deflate algorithm, which is a combination of LZ77 and Huffman coding. Data transmitted over the internet is generally compressed using Huffman coding in conjunction with another lossless compression method at the sending end then decompressed at the receiving end. Even "lossy" compression formats like MP3 and JPG use rely on Huffman coding to reduce the file size after the "lossy" algorithm has decreased the uniformity of symbol frequencies.

2 Pseudocode

The following pseusocode assumes that:

- Compression is applied in terms of bytes.
- `freq[]` is an array-like list of 256 bytes where the byte is represented by the index and the frequency by the value stored at that index.
- `Node` is an object with a single parameter copy constructor and a two parameter constructor that takes in a value and a frequency.
- `code` is a list of ones and zeros used to represent a Huffman code.
- `dictionary` is a list of 256 codes.
- `priority_queue`'s `.pop()` method returns a copy of the element on the top of the queue before removing it.

```
// Returns the root of the Huffman tree
Node* createHuffmanTree(freq[])
    priority_queue<Node> q
    Node *root

    // Populates the priority queue
    for i from 0 to freq.size()
        if freq[i] > 0
            Node node(i, freq[i])
            q.push(node)

    if q.size() == 0 // Start building the tree
        return // Empty file
    if q.size() == 1
        root = new Node
        root->left = new Node(q.top())
    while q.size > 1
        Node *lChild = new Node(q.pop())
        Node *rChild = new Node(q.pop())
        Node parent

        parent.left = lChild
        parent.right = rChild
        parent.freq = lChild->freq + rChild->freq
        q.push(parent)
        if q.size() == 1
            root = new Node(parent)

    return root
```

```
list createDictionary(Node *root) // Helper function
    dictionary[256] // list of codes
    list code // list of ones and zeros

    createDictionary(root, code, dictionary)
    return dictionary

// Perform preorder traversal of the Huffman tree
createDictionary(Node *root, code[], &dictionary[])
    // Add code to dictionary
    if root->left == NULL and root->right == NULL
        val = root->val
        dictionary[root->val] = code
    if root->left // Append 0 to code
        code.push_back(0)
        createDictionary(root->left, code, dictionary)
        code.pop_back()
    if root->right // Append 1 to code
        code.push_back(1)
        createDictionary(root->right, code, dictionary)
        code.pop_back()
```

3 Run Time Analysis

The worst case asymptotic runtime for the Huffman coding occurs when the frequencies of symbols are the Fibonacci sequence of each over. That is, if original data had the characters "a", "b", "c", "d"... with one "a" one "b" two "c"s three "d"s... This creates a very unbalanced Huffman tree where the longest code length is equal to the number of symbols minus three and no more than two codes have the same length.

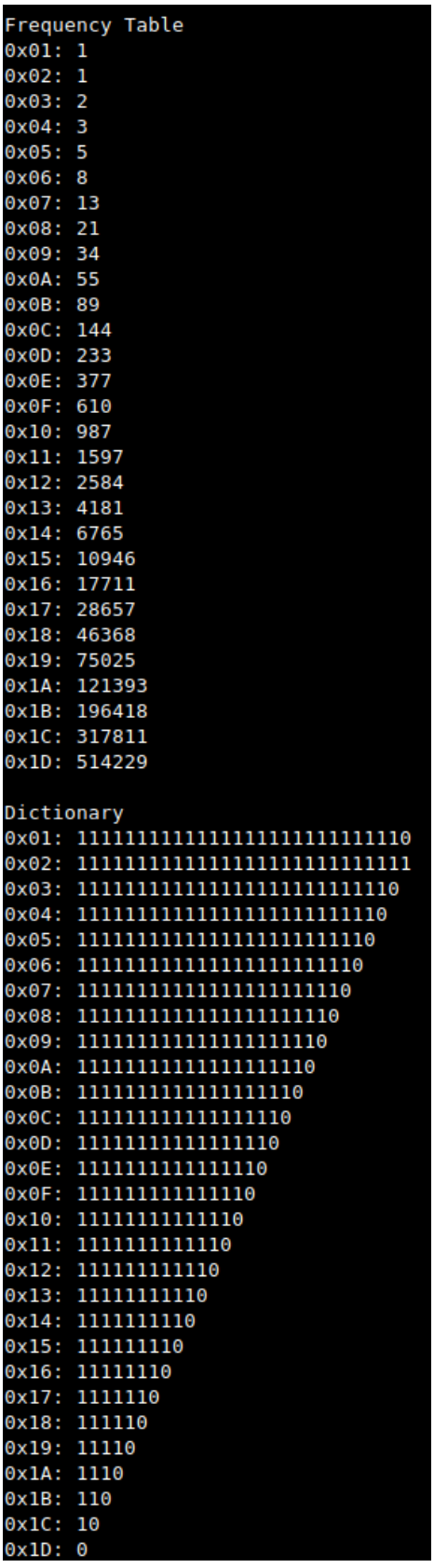

Figure 4: Fibonacci frequencies

This happens because the algorithm removes the two lowest frequency nodes from the queue and creates a new node with a frequency value equal to the sum of these two frequencies. Because the frequencies follow the Fibonacci sequence this new node's frequency will always be less than or equal to the frequency of the node after the next node. This leads to a tree where all right nodes are internal nodes with the exception of the deepest internal node's right child, which is a leaf, and all left nodes are leaf nodes. In other words, the tree is effectively a linked list with extra steps.

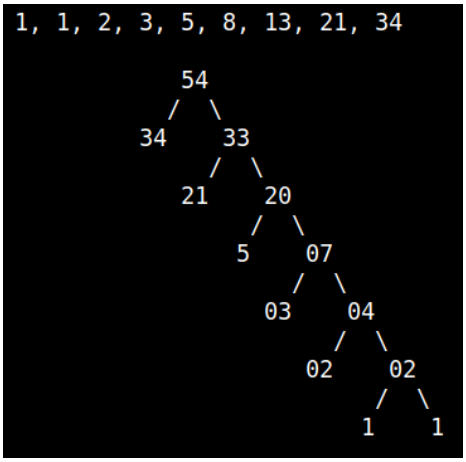

Figure 5: The most unbalanced Huffman tree

The runtime here $O(n^2)$ because traversing the tree is $O(n)$ and the tree must be traversed n times to generate the dictionary.

The best case (in terms of asymptotic run time) for Huffman coding would be a set of symbols with uniform, or nearly uniform frequencies. This would create a complete Huffman tree. Looking only at removals from the queue, removing every element from the queue is $O(n\log(n))$. $\frac{3n}{2}$ removals must be performed because for every two elements removed a new element is added and the removals have a cost of $O(\log(n))$, assuming a binary min heap based priority queue is used. Additions have a total cost of $O(n)$ in the best case. $\frac{3n}{2}$ elements must be added and if we assume frequencies are uniform, then each addition has a cost of $O(1)$. This is because the inserted element will be greater than it's parent in the heap and there will be no need heapify upward. In a more average case additions would have a cost of $O(log(n))$. Lastly, traversing the tree to generate the dictionary is $O(n\log(2n-1)) \in O(n\log(n))$ because the tree has $2n-1$ elements and traversing a complete binary tree from the root to a leaf is $O(\log(k))$ where k is the number of elements in the tree. $O(2n\log(n)+n) \in O(n\log(n))$ so the runtime is $O(n\log(n))$.

It's important to note that the "best case" described above is not necessarily the best case for the compression ratio. If the tree is complete then all codes will

be equal in length. If the algorithm is working in terms of bytes and all 256 bytes are used then the compression ratio will be negative since every code will be eight bits and the frequency table needs to be included in the compressed file so that compression can be reversible. Having a lot of variance in the the frequencies of symbols is ideal for achieving a high compression ratio because it leads to shorter codes for the more frequently occurring symbols. The Fibonacci case actually has a high compression ratio because although the codes for infrequent symbols can be longer than the original symbol, the shortness of the codes for more frequently occurring symbols more than makes up for it.

It's also important to note that in most cases, the runtime of the algorithm itself is negligible compared to the runtime of applying the compression, which is linear with respect to file size. In fact, while having a lot of variance in the frequency of symbols increasing the runtime of the algorithm it also speeds up decompression because the tree traversals used to convert the codes back to the original symbols will be shorter on average. This will have a much larger impact on runtime for a large file.

4 References

1. https://en.wikipedia.org/wiki/Huffman_coding
2. https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding
3. https://archive.org/details/fano-tr65.7z/fano-tr65-ocr/
4. https://www.huffmancoding.com/my-uncle/scientific-american