CMSC 351: Huffman Encoding

Justin Wyss-Gallifent

May 10, 2024

1 Introduction

1.1 An Introductory Example

Suppose we wish to encode the character string `HELLOOOO` (with lots of `O`s!) by assigning a binary string (called a *code word*) to each character.

There are four characters so one idea would be to assign `E`=`00`, `H`=`01`, `L`=`10`, `O`=`11`. Then we would have:

$$\texttt{HELLOOOO} = \texttt{0100101011111111}$$

We might wonder, however, if we can do this with fewer bits. What if we tried `E`=`0`, `H`=`1`, `L`=`00`, `O`=`11`? Then we would have:

$$\texttt{HELLOOOO} = \texttt{10000011111111}$$

This is 12 bits instead of 16.

However there is a problem which is that `10000011111111` could be other character strings such as `HEEEEEHHHHHHHH`.

But perhaps there is some other way that will work.

First off let's recognize what went wrong with our attempt above. One problem is that the string `0` (for `E`) is the prefix for `00` (which is `L`) so when we encounter `00` we don't know whether it should represent `EE` or `L`.

1.2 Prefix Codes and Fixed-Length Codes

First let's get some definitions down.

Definition 1.2.1. What we are trying to develop here is a *code* and the binary strings mentioned above are called *code words.*

Definition 1.2.2. A *prefix code* is a code which has the property that no code word is the prefix of another code word.

Example 1.1. Our initial code `E`=`00`, `H`=`01`, `L`=`10`, `O`=`11` is a prefix code but our second attempt `E`=`0`, `H`=`1`, `L`=`00`, `O`=`11` is not a prefix code.

Definition 1.2.3. A *fixed-length code* is a code in which each code has exactly the same length.

Example 1.2. Our initial code is a fixed-length code but our second attempt is not a fixed-length code.

It is easy to see that any fixed-length code is a prefix code but not every prefix code is a fixed-length code.

Okay, so are there any prefix codes which are not fixed-length codes?

Consider the code `H`=`000`, `O`=`1`, `L`=`01`, and `E`=`001`. This is a prefix code but not a fixed-length code.

Moreover using this code we have `HELLOOOO` = `00000101011111` which requires 14 bits, fewer than the 16 required for our fixed-length code.

2 Huffman Encoding

2.1 Introduction

Before proceeding let's reflect upon our last example. You may protest and suggest that while `HELLOOOO` uses fewer bits with our new code than with our fixed-length code, this will not be true of all character strings using this new code. For example the single character string `H` uses 3 bits instead of 2.

Thus you might ask - are there any prefix codes which are not fixed-length codes and which will result in every possible character string requiring no more bits than it would have using a fixed-length code? The answer to this is no, unfortunately.

The key point for us, though is to answer the following more specific question:

Given a specific character string, can we construct a prefix code which has the property that this code will result in our character string using the minimum number of bits possible amongst all codes?

The answer to this is yes!

2.2 Algorithm

It makes sense that when encoding a character string we should identify the characters which occur most frequently and the code should assign those characters a code word with as few bits as possible.

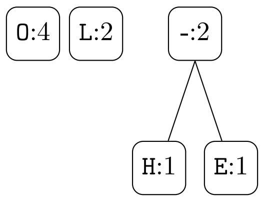
In light of that we first sort our characters by count. Let's do this with `HELLOOOO`. Note that `H` and `E` have the same count and could be in either order.

Character	Count
`O`	4
`L`	2
`H`	1
`E`	1

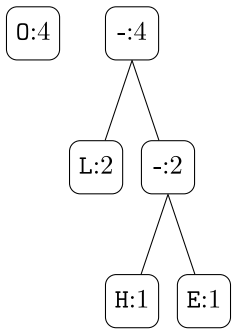
We start by creating a binary tree for each letter. Each will be simply a root node and each has a value assigned to it which is the character count.

$$\boxed{\texttt{O}:4}\ \boxed{\texttt{L}:2}\ \boxed{\texttt{H}:1}\ \boxed{\texttt{E}:1}$$

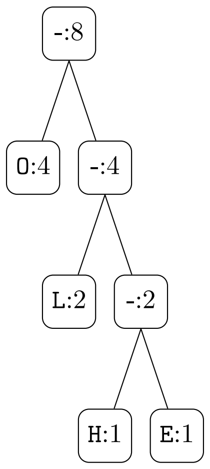
Next we pick the two binary trees with the lowest total count and combine them under a new parent root node. We assign that root (and the entire tree) a count equal to the sum of the counts of the two binary trees:



We then repeat, noting that the two binary triees with the lowest total count are the one containing the L and the one we just built:

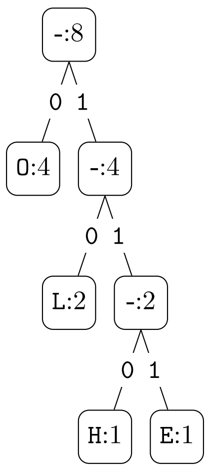


Then one last time:



Observe that the most frequent characters are closer to the top of the tree because they were picked up later on in the process.

We then label the branches with 0 on the left and 1 on the right:



We then read the encoding from the tree in that the binary string leading to each character will be the code word for that character. Here is the table including the counts:

Character	Count	Code Word
`O`	4	`0`
`L`	2	`10`
`H`	1	`110`
`E`	1	`111`

Then we have the final result requiring 14 bits:

$$\texttt{HELLO} = \texttt{11011110100000}$$

2.3 (Non)Uniqueness

Observe that the encoding is not unique for several reasons. There may be trees (even single characters) with equal counts and there may be more than one way to pick two trees with minimum total count. Moreover when we combine two trees to form a new tree it doesn't really matter which goes left and which goes right, meaning which gets assigned a branch value of `0` and which gets assigned a branch value of `1`.

2.4 Greediness

Observe that the Huffman algorithm is a greedy algorithm in that at each iteration it makes the best possible choice without knowing what will happen in future iterations.

Even though the algorithm is greedy, which does not in general guarantee an optimal result, we shall see that it does turn out nicely here.

3 Time Complexity

3.1 Introduction

There are two processes we are interested in here. First is the time complexity of building the tree and second is the time complexity of extracting the codes.

3.2 Tree Building

The most efficient way to construct the final binary tree is as follows:

```
identify each of the n starting binary trees with
their count and use this as a value to construct
a min heap H where each node is a count-tree pair.
for i = 1 to n-1:
    extract the binary tree t1 with the smallest count from H
    rebuild H
    extract the binary tree t2 with the smallest count from H
    rebuild H
    combine t1 and t2 to form the tree t
    insert t into H
end for
```

It takes time $\mathcal{O}(n \lg n)$ to construct H (this can be done in $\mathcal{O}(n)$ but the math is less obvious and it does not affect our final result). The loop iterates $n-1$ times and for each iteration the extraction and rebuilding takes time $\mathcal{O}(\lg n)$ (not Θ since the heap is getting smaller).

Consequently the overall time complexity is:

$$\mathcal{O}(n \lg n) + (n-1)\mathcal{O}(\lg n) = \mathcal{O}(n \lg n)$$

3.3 Code Extraction

Before proceeding, a small theorem:

Theorem 3.3.1. Suppose a binary tree has the property that each node has either 0 or 2 children - called a *full binary tree.* If N is the number of nodes and L is the number of leaves then we have $N = 2L - 1$.

Proof. An outline of the proof is as follows, a more rigorous proof would use structural induction. such a binary tree can be constructed by starting with a single node and then progressively picking a leaf and adding two children to it, then repeating. Each time we pick a leaf and add two children we increase N by 2 and increase L by 1, thus we have N following the pattern $1, 3, 5, 7, 9, ...$ and L following the pattern $1, 2, 3, 4, 5, ...$ and we see $N = 2L - 1$. $\mathcal{QED}$

Our approach to code extraction will be to construct an object which contains the characters and their codes. We will do this in one pass through the tree: meaning we will visit each node exactly once.

The pseudocode is very simple:

```
function extractcodes(node nd,binary string s):
    if nd is a leaf:
        object[leaf character] = s
    else:
        extractcodes(nd.leftchild,s+'0')
        extractcodes(nd.rightchild,s+'1')
    end if
end function
global object = {}
extractcodes(root,'')
```

Whenever `extractcodes` encounters a leaf node the current `binary string s` will be the associated binary string along the path to that node. Consequently the string will be assigned to the `leaf character` stored in the leaf.

Suppose we have n characters in our binary tree. Since each of those n characters corresponds to a leaf, by our theorem earlier there are $2n-1$ nodes in the tree and hence our pseudocode has time complexity $\Theta(2n-1) = \Theta(n)$.

4 Minimality Proof

Note 4.0.1. In all of the above we have used the count of the characters rather than the frequency. This is because the count is always an integer whereas the frequency can be an awkward fraction. In what follows, however, we will switch to frequency for easier calculation. Note that this does not affect anthing in the algorithm since higher count corresponds to higher frequency.

Intuition suggests that the prefix code constructed this way be efficient, meaning it should use very few bits. This is because characters which occur infrequently end up in trees which get picked repeatedly by the algorithm and consequently end up lower in the final tree, thereby requiring more bits, whereas characters which occur frequently end up higher in the tree, requiring fewer bits.

More formally though we have the following:

Theorem 4.0.1. A prefix tree constructed via the Huffman algorithm yields a minimal prefix code for the original character string. That is, using the prefix code resulting from the prefix tree grown by the Huffman algorithm, the total number of bits used to encode the character string is as small as possible.

We break the proof up into a series of steps:

(a) **Notation:**

In what follows we assume we have a given string. Let A be the set of characters in the string. For each $x \in A$ let $f(x)$ be the frequency of x in the string.

(b) **Definition:**

Suppose T is a binary tree such that each $x \in A$ corresponds to a leaf in T. For each $x \in A$ let $d(x, T)$ be the depth of x in the tree T. Define:

$$s(T) = \sum_{x \in A} f(x) d(x, T)$$

Observe that minimizing s is equivalent to minimizing the total number of bits used.

(c) **Swap Lemma:**

Suppose T is a binary tree such that each $x \in A$ corresponds to a leaf in T. Let $y, z \in A$ and let T' be the tree obtained by swapping y and z. Then:

$$s(T') - S(T) = (f(y) - f(z))\,(d(z,T) - d(y,T))$$

Proof:

We have the following, noting that because of the swap we have $d(y, T') = d(z, T)$ and $d(z, T') = d(y, T)$:

$$\begin{aligned} s(T') - s(T) &= \sum_{x \in A} f(x) d(x, T') - \sum_{x \in A} f(x) d(x, T) \\ &= [f(y)d(y,T') + f(z)d(z,T')] - [f(y)d(y,T) + f(z)d(z,T)] \\ &= [f(y)d(z,T) + f(z)d(y,T)] - [f(y)d(y,T) + f(z)d(z,T)] \\ &= f(y)\,(d(z,T) - d(y,T)) + f(z)\,(d(y,T) - d(z,T)) \\ &= f(y)\,(d(z,T) - d(y,T)) - f(z)\,(d(z,T) - d(y,T)) \\ &= (f(y) - f(z))\,(d(z,T) - d(y,T)) \end{aligned}$$

(d) **Optimal Lemma:**

There exists an optimal tree (minimal $s(T)$) such that two characters with the lowest frequencies are siblings and are at the deepest level of the tree.

Proof:

There are only finitely many possible trees so clearly there is one, call it T, with minimal $s(T)$. Choose y to be a character with lowest frequency and maximum depth and z to be chosen similarly after y. Proceed as follows:

First, if y is an only child then create T' by removing y and assigning y to the parent. Then $s(T') < s(T)$, a contradiction, so we can assume y has a sibling.

Second, if y is not at the deepest level of the tree then there exists a node w with $d(w, T) > d(y, T)$. Create T' by swapping w and y. Then by the Swap Lemma we have:

$$s(T') = s(T) + (f(w) - f(y))\,(d(y,T) - d(w,T))$$

Since $f(w) > f(y)$ (since y and z have lowest frequencies and y is, of those, deepest) and $d(w,T) > d(y,T)$ we have $s(T)' < s(T)$, a contradiction, so we can assume y is at the deepest level of the tree.

Third, if z is the sibling of y we are done, otherwise assume $w \neq z$ is the sibling of y. Create T' by swapping w with z. Then by the Swap Lemma we have:

$$s(T') = s(T) + (f(w) - f(z))\,(d(z,T) - d(w,T))$$

Since $f(w) \geq f(z)$ (since y and z have lowest frequencies) and $d(z,T) \leq d(y,T)$ (assumed) and $d(y,T) = d(w,T)$ (because they are siblings) we have $s(T') \leq s(T)$.

However since T is optimal we cannot have $<$ and hence $S(T') = S(T)$ and so T' is also optimal so we replace T with T'.

(e) **Proof of Theorem:**

The proof is by induction on n, the number of different characters in A.

Clearly the theorem is true for $n = 1$ so let us assume it is true for $n = k$ and we will show it is true for $n = k + 1$.

Let A be an alphabet with $k + 1$ letters and frequency function f. Let H_A be the prefix tree resulting from applying the Huffman algorithm to A and f, Let y, z be the two characters chosen first by the algorithm.

We know by the Optimal Lemma that there is an optimal tree OPT_A such that y, z are deepest siblings. Note that the Lemma doesn't guarantee exactly that but it guarantees that two characters with minimal frequencies are deepest siblings so we can just swap those two with y, z without changing $s(OPT_A)$.

We claim that $s(H_A) = s(OPT_A)$.

Define the alphabet $B = A - \{y, z\} \cup \{\alpha\}$ where α is some new character and with $f(\alpha) = f(y) + f(z)$.

Define OPT_B to be the tree obtained by removing $\{y, z\}$ from OPT_A and assigning α to their parent. Note that:

$$\begin{aligned}
s(OPT_A) &= \sum_{x\in A} f(x)d(x,A) \\
&= f(y)d(y,A) + f(z)d(z,A) + \sum_{x\in A-\{y,z\}} f(x)d(x,A) \\
&= f(y)d(y,A) + f(z)d(y,A) + \sum_{x\in A-\{y,z\}} f(x)d(x,A) \\
&= f(\alpha)d(y,A) + \sum_{x\in A-\{y,z\}} f(x)d(x,A) \\
&= f(\alpha)(d(\alpha,A)+1) + \sum_{x\in A-\{y,z\}} f(x)d(x,B) \\
&= f(\alpha) + f(\alpha)d(\alpha,A) + \sum_{x\in A-\{y,z\}} f(x)d(x,B) \\
&= f(\alpha) + \sum_{x\in B} f(x)d(x,B) \\
&= f(y) + f(z) + \sum_{x\in B} f(x)d(x,B) \\
&= f(y) + f(z) + s(OPT_B)
\end{aligned}$$

Define H_B to be the tree obtained by removing $\{y,z\}$ from H_A and assigning α to their parent. Note that $s(H_A) = s(H_B) + f(y) + f(z)$ by a similar calculation as the above.

In addition consider that when Huffman is applied to A and f we chose y and z first and combined them to form a new tree with frequency $f(y)+f(z)$ and then continue with Huffman. The remaining process is algorithmically equivalent to starting with the alphabet B and $f(\alpha) = f(y)+f(z)$ and hence H_B can be thought of as the result of applying the Huffman algorithm to B and f, and hence since B has k characters we have $s(H_B) = s(OPT_B)$ by the inductive assumption.

Then we have:

$$s(H_A) = S(H_B) + f(y) + f(z) \leq s(OPT_B) + f(y) + f(z) = s(OPT_A)$$

Since OPT_A is optimal for A we then must have $s(H_A) = s(OPT_A)$ and so the tree created by the Huffman algorithm is optimal.