Huffman Codes

Robert T. Lange, Nandan Rao, Euan Dowers

1 Prefix Codes

Given an alphabet A with letters $a \in A$, define a *code*, c, as an injective function c where x is replaced by a variable-length binary string $c(x)^1$. Define a *prefix code* as a code such that for all $x, y \in A$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$. Therefore, given the code c a prefix code can be decoded reading left to right.

Given a text T that uses an alphabet A, for each $x \in A$ let f_x denote the frequency of x in T. The average number of bits needed to encode T is therefore given by

$$C = \sum_{x \in A} f_x |c(x)|.$$

Our aim is to find a prefix code for T that minimises C. This problem was solved by David A. Huffman in 1952, when he was still a Ph.D student [1].

2 Algorithm

2.1 Introduction and Intuition

As previously mentioned, the goal of the Huffman algorithm is to minimize the number of bits required to encode a given text, by optimizing our choice of binary prefix for each character in the text's alphabet. Thinking intuitively, it is easy to imagine that one could possibly achieve short prefix codes for frequent characters at the cost of lengthening the prefix codes for infrequent characters. It is easy to see that this is exactly what the above-stated cost function achieves mathematically, and keeping that intuition in mind will make it clear how this algorithm garauntees a minimum number of encoded bits.

The Huffman algorithm achieves its stated goal by building a binary search tree where leaves are made up of the characters in the represented alphabet. "Directions" to a particular leaf, repsenting a character in the alphabet, consist of the binary prefix of that character (where, for example, we take 0 to mean "take the left branch" and 1 to mean "take the right branch").

The length (and correspondingly the number of bits) of any given prefix code is therefore directly determined by its depth in the tree. Following our previous line of intuitive thinking, We would expect frequently-used characters, therefore, to be located nearer to the root of the tree, and less-frequently-used characters to be further away.

The process of decoding, now framed as the process of following directions to a particular leaf, can be easily analogized with the process of searching in a binary search tree (although it should be clarified that the trees are not equivalent). In optimizing, we seek to balance the frequency, or "weight" of both edges extending out of every non-leaf node. This will minimize, after repeated "searches", the average depth travelled before finding the leaf, and therefore the total cost of "searching". In our case, this translates directly to minizing the length of our encoded "directions".

---

[1] i.e. $c : A \to \mathbb{Z}_1^2 \cup \mathbb{Z}_2^2 \cup \ldots \cup \mathbb{Z}_n^2$ for some n, and $c(x) = c(y) \Rightarrow x = y$.

The first step of the algorithm involves sorting, from low to high, all characters by their frequency, f_x, and placing them in a "node list". At each stage of the algorithm, we select the two nodes with the lowest weights, and combine them into a subtree. This subtree now consists of two child nodes we can call f_a and f_b, and one newly-created parent node. We give the parent node a frequency equal to $f_a + f_b$, and place it in the node list. We then resort the node list by frequency, and repeat.

In this way the algorithm moves through its sorted node list, two nodes at a time, and builds the binary tree from the bottom-up. This algorithm can, for this very reason, be described as a Greedy Algorithm, as it "greedily" takes the lowest-frequency pair of nodes at each step, without looking ahead at the rest of list, and relies on the assumption that if every pair of nodes in a subtree is as-close-to-equally weighted as possible, then the tree will come out being as-close-to-equally weighted as possible, and therefore optimal.

2.2   Encoding

The algorithm we have implemented in encoding a string, as is common in the literature [1][2] is as follows

---
Algorithm 1 Huffman Coding Algorithm

1: **procedure** LETTER COUNT(text)
2:     $A \leftarrow$ letters used in *text*
3:     *nodes* $\leftarrow$ empty list
4:     **for** character in A **do**
5:         add *node* with *character* and number of occurences of *character* in *text* to *nodes*
6:     sort *nodes* by frequency from low to high

7:
8: **procedure** HUFFMAN TREE(nodes)
9:     **while** length of *nodes* > 1 **do**
10:        $freq \leftarrow frequency(nodes[0]) + frequency(nodes[1])$
11:        $parent \leftarrow$ Tree(frequency = *freq*, left child = $nodes[0]$, right child = $nodes[1]$)
12:        remove $nodes[0]$ and $nodes[1]$ from *nodes*
13:        add *parent* to *nodes*
14:        sort *nodes* by frequency from low to high
15:    $tree \leftarrow nodes[0]$

16:
17: **procedure** HUFFMAN ENCODE(*text, tree*)
18:    *codedmessage* $\leftarrow$ string of length 0
19:    **for** letter in text **do**:
20:        *searchtree* $\leftarrow$ empty list
21:        add *tree* with *codeword* as empty string to *searchtree*
22:        **for** node in searchtree **do**
23:            **if** *node* has children **then**
24:                add leftchild(*node*) to *code nodes* with *codeword* of *node* + '0'
25:                add rightchild(*node*) to *code nodes* with *codeword* of *node* + '1'
26:                remove *node* from *searchtree*
27:            **else**
28:                **if** *character* of *node* = *letter* **then**
29:                    *codedmessage* $\leftarrow$ *codedmessage* + *codeword* of *node*
30:        *codedmessage* $\leftarrow$ *codedmessage* + *codedletter*

31:
32: **return** codedmessage

2.2.1 Proof of Correctness

Assume we are given a text T that uses some alphabet A and we wish to find the optimal prefix code for T. Note first that any binary code can be represented by a binary tree, and that any prefix code can be represented as a binary tree where every node that corresponds to a letter is a leaf node. The reasoning for this assertion is straightforward: starting from a root node, and appending a '0' for any subsequent left branch, and a '1' for any subsequent right branch, the code for any letter can be represented by the path taken to reach the node corresponding to that letter in our binary tree. Furthermore, since c is a prefix code, we know that for any letter x the codeword $c(x)$ cannot appear as the prefix of another codeword, so x must be a leaf node of our tree. An example of a (clearly suboptimal) prefix code for the text `hello` is given in Figure 2.1, which will give a coded message of `000001110110111`.
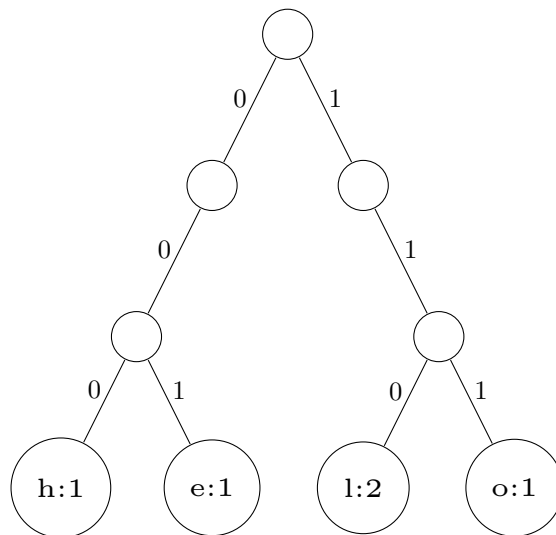

Figure 2.1: Suboptimal prefix code represented as a binary tree

It is clear that any optimal prefix code will not have any redundant digits, as the code in Figure 2.1 does (in the second digit of every codeword). This is equivalent to saying that all nodes other than leaf nodes must have two children. Therefore, any optimal prefix code must follow some kind of bottom-up pairing approach as in Algorithm **REF**.

At each iteration of the node parenting step in the algorithm described in the previous subsection, the algorithm is essentially adding $f_{(1)} + f_{(2)}$ bits to the length of the final encoded message, where $f_{(i)}$ represents the frequency of the i-th least frequent node in our list. Therefore, since we sort the list of available child nodes at the end of each iteration, we are adding the smallest possible length to our encoded message in each iteration.

Therefore, Algorithm **REF** defines the optimal prefix code defined by such a bottom up pairing algorithm, and since we have shown that any optimal prefix code must be able to be generated by such an algorithm, Algorithm **REF** defines an optimal prefix code.

2.2.2 Complexity

We will break down our encoding algorithm per procedure, as named in **REF**. We will signifiy the length of the text as N and the number of unique characters in the text as M. We will address time and space complexity in each procedure:

Letter Count   This step consists for traversing our entire text and counting the frequency of each character, which is of complexity O(N) in the length of the text. It then consists of sorting the list of characters, which

following general sorting conventions can be achieved in O(M log M) in the length of characters. In the typical case where $M << N$, we would expect this entire procedure to be linear in the length of the text, however in the worst case where $M = N$, this sorting would be log-linear in terms of the length of input text, giving an overal complexity of this procedure of O(N log N). As we are storing a list of pairs, letter:count, we consider the space complexity to be linear in M, $O(M)$.

Huffman Tree This stage consists of building a tree from the list of unique characters in the Alphabet of the text. Every step of this operation replaces two elements of the list with a single element, and the operation ends when there is only a single element left in the list. The number of steps is therefore equal to the number of characters in the alphabet, less one.

In a naive implementation, one must re-sort the list in every step, however it is easy to see that one does not need to fully re-sort an entire list when one adds one new element to an already-sorted list. This re-sort can, therefore, be achieved in linear time in the length of the list (at the time of the step).

As the length of the list is continually decreasing, and each step in the list involves a sort that takes linear time with respect to the list, we consider this procedure triangular in the number of unique characters in the text, $O(M(M + 1)/2))$. The space complexity here is also linear in M, as is common for binary trees, which can be natural to represent as a linked-list.

Huffman Encode Here we encode the text, with a step for each character in our text, and is therefore at least linear in N, the length of the input text. In every step, however, we are performing a breadth-first search of a tree to find our character. Because of the nature of our decisions in making the tree, this amounts to an optimized binary search tree, with the most frequent characters being found closest to the root. In the worst case, however, where $M = N$ and all characters occur with equal frequency, we could need to search the entire tree on every step.

Searching a full tree means that we have an operation for each edge, and one for each vertex, leading to complexity linear in the number of nodes and vertices, $O(|E| + |V|)$, which because of the strict binary-tree representation, our edges are never more than $(M + 1)/2$, so this search operation will be linear in the number of characters in our alphabet, $O(M)$.

Putting the two steps together with the linear operation over the text, we have a procedure that operates in $O(M * N)$ time. Again, in the usual case where $M << N$, this can be considered linear in N. However, in the worst case, where $M = N$, and where every character occurs with equal same frequency, this procedure must be considered $O(N^2)$. The space complexity consists of the tree from the previous procedure, plus the encoded text. In the worst case where $M = N$, the expected length of each encoded character, again represented as its depth on the tree, will be log M. In this case, the additional space complexity created by this step is O(N log N), the worst-case length of the encoded string.

Total Complexity Adding up all the procedures, we have an expected time complexity of $N + (M * N)$, which is considered $O(M * N)$. In the worst case where $M = N$, this procedure is therefore quadratic in N, the length of the text, however as stated previously, in the usual case where $M << N$, this can be considered linear in in the length of the text.

2.3 Decoding

With the described encoding process, decoding becomes quite simple:

Algorithm 2 Huffman Decoding Algorithm

1: **procedure** Huffman Decode(encodedtext, tree)
2:     $s \leftarrow$ minimum length of all codewords for all leaf nodes in *tree*
3:     $A \leftarrow$ empty string
4:     **while** length of *encodedtext* > 0 **do**
5:         $directions \leftarrow encodedtext[0]$
6:         $node \leftarrow tree$
7:         i $\leftarrow 0$
8:         **for** step in directions **do**
9:             $node \leftarrow$ child of number *step* from *node*
10:            **if** *node* is leaf **then**
11:                $A \leftarrow A + character$ from leaf
12:                $encodedtext \leftarrow encodedtext[i : end]$
13:                $i \leftarrow 0$
14:            **else**
15:                $i \leftarrow i + 1$
16:                add *encodedtext[i]* to directions
        return A

2.3.1 Proof of Correctness

The proof of correctness of the decoding follows directly from the construction of our binary tree. We decided that every character in our alphabet would be stored in a leaf of the tree, with zero children. Also, as previously mentioned, the encoding of a particular character is decided to be the "directions" to get to that character from the root of the tree.

From this it follows quite simply that our prefix codes are truly prefix codes, as we will never reach a character (a leaf) "on the way" to a different character (another leaf), as that defies the definition of a leaf in a tree.

2.3.2 Complexity

References

[1] David A. Huffman. *A Method for the Construction of Minimum-Redundancy Codes*. Proceedings of the I.R.E. 40(9), 1952.

[2] David J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.