Sample Final Report: Huffman Coding

Suhas Arehalli

COMP221 SP25— Arehalli

1 Introduction

Many computer applications involve transmitting information over a network under circumstances the size of transmissions is limited, or where larger transmissions are more costly. As a result, the task of reducing the size of the transmission used to represent some information — *data compression* — is critical.

In some applications, like photo or video compression, some amount of information loss is acceptable, since small changes in a photo or video can be tolerated by the recipient. This is a use case for *lossy* compression algorithms, which encode and decode messages with some acceptable tolerance for deviance between the original and decoded message. In other settings — for example, text compression — small changes to the data are unacceptable. In this second set of cases, *lossless* compression algorithms — those that enforce that the original and decoded messages are identical — are required.

Huffman Coding is one of these lossless data compression algorithms. In practice, it operates by assigning every symbol in the message (i.e., a letter in a text compression setting) a variable-length binary representation based on its relative frequency within the space of possible messages. This resulting code table is then used to encode messages, with the guarantee that the length of the *average* encoding is minimized.

2 Problem Statement

We must introduce some formal structure before getting to the problem description:

A message is a string of symbols $x_1 \dots x_n$, with each symbol drawn from some finite alphabet Σ (i.e., $x_i \in \Sigma$). We call the space of possible messages Σ^*.

Huffman coding is a type of compression scheme that uses a **code table**: a one-to-one map $e : \Sigma \to \{0,1\}^*$ that gives us, for each symbol in a string $c \in \Sigma$, a binary string $e(c)$ that encodes it. Since the map is one-to-one, each symbol maps to a *unique* binary string to facilitate decoding.

To encode a text with a code table, we simply map each symbol c to it's corresponding binary code $e(c)$ and concatenate them together. Formally, if we let $\oplus$ represent string concatenation, we have

$$e(x_1 \dots x_n) = e(x_1) \oplus \dots \oplus e(x_n)$$

A code table e is called *prefix-free* if no symbol has a code that is a *prefix* of some other symbol's code. Formally, for a code table e for an alphabet Σ, $\forall c_1, c_2 \in \Sigma$ where $c_1 \neq c_2$, $e(c_1)$ is not a prefix of $e(c_2)$. If a code table e is prefix-free, we have a simple decoding algorithm: keep scanning until we see a code that corresponds to a symbol in our code table, and then decode that symbol (Alg. **??**).

Now we can formally define the problem of finding an optimal code table e for a given set of symbols and their frequencies!

Problem Statement (Prefix-Free Coding)**.**
Input: an alphabet Σ and a map from symbols to frequencies $f : \Sigma \to \mathbb{R}$
Output: a prefix-free coding table e such that the expected length for the encoding of a message $x_1 \dots x_n$,

$$\mathbb{E}\big[|e(x_1 \dots x_n)|\big]$$

is minimized.

3 Huffman Coding

Huffman Coding models the process of assigning prefix-free codes to symbols from our alphabet

Algorithm 1 A decoding scheme given a prefix-free coding table

```
function DECODE(b_1 ... b_n ∈ {0,1}*, e : Σ → {0,1}*)
    out ← ε
    current ← ε
    for i ← 1 to n do
        current ← current ⊕ b_i
        if exists c such that current = e(c) then
            out ← out ⊕ c
        end if
    end for
end function
```

as building a binary tree where a symbol's code corresponds a path from the root to a special leaf node representing that symbol. We derive a binary code from the path by converting each edge in that path to a 0 if it connects from a parent to a left child, and a 1 if it connects a parent to a right child. Since these paths are always from a root to a *leaf*, a code can never be a prefix of another, ensuring that the code table corresponding to the tree is prefix-free.

Given that, we begin with a forest $T = (V, E)$ with vertices $V = \{(c, f(c)) \mid c \in \Sigma\}$, representing a symbol and its frequency, and $E = \emptyset$. These will be the eventual leaves of our tree, so we will iteratively merge them by creating internal nodes and assigning those internal nodes the roots of two existing trees as children. Each internal node represents all of the symbols represented by its children, so it's frequency label will be the sum of the frequency labels of its children.

Since leaves closer to the root will be assigned shorter codes, we want to merge high-frequency nodes as late as possible in hopes that they end up close to our root. To achieve this, we will adopt a *greedy strategy*, always selecting the two lowest frequency subtrees remaining. Pseudocode is provided in Alg. **??**.

4 A Worked Example

Consider the alphabet $\Sigma = \{a, b, d, f\}$ with corresponding frequencies $c_a = 0.7$, $c_b = 0.2$, $c_d = 0.05$, and $c_f = 0.05$.

The algorithm would proceed as follows, summarized in Figure **??**: First, we construct a priority queue containing a tree for each symbol. This queue will store the roots of our forest. Then we merge the two lowest frequency characters, *d* and *f*. We then repeat our greedy merging: *d/f* and *b*, then *a* and *b/d/f*. This results in a full Huffman Coding tree. We then construct a coding table from the tree, assigning a representation of the path to the node labeled $c \in \Sigma$ from the root to $e(c)$ using BUILDCODETABLE. This will result in *a* getting the code 0, *b* getting the code 10, *d* getting the code 110, and *f* getting the code 111.

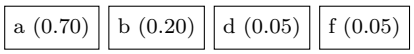
Now we can measure the efficiency of the resulting code table by computing the expected length of the encoding of a string of length n. We apply properties of the expectation $\mathbb{E}[\cdot]$ to find

$$\begin{aligned}
\mathbb{E}\big[|e(x_1 \dots x_n)|\big] &= \mathbb{E}\big[\sum_{i=1}^{n} |e(x_i)|\big] \\
&= \sum_{i=1}^{n} \sum_{c \in \Sigma} |e(c)| p(c) \\
&= n(1(0.7) + 2(0.2) + 3(0.05) + 3(0.05)) \\
&= 1.4n
\end{aligned}$$

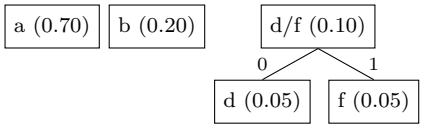
Thus, we expect a string of length n to have an encoding of length $1.4n$. Note that a simple fixed-length code would assign each of our 4 characters a 2-bit string, leading to an encoding

Algorithm 2 The Huffman Coding Algorithm

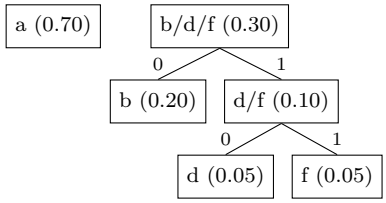
function HUFFMAN(alphabet Σ, frequencies $f : \Sigma \to \mathbb{R}$)
  Let Q be a priority queue of nodes sorted by frequency.
  For each $c \in \Sigma$, enqueue into Q a node with $freq(n) = f(c)$ and $label(n) = c$
  while $|Q| > 1$ **do**
    $l, r \leftarrow Q.removeMin(), Q.removeMin()$ ▷ Dequeue the 2 lowest frequency trees
    Let n be a node with children l and r ▷ Merge
    $freq(n) \leftarrow freq(l) + freq(r)$
    Enqueue n into Q
  end while
  $root \leftarrow Q.removeMin()$
  Let e be an empty code table.
  return BUILDCODETABLE($root$, ε, e) ▷ Convert the tree to a code table via a traversal
end function
function BUILDCODETABLE(Tree $root$, binary string b, Code table e)
  if $root$ has no children **then**
    $e(label(root)) \leftarrow b$
  end if
  BUILDCODETABLE($leftChild(root)$, $b \oplus 0$, e)
  BUILDCODETABLE($rightChild(root)$, $b \oplus 1$, e)
end function



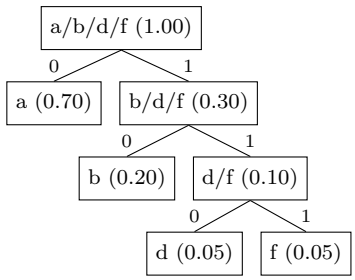
(a) Initial forest before any merges



(b) The forest after the first merge of d/f.



(c) The forest after the second merge of b/d/f.



(d) The tree after the final merge.

Figure 1: The construction of the Huffman Coding tree in 3 merges.

of length n having length $2n$, so on average, the Huffman code is more efficient!

5 Runtime Analysis

Let n be the number of symbols in Σ. First observe that initializing our priority queue can be done in $O(n \log n)$ time as written, but can be done in $O(n)$ time by an efficient heap construction algorithm (though this will not matter in the end). The while loop will iterate $n-1$ times, since a tree with n leaves has $n-1$ internal nodes. Each iteration is dominated by removing 2 items from our priority queue and inserting another, each of which is $O(\log n)$, resulting in the while loop taking $O(n \log n)$ time. Finally, BUILDCODETABLE is simply a tree traversal, and thus with $O(n)$ nodes in the tree, will run in $O(n)$ time. Thus, the algorithm is dominated by the while loop, resulting in a total $O(n \log n)$ time complexity.

6 Proof of Correctness

Observe that the correctness of the decoding algorithm follows directly from the code being prefix-free. As a result, we'll assume that encoding and decoding is lossless, and will instead focus on proving the optimality of the resultant coding table.

Statement. *Let e be a prefix-free coding table generated by the Huffman Coding algorithm. For any prefix-free coding table e',*

$$\mathbb{E}\big[|e(x_1 \dots x_n)|\big] \leq \mathbb{E}\big[|e'(x_1 \dots x_n)|\big]$$

We prove this via an exchange argument:

Proof. Since every prefix-free code can be represented by a binary tree of the sort constructed during Huffman coding, we will prove that each merge we make will construct a subtree of the tree representing the optimal coding table.

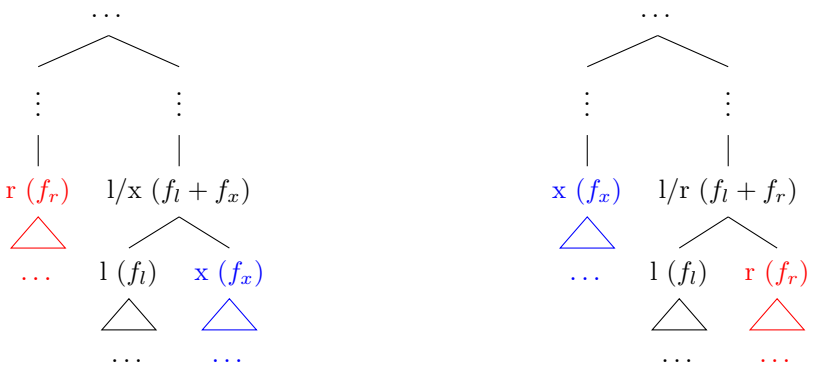
Proceed by induction over merge decisions.

Base Case: Before the first iteration, all trees are single leaves, and thus are all trivially subtrees of an optimal coding table's tree representation.

Inductive Step: Assume via our inductive hypothesis that all trees in our forest are subtrees of an optimal coding table's tree representation. We will show the next merge will preserve this property. To see this, assume for contradiction that our merge of the two lowest frequency trees, rooted at l and r, into a tree rooted at a node labeled l/r, is not a subtree of any optimal coding tree.

Consider an optimal coding tree T^* that contains both the subtrees l and r. Observe that this optimal tree must exist given our inductive hypothesis, and does not merge l/r given the assumption we made for contradiction. Let x be the subtree that was merged with l in T^*. Now we construct the tree T'^* by exchanging the positions of the subtrees rooted at x and r. We claim that this tree will be just as optimal, if not more optimal than T^*. This construction is schematized in Fig. **??**.

To confirm this claim, we'll show that our exchange maintains or lowers the expected encoding length. Let f_r, f_x represent the total probabilities (i.e., normalized frequencies) of the characters rooted at r and x respectively and let d_r, d_x represent the depth of the root of each of the subtrees in T^*. Since leaves' positions within subtrees are unaffected by the swap, let d_c^n represent the depth of the leaf containing character $c \in \Sigma$ within the subtree rooted at n. It follows that the length of the code of a character c under subtree rooted at r in T^* is $d_r + d_c^r$. Since we swap the positions of r and x in T'^*, the length of that same character would be $d_x + d_c^r$ using T'^*. Finally, let e and e' be the encoding tables represented by T^* and T'^* Now we compute the difference in expected string lengths, $\Delta\mathbb{E} = \mathbb{E}\big[|e(x_1 \dots x_n)|\big] - \mathbb{E}\big[|e'(x_1 \dots x_n)|\big]$. Since the only characters whose codes change are those



(a) The tree T^*, pre-exchange

(b) The tree T'^*, post-exchange

Figure 2: A diagram of the structure of our exchange at the step where we merge l and r

in the subtrees labeled r and x, this will be

$$\begin{aligned}
\frac{\Delta\mathbb{E}}{n} &= \sum_{\substack{c\in\Sigma \\ \text{under } x}} (|e(c)| - |e'(c)|)p(c) \\
&\quad + \sum_{\substack{c\in\Sigma \\ \text{under } r}} (|e(c)| - |e'(c)|)p(c) \\
&= \sum_{\substack{c\in\Sigma \\ \text{under } x}} (d_x + d_c^x - d_r - d_c^x)p(c) \\
&\quad + \sum_{\substack{c\in\Sigma \\ \text{under } r}} (d_r + d_c^r - d_x - d_c^r)p(c) \\
&= (d_x - d_r) \sum_{\substack{c\in\Sigma \\ \text{under } x}} p(c) \\
&\quad + (d_r - d_x) \sum_{\substack{c\in\Sigma \\ \text{under } r}} p(c)
\end{aligned}$$

Since the frequency of a node n is the sum of all of the frequencies of the characters under that node, and $p(c)$ is proportional to its $f(c)$, we know $f_n \propto \sum_{\substack{c\in\Sigma \\ \text{under } n}} p(c)$ for every node n. This gets us

$$\begin{aligned}
\frac{\Delta\mathbb{E}}{n} &\propto (d_x - d_r)f_x + (d_r - d_x)f_r \\
&\propto f_x d_x - f_x d_r + f_r d_r - f_r d_x
\end{aligned}$$

And so e', the encoding table derived from T'^* is has a smaller or equal expected encoding length if $f_x d_x - f_x d_r + f_r d_r - f_r d_x \geq 0$

To show this, we make 2 claims: First, that $f_r \leq f_x$. This follows from the fact that we adopt a greedy strategy for merges, and selected l and r before x. Second, we claim that $d_r \leq d_x$. This follows from the fact that r is merged into the binary tree T^* *after* l and x are merged; At the earliest, l/x was immediately merged with r, making the depth of $d_r \leq d_x - 1$. Given these two inequalities, we have that

$$\begin{aligned}
(f_x - f_r)(d_x - d_r) &\geq 0 \\
f_x d_x - f_r d_x - f_x d_r + f_r d_r &\geq 0
\end{aligned}$$

And thus the expected string length from an encoding given T'^* is less or equal to than that given T^*. However, since T^* was constructed to be an optimal coding tree, this contradicts the fact that we assume no optimal coding tree would allow our merges up to l and r. Thus, an optimal solution contains all prior merges plus one merging l and r.

To conclude, we observe that after we construct a single tree, the optimal solution must include that full tree as a subtree, and thus our tree must represent an optimal coding table. □