CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Greedy 3: Huffman Codes

Instructor: Xiao Liang[1]

Department of Computer Science and Engineering
Chinese University of Hong Kong

[1]These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

Given an alphabet Σ (like the English alphabet), an **encoding** is a function that maps each letter in Σ to a binary string, called a **codeword**.

For example, suppose $\Sigma = \{a, b, c, d, e, f\}$ and consider the following encoding

$$a = 000, b = 001, c = 010, d = 011, e = 100, f = 101.$$

The word "bed" can be encoded as 001100011.

Think: it is interesting to notice that there is no ambiguity when we decode the word "bed." Is this always true for any encoding? Or is it due to the special design of the above encoding?

We can reduce the length of encoding if letters' usage frequencies are known.

Suppose that, in a document, 10% of the letters are a, namely, the letter has **frequency** 10%. Similarly, suppose that letters b, c, d, e, and f have frequencies $20\%, 13\%, 9\%, 40\%$, and 8%, respectively.

If we use the encoding $a = 100$, $b = 111$, $c = 101$, $d = 1101$, $e = 0$, $f = 1100$, the **average number** of bits per letter is:

$$3 \cdot 0.1 + 3 \cdot 0.2 + 3 \cdot 0.13 + 4 \cdot 0.09 + 1 \cdot 0.4 + 4 \cdot 0.08 = 2.37.$$

This is better than using 3 bits per letter.

However, is this the best we can do? What if there is another encoding that cost less on average? E.g.,

$$e = 0, b = 1, c = 00, a = 01, d = 10, f = 11.$$

What is wrong with the encoding $e = 0, b = 1, c = 00, a = 01, d = 10, f = 11$?

- **Ambiguity in decoding!** For example, does the string 10 mean "be" or "d"?

To allow decoding, we enforce the following constraint:

> No letter's codeword should be a prefix of another letter's codeword.

An encoding satisfying the constraint is said to be a **prefix code**.

Example: The encoding $a = 100$, $b = 111$, $c = 101$, $d = 1101$, $e = 0$, $f = 1100$ is a prefix code. Just for fun, try decoding the following binary string.

$$10011010100110011100$$

The Prefix Coding Problem

For each letter $\sigma \in \Sigma$, let $freq(\sigma)$ denote the frequency of σ. Also, denote by $len(\sigma)$ the number of bits in the codeword of σ.

Given an encoding, its **average length** is

$$\sum_{\sigma \in \Sigma} freq(\sigma) \cdot len(\sigma)$$

The **prefix coding problem**:

- Given an alphabet Σ and the frequency for each letter in it, find a prefix code for Σ with the shortest average length.

Prefix Codes and Binary Trees

There is an interesting connection between prefix codes and binary trees. We will utilize this connection later.

A **code tree** on Σ as a binary tree T satisfying:

- Every leaf node of T corresponds to a unique letter in Σ; every letter in Σ corresponds to a unique leaf node in T.
- For every internal node of T, its left edge (if exists) is labeled 0, and its right edge (if exists) is labeled 1.

T generates a prefix code as follows:

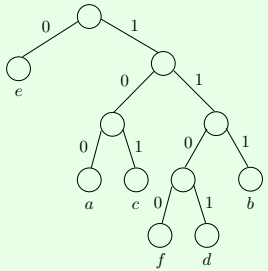
- For each letter $\sigma \in \Sigma$, generate its codeword by concatenating the bit labels of the edges on the path from the root of T to σ.

Think: Why must the encoding be a prefix code?

The Binary-Tree Representation of Prefix Codes

Prefix Codes and Binary Trees: An Example

Example: For our encoding $a = 100$, $b = 111$, $c = 101$, $d = 1101$, $e = 0$, and $f = 1100$, the code tree is:



Prefix Codes and Binary Trees

Lemma: Every prefix code is generated by a code tree.

The proof will be left as a regular exercise.

Prefix Coding Problem, Restated

Let T be the code tree generating a prefix code. Given a letter σ of Σ, its code word length $len(\sigma)$ is the **level** of its leaf node $level(\sigma)$ in T (i.e., the number of edges from the root to node σ).
Hence:

$$\text{avg length} = \sum_{\sigma \in \Sigma} freq(\sigma) \cdot len(\sigma) = \sum_{\sigma \in \Sigma} freq(\sigma) \cdot level(\sigma) = \text{avg height of } T$$

Goal (restated): Find a code tree on Σ with the smallest average height.

Huffman Codes: Solution to the prefix coding problem

Huffman Coding: An Anecdote I

In 1951, David A. Huffman was a graduate student in electrical engineering at MIT. In a course on information theory taught by Robert M. Fano, students were given a choice for their term project:

- Either write a term paper on the topic of data encoding, or develop a better method of encoding information.

Huffman initially struggled with the problem of finding an optimal way to encode symbols based on their frequency. He spent weeks looking for a solution but was about to give up and start writing the paper instead.

Then, in a moment of insight, he realized that he could construct an optimal prefix code greedily, by building a binary tree from the bottom up, always combining the two least frequent symbols. This method became what we now know as **Huffman coding**.

Huffman Coding: An Anecdote II

Huffman submitted his algorithm instead of the paper—and his professor, Fano, immediately recognized its significance. Interestingly, Fano himself had been working on a similar problem and had developed **Shannon–Fano** coding, which is less efficient than Huffman's method.

David Huffman's homework assignment not only earned him an "A," it also became one of the most important and widely used algorithms in data compression, used in formats like JPEG, MP3, and ZIP files.

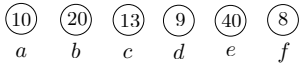
You can access Huffman's original paper titled A Method for the Construction of Minimum-Redundancy Codes.

Huffman Coding: Demonstrated by an Example I

Example

Consider our earlier example where a, b, c, d, e, and f have frequencies $0.1, 0.2, 0.13, 0.09, 0.4$, and 0.08, respectively.

Initially, S has 6 nodes:



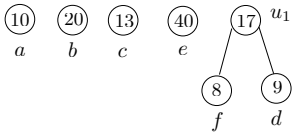
The number in each circle represents frequency (e.g., 10 means 10%).

Huffman Coding: Demonstrated by an Example II

In short, Huffman's algorithm is to keep merging the two trees with the lowest frequency into a new tree (whose frequency is the sum of the two sub-trees), until there is only one tree left.

Example

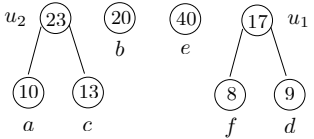
Merge the two nodes with the smallest frequencies 8 and 9. Now S has 5 nodes $\{a, b, c, e, u_1\}$:



Huffman Coding: Demonstrated by an Example III

Example

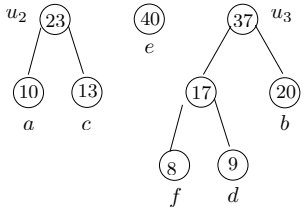
Merge the two nodes with the smallest frequencies 10 and 13. Now S has 4 nodes $\{b, e, u_1, u_2\}$:



Huffman Coding: Demonstrated by an Example IV

Example

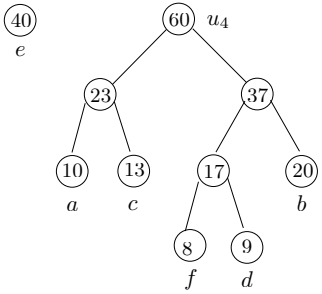
Merge the two nodes with the smallest frequencies 17 and 20. Now S has 3 nodes $\{e, u_2, u_3\}$:



Huffman Coding: Demonstrated by an Example V

Example

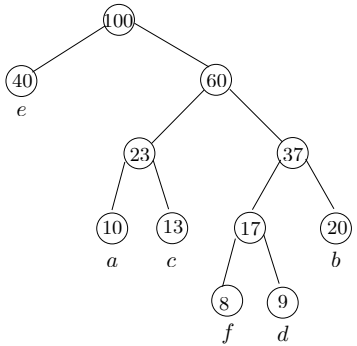
Merge the two nodes with the smallest frequencies 23 and 37. Now S has 2 nodes $\{e, u_4\}$:



Huffman Coding: Demonstrated by an Example VI

Example

Merge the two remaining nodes. Now S has a single node left.



This is the final code tree.

Huffman Coding: the Formal Description

Huffman's Algorithm

Let $n = |\Sigma|$. In the beginning, create a set S of n stand-alone leaves, each corresponding to a distinct letter in Σ.
We also define the notion of "frequency" for leaves: If a leaf is corresponding to a letter σ, define the **frequency** of this leaf to be $freq(\sigma)$.

Then, repeat until $|S| = 1$:

1. Remove from S two nodes u_1 and u_2 with the smallest frequencies.
2. Create a node v with u_1 and u_2 as the children. Set the **frequency** of v to be the frequency sum of u_1 and u_2.
3. Add v to S.

When $|S| = 1$, we have obtained a code tree. The prefix code derived from this tree is a Huffman code.

Huffman Coding: Running Time

It is easy to implement the algorithm in $O(n \log n)$ time (exercise).

Next, we prove that the algorithm gives an optimal code tree, i.e., one that minimizes the average height.

Huffman Coding: Proof of Correctness

Proof of Correctness

We now proceed to prove the correctness of Huffman's algorithm.

The proof relies on two important properties about the binary of prefix codes. In the following, we first discuss about the two facts, and then show the proof.

Two Properties

We will rely on the following two properties:

Property 1: In an optimal code tree, every internal node of T must have two children.

- The proof is left as an exercise.

Property 2: Let σ_1 and σ_2 be two letters in Σ with the lowest frequencies. There exists an optimal code tree where σ_1 and σ_2 have the same parent.

- The proof is left as an exercise. (Hint: If σ_1 and σ_2 do not have the same parent in a given optimal code tree, use an exchange argument to find another optimal code tree where they do share the same parent.)

Correctness of Huffman's Algorithm

Theorem: Huffman's algorithm produces an optimal prefix code.

Proof: We will prove by induction on the size n of the alphabet Σ.

Base Case: $n = 2$. In this case, the algorithm encodes one letter with 0, and the other with 1, which is clearly optimal.

Induction Step: Assuming the theorem's correctness for $n = k - 1$ where $k \geq 3$, next we show that it also holds for $n = k$.

The Induction Step I

Intuition: we want to utilize the induction hypothesis about the case $n = k - 1$, to prove for the case $n = k$.

We now define two sets of notations, one corresponding to the case $n = k - 1$, the other corresponding to the case $n = k$.

These notations will reveal important structure of the problem, and help us fulfill the above intuition.

The Induction Step II

Notations "corresponding" to $n = k - 1$:

- Σ : a alphabet of size $k - 1$
  - Let σ_1 and σ_2 be two letters in Σ with the lowest frequencies.
- T : an optimal code tree on Σ, where leaves σ_1 and σ_2 have the same parent p. (Guaranteed by **Property 2**)
- T_{huff} : output of Huffman's algorithm on Σ.

Notations "corresponding" to $n = k$:

- Σ' : an alphabet Constructed from Σ by removing σ_1 and σ_2, and adding a letter σ^* with frequency $freq(\sigma_1) + freq(\sigma_2)$.
- T' : the tree obtained by removing leaves σ_1 and σ_2 from T (thus making p a leaf).
- T'_{huff} : output of Huffman's algorithm on Σ'.

The Induction Step III

Recall our goal: to prove that

$$\text{avg height of } T_{huff} \leq \text{avg height of } T$$

Facts about T_{huff} and T'_{huff}:

1. **Fact 1:** σ_1 and σ_2 have the same parent in T_{huff}; and T'_{huff} is also the tree obtained by removing leaves σ_1 and σ_2 from T_{huff}. (By how Huffman's algorithm works.)
2. **Fact 2:** T'_{huff} is the optimal prefix code tree on Σ'. (By our induction hypothesis.)

The Induction Step IV

It follows from **Fact 1** that

$$\text{avg height of } T_{huff} = \text{avg height of } T'_{huff} + freq(\sigma_1) + freq(\sigma_2).$$

It follows from the definition of T' that

$$\text{avg height of } T = \text{avg height of } T' + freq(\sigma_1) + freq(\sigma_2).$$

It follows from **Fact 2** that

$$\text{avg height of } T'_{huff} \leq \text{avg height of } T'.$$

The above three inequalities imply:

$$\text{avg height of } T_{huff} \leq \text{avg height of } T.$$

Q.E.D.