CS1570 Lecture Notes

Claire Mathieu

Fall 2026

1 September 22: Huffman Coding and Text Compression

1.1 Introduction to Text Compression

The goal of text compression is to adjust binary encoding to minimize the number of bits needed to store or transmit a text file. Formally, a code maps an alphabet of characters to sequences of binary digits:

$$\text{Alphabet} \longrightarrow \{0,1\}^*$$

$$\text{letter } x \longmapsto \text{codeword for } x$$

1.1.1 Fixed-Length vs. Variable-Length Coding

Consider an alphabet with six letters and their respective frequencies (or percentages of occurrence in a text):

Letters	a	b	c	d	e	f
Frequencies	45	13	12	16	9	5

- **Fixed-Length Encoding:** Using standard binary blocks, representing 6 distinct characters requires $\lceil \log_2 6 \rceil = 3$ bits per letter. For a text containing n letters, the total cost is $3n$ bits.
- **Variable-Length Encoding:** Characters that appear more frequently can be assigned shorter codewords, while rarer characters receive longer codewords.

Using the following variable-length code:

Letter	a	b	c	d	e	f
Codeword	0	101	100	111	1101	1100
Bits	1	3	3	3	4	4

The expected number of bits per letter becomes:

$$\begin{aligned}\text{Average bits/letter} &= 0.45 \times 1 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 4 + 0.05 \times 4 \\ &= 0.45 + 0.39 + 0.36 + 0.48 + 0.36 + 0.20 \\ &= 2.24 \text{ bits}\end{aligned}$$

For n letters, the file requires $2.24n$ bits, a significant improvement over $3n$.

1.1.2 Unique Decodability and Prefix-Free Codes

Any valid code must be **uniquely decodable**. For example, consider two candidate codes C and C':

Code	a	b	c	d
C	0	110	10	111
C'	1	110	10	111

If we receive the string `110111` under code C, it could be decoded as `b d`. Under code C', the exact same binary stream `110111` could be parsed as either `a c d` or `b d`, creating ambiguity.

To achieve unambiguous decoding, we target **prefix-free codes** (or prefix codes), where no codeword is a prefix of another codeword.

- *Question:* Does there exist a uniquely decodable code that compresses strictly better than the best prefix-free code?
- *Answer:* In class, I was not sure how to answer it. Actually the answer is No. It is the Kraft-McMillan inequality (see Wikipedia). By Kraft's Inequality, any uniquely decodable code can be transformed into a prefix-free code with the exact same codeword lengths.

Remark: Advanced methods like Lempel-Ziv (LZ77/LZ78) relax the strict letter-by-letter mapping condition by encoding variable-length sequences of characters to achieve even better compression.

1.2 Formal Problem Definition

- **Input:** A set of letters and their corresponding frequencies.
- **Output:** A prefix-free binary code minimizing the total length of the encoded text.
- **Objective Function:** Minimize $\sum_{x \in \text{letters}} \text{freq}(x) \cdot \text{length}(\text{encoding}(x))$.

1.3 Unsuccessful Algorithmic Approaches

Before examining the optimal solution, it is instructive to explore intuitive strategies that turn out to be incorrect or suboptimal.

1.3.1 Failed Attempt 1: Top-Down Naive Greedy Strategy

An intuitive greedy idea is to process characters in order of decreasing frequency and assign them the shortest available binary sequence that does not conflict with existing codewords.

- **Strategy:** Sort letters by frequency. Assign the shortest sequence that is not a prefix of any previously selected codeword.
- **Why it fails:** Consider letters $\{e, s, a, i\}$ sorted by decreasing frequency.
  1. Assign $e \to 0$.
  2. The next shortest available sequence is $s \to 1$.
  3. Now, every binary sequence begins with either 0 or 1. Consequently, no further codewords can be created without violating the prefix-free rule. The algorithm gets stuck and is **incorrect**.

1.3.2 Failed Attempt 2: Divide-and-Conquer (Shannon-Fano Coding)

Another approach is to design a top-down partition mechanism.

- **Strategy:**
  1. Sort the alphabet by frequency.
  2. Partition the list into two parts such that the sum of the frequencies on each side is as close to 50% as possible.
  3. Recursively solve each subset.
  4. Combine the solutions by prepending a `0` to the codewords in the first set and a `1` to the codewords in the second set.
- **Why it fails:** While this heuristic produces valid prefix-free codes, it is **not optimal in general**. Balancing the frequencies at the top level does not guarantee a globally minimal total path length across all individual leaves. (to prove this: **Exercise**

1.4 Tree Representation of Codes

A binary code can be naturally represented as a tree where:

- Every left branch corresponds to a `0`.
- Every right branch corresponds to a `1`.
- Letters are placed exclusively at the **leaves** to ensure the code is prefix-free.
- The path from the root to a leaf determines its codeword, and the **depth** of the leaf equals the length of the codeword.

If a code tree contains internal nodes with only one child, or if there are unused leaves, the representation is not optimal (since it can be improved by getting rid of a superfluous 0 or 1 where there is no sibling). An optimal prefix code always forms a **full binary tree**, meaning every internal node has exactly two children.

1.4.1 Rephrasing the Optimization Problem

Using the structural insights of trees, our problem can be restated:

- **Input:** A set of letters and their frequencies.
- **Output:** A full binary tree where there is a 1-to-1 correspondence between leaves and letters.
- **Goal:** Minimize $\sum_{x \in \text{letters}} \text{freq}(x) \cdot \text{depth}(x)$.

A brute-force strategy would search through every full binary tree with n leaves and evaluate every possible assignment of letters to those leaves. This is a very slow algorithm due to the exponential growth of tree topologies and the super-exponential number of bujections mapping letters to leaves.

1.4.2 Key Structural Properties

- **Fact:** If we order the letters by decreasing frequency ($\text{freq}(x_1) \geq \text{freq}(x_2) \geq \cdots \geq \text{freq}(x_n)$), then their corresponding depths in an optimal tree must be non-decreasing ($\text{depth}(x_1) \leq \text{depth}(x_2) \leq \cdots \leq \text{depth}(x_n)$). In other words, high-frequency items must sit closer to the root.
- **Top-Down Difficulty:** Attempting to guess the optimal split from the root downwards requires finding the perfect subset partition. Because a subproblem can consist of any arbitrary subset of the n letters, this yields too many possibilities (exponential) to evaluate efficiently.

1.5 The Bottom-Up Solution: Huffman's Algorithm

Instead of working top-down from the root, Huffman's algorithm works **bottom-up** from the deepest leaves.

1.5.1 Algorithm Sketch

1. Find the two least frequent letters in the current set (say, e and f).
2. Pair them up under a common parent node. This establishes that they will be sibling leaves.
   Note: when I asked my favorite AI agent to create lecture notes from the pictures of the board, here it wrote: This establishes that they will be sibling leaves at the maximum depth of the tree", which is wrong: yes, in Huffman coding e and f will end up being siblings at the maximum depth of the tree, but no, doing this second step, in itself, only guaranties that they will be siblings, not that they will be at maximum depth; that follows from other properties. So this is one instance where the AI writeup gets it wrong in a subtle way that is hard to detect with a superficial examination!
   Note number 2: the latex help that proposes corrections to improve my writing suggested removing "when I asked my favorite AI agent to create lecture notes from the pictures of the board, here it wrote:" – AI does not like criticism of AI, apparently!
3. Combine their individual frequencies (freq(e) + freq(f)) and assign this sum as the weight of a new letter x.
4. Remove e and f from the set of letters, replace them with the new letter, and recursively find prefix-free code for the new set of letters.
5. Deduce a prefix-free code by taking the solution and padding the encodeing of x with a zero and with a 1 to get an encoding of e and of f respectively.

1.5.2 Example of Bottom-Up Execution

Given our original frequencies for $\{a : 45, b : 13, c : 12, d : 16, e : 9, f : 5\}$:

- The two smallest weights are $f(5)$ and $e(9)$. We pair them to create a combined node ef with a weight of $5 + 9 = 14$.
- The active set becomes $\{a : 45, b : 13, c : 12, d : 16, ef : 14\}$.
- The next two smallest weights are $c(12)$ and $b(13)$. We pair them to form cb with a weight of $12 + 13 = 25$.
- The process continues bottom-up, guaranteeing that the rarest letters are automatically positioned furthest down in the tree topology.