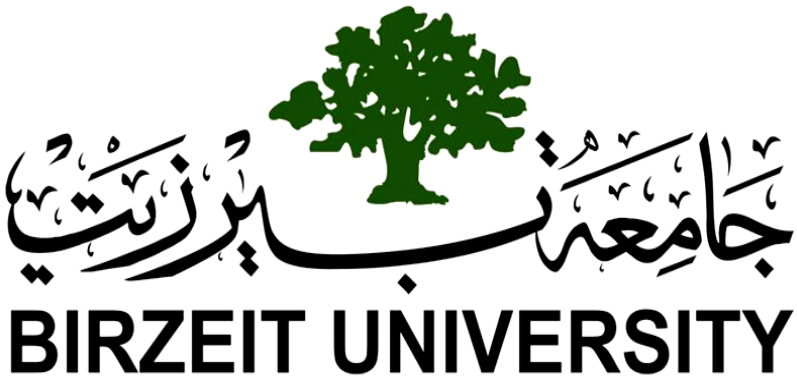

Faculty of Engineering & Technology
Electrical & Computer Engineering Department
INFORMATION AND CODING THEORY - ENEE5304

Course Project: Huffman Code Implementation

Prepared by:

Hatem Hussein – 1200894

Nafe Abubaker – 1200047

Instructor: Dr. Wael Hashlamoun

Section: 1

Date: January 10, 2025

Table of Contents

List of Figures

List of Tables

1. Introduction

In this project, we aim to address the problem of efficiently compressing text data without losing any information. Data compression is essential in minimizing storage requirements and speeding up data transmission, especially for large files. The challenge lies in creating a coding system that reduces the size of data while maintaining its integrity and ensuring that it can be accurately decoded.

Using the short story "To Build a Fire" by Jack London, this project focuses on implementing Huffman coding to analyze character frequencies, calculate their probabilities, and generate optimal prefix codes. The goal is to compare the efficiency of Huffman coding against the standard ASCII encoding and evaluate the compression achieved. This involves calculating the entropy of the dataset, determining the average bits per character, and assessing the overall reduction in data size. Through this approach, we aim to demonstrate the effectiveness of Huffman coding as a tool for lossless data compression.

2. Background and Methodology

Huffman coding is an efficient method of compressing data without losing information. In computer science, information is encoded as bits—1's and 0's. Strings of bits represent the information that tells a computer which instructions to execute. Data like video games, photographs, and movies are all encoded as strings of bits in a computer. Given the vast amount of data computers handle—often billions of bits for a single file—efficient and unambiguous encoding is a key area of focus in computer science [1].

Huffman coding works by analyzing the frequencies of symbols in a message. Symbols that appear more often are assigned shorter bit strings, while less frequent symbols are assigned longer bit strings. This makes the overall encoded message smaller. However, because the frequencies of symbols vary between messages, there isn’t a universal Huffman coding; the code is generated based on the specific message being encoded [1].

Huffman coding uses a specific method for choosing how each symbol is represented, resulting in a prefix code. A prefix code means that no codeword is a prefix of another, ensuring that the encoded message can be decoded unambiguously. This property makes Huffman coding highly reliable and widely used for creating prefix-free codes [3], [4].

To construct a Huffman tree, we begin with a set of symbols and their weights, where weights are typically proportional to their probabilities of occurrence [4]. The goal is to find a prefix-free binary code with the minimum expected codeword length. Formally:

Given: Alphabet, where is the size of the alphabet. In addition, Tuple of weights, where is the weight (or probability) of symbol.

Find: A binary code such that each codeword corresponds to.

Goal: Minimize the weighted path length.

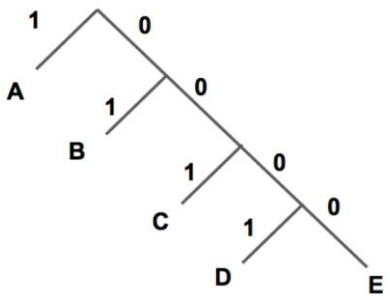

Figure 2.1: Constructing a Huffman Tree [2]

The entropy of an alphabet provides the theoretical lower bound for the average number of bits per symbol. In practice, the average codeword length achieved by Huffman coding often approaches this entropy value, making it nearly optimal. For example, if symbols have weights, their Huffman codes would minimize, which could be compared to evaluate efficiency. This demonstrates how Huffman coding achieves effective compression while adhering to the theoretical limits established by Shannon's source coding theorem [4].

To implement Huffman coding, the short story "To Build a Fire" by Jack London was processed using a Java program. The methodology involved the following steps:

1. **Character Occurrence Counting**:
    - Each character, including spaces and select punctuation marks (comma, dash, apostrophe, and period), was counted to determine its frequency in the text. Capital letters were converted to lowercase to simplify analysis.
    - An array stored the occurrences of 26 letters and 5 symbols, resulting in a total of 31 entries.
2. **Frequency Calculation**:
    - The frequency of each character was calculated by dividing its count by the total number of characters in the text.
    - These frequencies were then used to determine the probability of each character appearing in the text.
3. **Entropy Calculation**:
    - Using the formula, the entropy of the text was computed to measure the theoretical minimum number of bits required per character.
4. **Huffman Tree Construction**:
    - A priority queue was used to build the Huffman tree, where nodes with the lowest frequencies were combined iteratively until a single tree was formed.
    - Each leaf node in the tree represented a character and its Huffman code.
5. **Huffman Code Generation**:
    - The Huffman codes for each character were derived from the tree structure. The length of each code depended on the character's frequency, with shorter codes assigned to more frequent characters.
6. **Comparison with ASCII Encoding**:
    - The total number of bits required for encoding the text using Huffman coding () was calculated and compared to the bits required using ASCII (, where each character uses 8 bits).
7. **Compression Percentage and Average Bits per Character**:
    - The compression percentage was computed using the formula:
    - The average bits per character were calculated as.

3. Results and Analysis

The program analyzed the text and produced the following results:

```
=== Summary ===
Total Characters         : 37631
Entropy                  : 4.160866
NASCII (ASCII bits)      : 301048 bits
Nhuffman (Huffman bits)  : 158348 bits
Avg Bits/Char (Huffman)  : 4.207914 bits
Compression Percentage   : 47.40%
```

Figure 3.1: Summary Results Simulation

The entropy value of 4.16 bits per character represents the theoretical minimum average bits required to encode the text. The Huffman coding implementation achieved an average of 4.21 bits per character, which is close to the entropy, demonstrating the near-optimal efficiency of the Huffman algorithm.

The ASCII encoding method required 301,048 bits to encode the text, while the Huffman encoding reduced this to 158,348 bits. This reduction results in a compression percentage of 47.40%, highlighting the effectiveness of Huffman coding in minimizing data size without losing information.

The table below summarizes the Huffman coding results for select symbols, including their occurrences, probabilities, codewords, and codeword lengths:

Table 3.1: Results Table

Symbol	Occurrences	Probability	Codeword	Codeword Length
a	2,262	0.060110	1001	4
b	482	0.012809	100000	6
c	778	0.020674	110110	6
d	1,513	0.040206	11010	5
e	3,886	0.103266	010	3
f	793	0.021073	00000	5
g	620	0.016476	100001	6
h	2,278	0.060535	1010	4
i	1,981	0.052643	0110	4
j	19	0.000505	00011011101	11
k	303	0.008052	1011000	7
l	1,125	0.029896	10001	5
m	678	0.018017	101101	6
n	2,075	0.055141	0111	4
o	1,968	0.052297	0011	4
p	421	0.011188	000101	6

q	17	0.000452	00011011100	11
r	1,480	0.039329	10111	5
s	1,795	0.047700	0010	4
t	2,936	0.078021	1100	4
u	799	0.021232	00001	5
v	179	0.004757	0001100	7
w	788	0.020940	110111	6
x	34	0.000904	00011011111	11
y	355	0.009434	1011001	7
z	61	0.001621	000110110	9
space	7,044	0.187186	111	3
. (period)	414	0.011002	000100	6
, (comma)	436	0.011586	000111	6
- (dash)	91	0.002418	00011010	8
' (apostrophe)	20	0.000531	00011011110	11

4. Conclusion

The results demonstrate the efficiency of Huffman coding in reducing data size compared to fixed-length encoding schemes like ASCII. The close alignment between the entropy and average bits per character illustrates the algorithm's effectiveness in approaching the theoretical limits of compression. However, Huffman coding is not always optimal for all scenarios. It works best when symbol probabilities are well-defined and static, but it may struggle with dynamic data distributions or when handling extremely small datasets. Additionally, modern techniques like arithmetic coding or range encoding can outperform Huffman coding in achieving higher compression ratios in some cases. These insights highlight both the strengths and the limitations of Huffman coding, making it an important but context-dependent tool in data compression applications.

5. References

[1] *Huffman Code*. (2025). Retrieved from brilliant: https://brilliant.org/wiki/huffman-encoding/

[2] Huffman Code Tree. (2025). Retrieved from brilliant: https://brilliant.org/problems/find-the-encoding/

[3] Huffman coding. (2023). Retrieved from programiz: https://www.programiz.com/dsa/huffman-coding

[4] Huffman coding. (2025). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Huffman_coding

6. Appendix

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.PriorityQueue;

// Class for Huffman Tree Nodes
class HuffmanNode implements Comparable<HuffmanNode> {
    char character;
    int frequency;
    HuffmanNode left, right;

    HuffmanNode(char character, int frequency) {
        this.character = character;
        this.frequency = frequency;
        this.left = this.right = null;
    }

    @Override
    public int compareTo(HuffmanNode other) {
        return this.frequency - other.frequency;
    }
}

public class main {
    static int Nhuffman = 0; // Total bits required for Huffman encoding

    public static void main(String[] args) throws IOException {
        try {
            int character; // for reading from the text file
            int[] charactersOccurances = new int[31]; // Array to hold the occurrences of the characters (26 letters +
                                                      // 5// symbols)
            double[] charactersFrequencies = new double[31]; // Array to hold the frequencies of the characters
```

```
int charactersCount = 0; // Counter for all the characters
double Entropy = 0;

// Create a FileReader to read "Story.txt"
FileReader reader = new FileReader("Story.txt");

while ((character = reader.read()) != -1) {
    // Convert the character code to a char
    char ch = (char) character;

    // Check if the character is a letter or a target symbol
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {
        charactersOccurances[Character.toLowerCase(ch) - 'a']++;
        charactersCount++;
    } else if (ch == ' ') {
        charactersOccurances[26]++;
        charactersCount++;
    } else if (ch == ',') {
        charactersOccurances[27]++;
        charactersCount++;
    } else if (ch == '-') {
        charactersOccurances[28]++;
        charactersCount++;
    } else if (ch == '\'') {
        charactersOccurances[29]++;
        charactersCount++;
    } else if (ch == '.') {
        charactersOccurances[30]++;
        charactersCount++;
    }
}

// Print the occurrences of each letter and symbol in the text file.
System.out.println("\n=== Character Occurrences ===");
System.out.printf("%-10s%-10s%n", "Character", "Occurrences");
System.out.println("----------------------------");
```

```
for (int i = 0; i < 26; i++) {
    System.out.printf("%-10c%-10d%n", (char) (i + 'a'), charactersOccurances[i]);
}
System.out.printf("%-10s%-10d%n", "Space", charactersOccurances[26]);
System.out.printf("%-10s%-10d%n", ",", charactersOccurances[27]);
System.out.printf("%-10s%-10d%n", "-", charactersOccurances[28]);
System.out.printf("%-10s%-10d%n", "'", charactersOccurances[29]);
System.out.printf("%-10s%-10d%n", ".", charactersOccurances[30]);

// Calculate the frequencies of each letter and symbol by dividing its
// occurrence over the total characters count.
System.out.println("\n=== Character Frequencies ===");
System.out.printf("%-10s%-15s%n", "Character", "Frequency");
System.out.println("--------------------------------");
for (int i = 0; i < 31; i++) {
    charactersFrequencies[i] = (double) charactersOccurances[i] / charactersCount;
    if (i < 26) {
        System.out.printf("%-10c%-15.6f%n", (char) (i + 'a'), charactersFrequencies[i]);
    } else if (i == 26) {
        System.out.printf("%-10s%-15.6f%n", "Space", charactersFrequencies[i]);
    } else if (i == 27) {
        System.out.printf("%-10s%-15.6f%n", ",", charactersFrequencies[i]);
    } else if (i == 28) {
        System.out.printf("%-10s%-15.6f%n", "-", charactersFrequencies[i]);
    } else if (i == 29) {
        System.out.printf("%-10s%-15.6f%n", "'", charactersFrequencies[i]);
    } else if (i == 30) {
        System.out.printf("%-10s%-15.6f%n", ".", charactersFrequencies[i]);
    }
}

// Calculating the Entropy of the source
for (int i = 0; i < 31; i++) {
    if (charactersFrequencies[i] > 0) { // Avoid log(0)
        Entropy += -1 * charactersFrequencies[i] * Math.log10(charactersFrequencies[i]) / Math.log10(2);
    }
```

```
}
System.out.println("\nEntropy = " + Entropy);

// Generating Huffman Codes
generateHuffmanCodes(charactersOccurances);

// Calculating NASCII (ASCII encoding size)
int NASCII = charactersCount * 8; // Each character uses 8 bits in ASCII
System.out.println("\nNASCII (ASCII bits required): " + NASCII);

// Printing Nhuffman (already calculated during Huffman coding)
System.out.println("Nhuffman (Huffman bits required): " + Nhuffman);

// Calculating Compression Percentage
double compressionPercentage = ((double) (NASCII - Nhuffman) / NASCII) * 100;
System.out.println("Compression Percentage: " + compressionPercentage + "%");

// Calculating Average Bits per Character using Huffman Code
double averageBitsPerCharacter = (double) Nhuffman / charactersCount;
System.out.println("Average Bits per Character (Huffman): " + averageBitsPerCharacter);

// Comparing Average Bits per Character to Entropy
System.out.println(
      "Comparison: Average Bits/Character = " + averageBitsPerCharacter + ", Entropy = " + Entropy);

// Printing Summary
System.out.println("\n=== Summary ===");
System.out.printf("%-25s: %d%n", "Total Characters", charactersCount);
System.out.printf("%-25s: %.6f%n", "Entropy", Entropy);
System.out.printf("%-25s: %d bits%n", "NASCII (ASCII bits)", NASCII);
System.out.printf("%-25s: %d bits%n", "Nhuffman (Huffman bits)", Nhuffman);
System.out.printf("%-25s: %.6f bits%n", "Avg Bits/Char (Huffman)", averageBitsPerCharacter);
System.out.printf("%-25s: %.2f%%%n", "Compression Percentage", compressionPercentage);

// Closing the reader
reader.close();
```

```
    } catch (FileNotFoundException e) {
        // Handle the case where the file is not found
        System.out.println("The file 'Story.txt' was not found!");
    }
}

// Function to generate Huffman codes
public static void generateHuffmanCodes(int[] frequencies) {
    PriorityQueue<HuffmanNode> queue = new PriorityQueue<>();

    // Create a node for each character with its frequency
    for (int i = 0; i < frequencies.length; i++) {
        if (frequencies[i] > 0) {
            if (i < 26) {
                queue.add(new HuffmanNode((char) (i + 'a'), frequencies[i]));
            } else if (i == 26) {
                queue.add(new HuffmanNode(' ', frequencies[i]));
            } else if (i == 27) {
                queue.add(new HuffmanNode(',', frequencies[i]));
            } else if (i == 28) {
                queue.add(new HuffmanNode('-', frequencies[i]));
            } else if (i == 29) {
                queue.add(new HuffmanNode('\'', frequencies[i]));
            } else if (i == 30) {
                queue.add(new HuffmanNode('.', frequencies[i]));
            }
        }
    }

    // Build the Huffman Tree
    while (queue.size() > 1) {
        HuffmanNode left = queue.poll();
        HuffmanNode right = queue.poll();

        HuffmanNode newNode = new HuffmanNode('\0', left.frequency + right.frequency);
        newNode.left = left;
```

```
            newNode.right = right;

            queue.add(newNode);
        }

        // Generate the Huffman codes
        HuffmanNode root = queue.poll();
        System.out.println("\n=== Huffman Codes ===");
        System.out.printf("%-10s%-15s%n", "Character", "Huffman Code");
        System.out.println("---------------------------------");
        printHuffmanCodes(root, "");
    }

    // Function to print Huffman codes and calculate Nhuffman
    public static void printHuffmanCodes(HuffmanNode root, String code) {
        if (root == null) {
            return;
        }
        if (root.character != '\0') { // Leaf node
            System.out.printf("%-10c%-15s%n", root.character, code);
            Nhuffman += root.frequency * code.length();
        }
        printHuffmanCodes(root.left, code + "0");
        printHuffmanCodes(root.right, code + "1");
    }
}
```