Faculty of Engineering & Technology

Electrical & Computer Engineering Department

INFORMATION AND CODING THEORY - ENEE5304

Course Project on Huffman Code

Project Report

Prepared by:

Name: Mohammad Abu Shams **ID:** 1200549

Name: Joud Hijaz **ID:** 1200342

Instructor: Dr. Wael Hashlamoun

Section: 2

Date: 7-1-2025

BIRZEIT

Table of Contents

Introduction

In today's world, digital communication and data storage are growing very fast. Making data smaller is very important to save space and send it faster. Huffman coding is a famous way to make data smaller without losing any information. It was created by David A. Huffman in 1952 [1]. Huffman coding looks at how many times each character appears in the data. Characters that appear more times get shorter binary codes, and characters that appear less times get longer binary codes.

This project uses Huffman coding to make the English story `*To Build A Fire*` `by Jack London`. The steps include counting how many times each character appears, finding their probabilities, and creating Huffman codes. The project also checks the entropy of the data and compares Huffman coding with the fixed-size ASCII method. By calculating the bits needed for both methods and finding the compression percentage, this project shows how Huffman coding helps to reduce data size.

Huffman coding has been studied a lot and is compared with other methods like arithmetic coding. It is simple and works well [2]. Research also shows it helps use bandwidth better and improve security in communication systems. This makes Huffman coding important not only for saving space but also for secure and fast communication.

Theoretical Background

1. Huffman Coding

Huffman coding is a way to make data smaller without losing any information. It was made by David A. Huffman [1]. It builds a special binary tree called the Huffman tree. In this tree, each leaf is a character, and how deep the leaf is shows the length of its binary code. The codes are prefix-free, meaning no code is the start of another code, so decoding is always clear.

The main steps of Huffman coding are:

1. **Frequency Analysis**: Count how many times each character appear in the data.
2. **Priority Queue**: Put the characters into a queue, sorted by how many times they appear (less frequent first).
3. **Tree Construction**: Combine the two characters with the smallest frequency into one node. Keep doing this until there is one root node left.
4. **Code Generation**: Move through the tree. Use 0 for left branches and 1 for right branches to make binary codes.

Huffman coding is fast and works well when characters in the data do not appear equally. It makes data much smaller in those cases.

2. Entropy

Entropy was introduced by Claude Shannon in his important work about information theory [1]. It shows the smallest number of bits we need to store or send data. Entropy measures the average information of each symbol in the dataset. The formula is:

$$(S) = -\sum (s) \log_2 (s)$$

Where:

- **S** is the set of all symbols.
- **p(s)** is the probability of a symbol **s**.

Entropy tells us the least number of bits needed to represent data. Huffman coding tries to get close to this limit by removing extra or useless bits in the data.

3. Comparison with ASCII Encoding

ASCII (American Standard Code for Information Interchange) is a simple way to encode data. It uses 8 bits for every character, no matter how often it appears. This makes it easy to use but not very good for natural language. In English, some letters like **e** or **t** appear many times, but others like **z** or **q** are rare. ASCII gives them the same size, which is not efficient.

Huffman coding fix this problem by giving shorter codes to letters that appear more often and longer codes to rare letters. This way, it makes the data smaller and saves space.

4. Efficiency of Huffman Coding

To check how efficient Huffman coding is, we can compare how many bits are needed to encode data with Huffman codes (**N_Huffman**) and with ASCII encoding (**N_ASCII**). The compression percentage can be calculated like this:

Compression Percentage = (1 − **N_Huffman / N_ASCII**) × 100

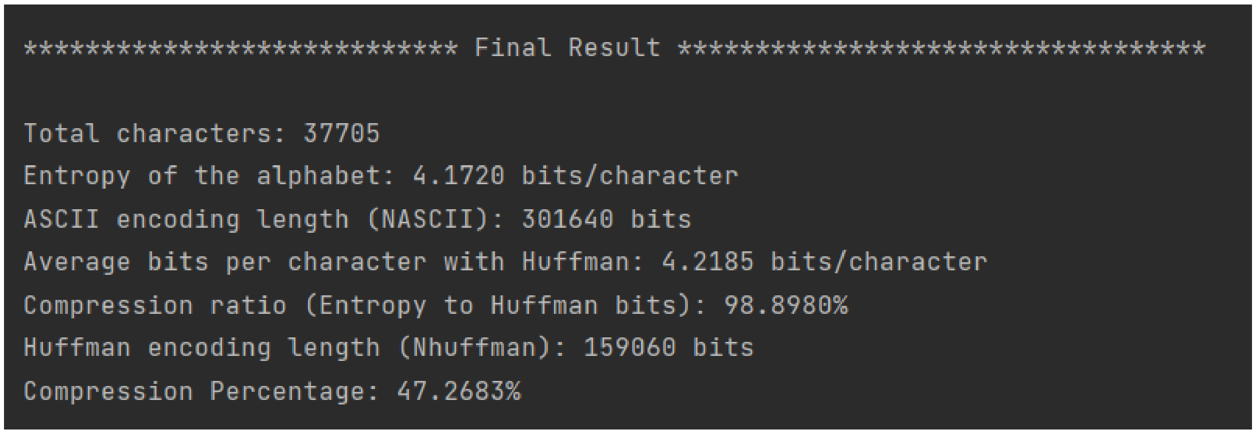
This formula shows how much smaller the data becomes with Huffman coding. Recent studies [2] have shown that Huffman coding gives a high compression rate while being simple and fast to use.

5. Applications of Huffman Coding

Huffman coding is used in many places, such as:

- **File Compression**: Programs like ZIP use Huffman coding to make files smaller.
- **Image and Multimedia Compression**: Formats like JPEG and MP3 use Huffman coding to make images and music smaller while keeping good quality.
- **Bandwidth Optimization**: In communication systems, Huffman coding helps save bandwidth by reducing unnecessary data [2].

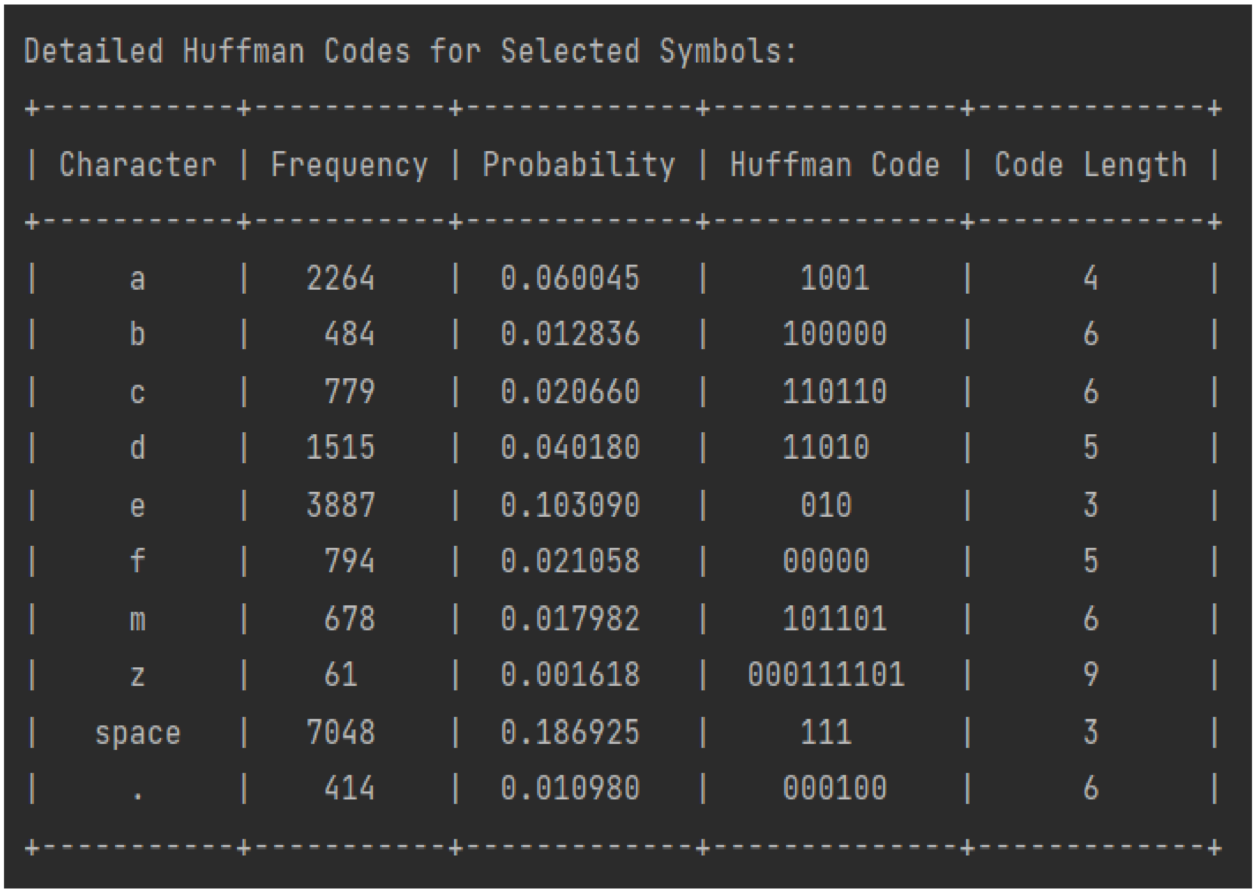
Results



```
*************************** Final Result **********************************

Total characters: 37705
Entropy of the alphabet: 4.1720 bits/character
ASCII encoding length (NASCII): 301640 bits
Average bits per character with Huffman: 4.2185 bits/character
Compression ratio (Entropy to Huffman bits): 98.8980%
Huffman encoding length (Nhuffman): 159060 bits
Compression Percentage: 47.2683%
```

The entropy of the alphabet is 4.172 bits per character. This shows the smallest number of bits needed to store the story's characters for the best compression. Huffman coding uses 4.2185 bits per character on average, which is very close to the entropy value. The small difference of 0.0465 bits per character happens because Huffman coding uses full numbers for binary codes, while entropy uses fractions. Even with this small difference, Huffman coding is very good, reaching 98.8980% of the best possible compression. This shows how Huffman coding gives almost perfect compression while keeping all the data correct.



```
Detailed Huffman Codes for Selected Symbols:
+-----------+-----------+-------------+-------------+------------+
| Character | Frequency | Probability | Huffman Code | Code Length |
+-----------+-----------+-------------+-------------+------------+
|     a     |   2264    |  0.060045   |    1001     |      4     |
|     b     |    484    |  0.012836   |   100000    |      6     |
|     c     |    779    |  0.020660   |   110110    |      6     |
|     d     |   1515    |  0.040180   |   11010     |      5     |
|     e     |   3887    |  0.103090   |    010      |      3     |
|     f     |    794    |  0.021058   |   00000     |      5     |
|     m     |    678    |  0.017982   |   101101    |      6     |
|     z     |    61     |  0.001618   |  000111101  |      9     |
|   space   |   7048    |  0.186925   |    111      |      3     |
|     .     |    414    |  0.010980   |   000100    |      6     |
+-----------+-----------+-------------+-------------+------------+
```

Conclusion

This project showed how good Huffman coding is for compressing the English story `To Build A Fire by Jack London`. By checking how often characters appear and their probabilities, Huffman codes were made, and the entropy of the dataset was calculated. The entropy was 4.172 bits per character, showing the smallest number of bits needed to store the data. Huffman coding used 4.2185 bits per character, which is very close to the entropy. This shows it can compress data almost perfectly.

The compression ratio, comparing entropy to Huffman bits, was 98.8980%, showing its great performance. Huffman coding also made the encoding size 47.2683% smaller than ASCII encoding, showing how it saves storage space while keeping all information.

In the end, this project showed that Huffman coding is a very good and useful way to compress data. It is simple and works well for many uses, like saving files, compressing videos or music, and sending data in communication systems.

References

[1] A Method for the Construction of Minimum-Redundancy Codes. (2025, 1 7). Retrieved from ieeexplore: https://ieeexplore.ieee.org/document/4051119

[2] Comparative study of Arithmetic and Huffman Compression Techniques for Enhancing Security and Effective Bandwidth Utilization in the Context of ECC for Text. (2025, 1 7). Retrieved from researchgate: https://www.researchgate.net/publication/258493922_Comparative_study_of_Arithmetic_and_Huffman_Compression_Techniques_for_Enhancing_Security_and_Effective_Bandwidth_Utilization_in_the_Context_of_ECC_for_Text

Appendix

```
# Mohammad Abu Shams 1200549.
# Joud Hijaz 1200342.

# Import necessary libraries.
import collections
import math
import heapq
from tabulate import tabulate
import docx2txt

# Define a class to represent a node in the Huffman tree.
class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq


# Function to calculate the frequency of each character in the text.
def calculate_frequencies(text):
    text = text.replace("\n", "").lower()
    freq = collections.Counter(text)
    return freq


# Function to calculate the probability of each character in the text.
def calculate_probabilities(freq, total_chars):
    probabilities = {char: count / total_chars for char, count in
freq.items()}
    return probabilities


# Function to calculate the entropy of the text.
def calculate_entropy(probabilities):
    entropy = -sum(p * math.log2(p) for p in probabilities.values() if p > 0)
    return entropy


# Function to build the Huffman tree.
def build_huffman_tree(freq):
    heap = [HuffmanNode(char, freq) for char, freq in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
```

```python
    return heap[0]  # Root of the tree


# Function to generate the Huffman codes.
def generate_huffman_codes(root, prefix="", codebook=None):
    if codebook is None:
        codebook = {}

    if root:
        if root.char is not None:  # Leaf node
            codebook[root.char] = prefix
        generate_huffman_codes(root.left, prefix + "0", codebook)
        generate_huffman_codes(root.right, prefix + "1", codebook)

    return codebook


# Function to calculate the length of the Huffman encoding.
def calculate_huffman_encoding_length(freq, codebook):
    return sum(len(codebook[char]) * freq[char] for char in freq)


# Function to calculate the compression percentage.
def calculate_compression_percentage(n_ascii, n_huffman):
    return (1 - (n_huffman / n_ascii)) * 100


# Function to read the .docx file.
def readText():
    text = docx2txt.process("To+Build+A+Fire+by+Jack+London.docx")
    return text


# MAIN.
if __name__ == "__main__":
    # Read the text file.
    story = readText()

    # Calculate frequencies.
    freq = calculate_frequencies(story)
    total_chars = sum(freq.values())

    # Calculate probabilities.
    probabilities = calculate_probabilities(freq, total_chars)

    # Calculate entropy.
    entropy = calculate_entropy(probabilities)

    # Build Huffman tree.
    huffman_tree_root = build_huffman_tree(freq)

    # Generate Huffman codes.
    huffman_codes = generate_huffman_codes(huffman_tree_root)

    # Calculate ASCII and Huffman encoding lengths.
    n_ascii = total_chars * 8  # ASCII uses 8 bits per character
    n_huffman = calculate_huffman_encoding_length(freq, huffman_codes)
    # Calculate average bits per character and compression.
    avg_bits_per_char = n_huffman / total_chars
    compression_percentage = calculate_compression_percentage(n_ascii,
n_huffman)

    # Print summary results.
    print("\n**************************** Final Result
**********************************")
    print(f"\nTotal characters: {total_chars}")
    print(f"Entropy of the alphabet: {entropy:.4f} bits/character")
    print(f"ASCII encoding length (NASCII): {n_ascii} bits")
    print(f"Average bits per character with Huffman: {avg_bits_per_char:.4f}
bits/character")
    print(f"Compression ratio (Entropy to Huffman bits): {(entropy /
avg_bits_per_char)*100:.4f}%")
    print(f"Huffman encoding length (Nhuffman): {n_huffman} bits")
    print(f"Compression Percentage: {compression_percentage:.4f}%")


 # Display selected symbols in a detailed table.
    symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', ' ', '.']
    symbol_table = [
        [symbol if symbol != " " else "space",f"{freq.get(symbol, 0)}"
,f"{probabilities.get(symbol, 0):.6f}", huffman_codes.get(symbol, ""),
len(huffman_codes.get(symbol, ""))]
        for symbol in symbols
    ]
    print("\nDetailed Huffman Codes for Selected Symbols:")
    print(tabulate(symbol_table, headers=["Character",
"Frequency","Probability", "Huffman Code", "Code Length"],
tablefmt="pretty"))
```