BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering
Department

ENEE5304

Project Report

Prepared by:
Alhasan Manasra 1211705

Instructor: **Dr. Wael Hashlamoun**
Section: 2
Date: January 10, 2025

Table Of Contents

1. Huffman Coding

1.1 Introduction

This project focuses on implementing a program to encode a story from a Word document using Huffman coding. The program provides a summary of the encoding process, including the average number of bits per codeword, the entropy, the percentage of compression, as well as the probabilities, codeword lengths, and Huffman codes for selected characters.

1.2 Theoretical Background

Huffman coding is a method that relies on the probabilities of each symbol in a given message. Initially, the frequency of each symbol is calculated, and the symbols are sorted in descending order based on their frequencies. The two symbols with the lowest frequencies are then combined into a single node, with their frequencies summed up, and each branch of the node is assigned a unique binary digit (0 and 1). This process is repeated until all symbols are merged into one tree. The codeword for each symbol is determined by tracing the binary digits along the branches from the root of the tree back to the symbol.

The core principle of Huffman coding is to assign shorter binary codes to symbols that appear more frequently, making it an efficient method for lossless data compression.

1.3 Results

- **Sum of Probabilities = 1**

The sum of all probabilities confirms that the calculated probabilities for all symbols are accurate. This is a validation step to ensure the correctness of the computations.

- **Entropy for all characters = 4.172 Bits/Character**

Entropy represents the theoretical minimum number of bits needed per character for encoding, based on the symbol probabilities. It serves as the lower bound for any lossless compression method.

- **Entropy for alphabetic chars = 3.533 Bits/Character**

This is the entropy calculated specifically for alphabetic characters (a-z), excluding spaces or other symbols. It provides a focused view of the efficiency of encoding letters.

- **For ASCII coding, num of bits = 301640**

ASCII encoding uses a fixed length of 8 bits per character. This total represents the number of bits required to encode the text using ASCII.

- **For Huffman coding, num of bits = 159060**

This is the total number of bits required to encode the text using Huffman coding. Since Huffman uses variable-length encoding, this value is significantly smaller than the ASCII bit count.

- **Average Length of the Code = 4.218 Bits/Character**

This is the average number of bits required to encode a single character using Huffman coding. It reflects the efficiency of the Huffman algorithm by assigning shorter codes to more frequent characters.

- **Informational Efficiency = 98.90%**

Informational efficiency is the ratio of the entropy to the average length of the code, expressed as a percentage. A value of 98.90% means Huffman coding achieves near optimal compression, approaching the theoretical minimum (entropy).

- **Percentage of Compression = 47.27%**

This shows how much the Huffman coding reduces the total number of bits compared to ASCII. A compression percentage of 47.27% indicates that Huffman coding effectively reduces the storage or transmission size of the text.

```
Sum of Probabilities = 1.00
Entropy for all char = 4.172049
Entropy for alphabet char = 3.533082
For ASCII coding, num of bits = 301640
For Huffman coding, num of bits = 159060
Average length of the code = 4.218539
Informational Efficiency = 98.90%
Percentage of Compression = 47.27%
```

Symbol	Probability	Length Of Codeword	Codeword
' '	0.187	3	111
'!'	0.000	14	00011111011011
'"'	0.000	14	00011111011001
"'"	0.001	11	00011111001
','	0.012	6	000110
'-'	0.002	9	000111111
'.'	0.011	6	000100
':'	0.000	14	00011111011010
';'	0.001	10	0001111000
'?'	0.000	14	00011111011000
'a'	0.060	4	1001
'b'	0.013	6	100000
'c'	0.021	6	110110
'd'	0.040	5	11010
'e'	0.103	3	010
'f'	0.021	5	00000
'g'	0.016	6	100001
'h'	0.060	4	1010
'i'	0.053	4	0110
'j'	0.001	11	00011111010
'k'	0.008	7	1011000
'l'	0.030	5	10001
'm'	0.018	6	101101
'n'	0.055	4	0111
'o'	0.052	4	0011
'p'	0.011	6	000101
'q'	0.000	11	00011111000
'r'	0.039	5	10111
's'	0.048	4	0010
't'	0.078	4	1100
'u'	0.021	5	00001
'v'	0.005	7	0001110
'w'	0.021	6	110111
'x'	0.001	10	0001111001
'y'	0.009	7	1011001
'z'	0.002	9	000111101
'—'	0.000	12	000111110111

1.4 Conclusions

The results show that the average number of bits per symbol is very close to the entropy, indicating the efficiency of Huffman coding. Compared to ASCII coding, Huffman coding achieves significant compression by assigning shorter codewords to more frequent symbols and longer codewords to less frequent ones. Additionally, the Huffman codes are prefix-free, making them uniquely decodable and suitable for lossless data compression.

1.5 References

- https://en.wikipedia.org/wiki/Huffman_coding
- https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
- https://www.w3schools.com/dsa/dsa_ref_huffman_coding.php
- https://www.youtube.com/watch?v=acEaM2W-Mfw
- https://www.youtube.com/watch?v=0kNXhFIEd_w

2. Appendix

```
from docx import Document
from collections import Counter
import heapq
from math import log2


# Read the full story from a .docx file, including paragraphs and tables
def read_story(filename):
    doc = Document(filename)
    content = []

    # Extract paragraphs
    for i, paragraph in enumerate(doc.paragraphs):
        if paragraph.text.strip():
            content.append(paragraph.text.strip())

    # Extract tables
    for table_index, table in enumerate(doc.tables):
        for row in table.rows:
            for cell in row.cells:
                cell_text = cell.text.strip()
                if cell_text:
                    content.append(cell_text)

    # Combine all text into a single string
    full_text = "\n".join(content)
    return full_text


# Preprocess text
def preprocess_text(text):
    text = text.lower()
    text = text.replace("\n", "")  # Remove newline characters
    return text


# Calculate character frequencies
def calculate_frequencies(text):
    return Counter(text)


# Calculate probabilities
def calculate_probabilities(frequencies, total_chars):
    return {char: freq / total_chars for char, freq in frequencies.items()}


# Calculate entropy
def calculate_entropy(probabilities):
    return -sum(p * log2(p) for p in probabilities.values() if p > 0)


# Build Huffman tree and generate codes
def build_huffman_tree(frequencies):
    heap = [[weight, [char, ""]] for char, weight in frequencies.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))


# Calculate bits for ASCII and Huffman
def calculate_bits(frequencies, huffman_codes):
    total_ascii_bits = sum(freq * 8 for freq in frequencies.values())
```

```
    total_huffman_bits = sum(frequencies[char] * len(code) for char, code in huffman_codes)
    return total_ascii_bits, total_huffman_bits


# Main function
def main():
    file_path = r"C:\\Users\\hp\Downloads\\BZU\\1st Sem
4th\\Coding\\To+Build+A+Fire+by+Jack+London.docx"
    text = read_story(file_path)

    text = preprocess_text(text)

    frequencies = calculate_frequencies(text)
    total_chars = sum(frequencies.values())
    probabilities = calculate_probabilities(frequencies, total_chars)
    entropy = calculate_entropy(probabilities)

    huffman_codes = build_huffman_tree(frequencies)
    total_ascii_bits, total_huffman_bits = calculate_bits(frequencies, huffman_codes)
    compression_percentage = (total_ascii_bits - total_huffman_bits) / total_ascii_bits * 100

    # Calculate average bits per character
    avg_bits_per_char = total_huffman_bits / total_chars

    # Calculate sum of probabilities
    sum_probabilities = sum(probabilities.values())

    # Calculate entropy for alphabet characters only
    alphabet_probs = {char: prob for char, prob in probabilities.items() if char.isalpha()}
    alphabet_entropy = -sum(p * log2(p) for p in alphabet_probs.values() if p > 0)

    # Calculate informational efficiency
    informational_efficiency = (entropy / avg_bits_per_char) * 100

    # Sort characters alphabetically
    sorted_huffman_codes = sorted(huffman_codes, key=lambda x: x[0])

    # Print results in the desired format
    print(f"Sum of Probabilities = {sum_probabilities:.2f}")
    print(f"Entropy for all char = {entropy:.6f}")
    print(f"Entropy for alphabet char = {alphabet_entropy:.6f}")
    print(f"For ASCII coding, num of bits = {total_ascii_bits}")
    print(f"For Huffman coding, num of bits = {total_huffman_bits}")
    print(f"Average length of the code = {avg_bits_per_char:.6f}")
    print(f"Informational Efficiency = {informational_efficiency:.2f}%")
    print(f"Percentage of Compression = {compression_percentage:.2f}%\n")

    print(f"{'Symbol':<10}{'Probability':<15}{'Length Of Codeword':<20}{'Codeword':<10}")
    for char, code in sorted_huffman_codes:
        prob = probabilities[char]
        print(f"{repr(char):<10}{prob:<15.3f}{len(code):<20}{code:<10}")


if __name__ == "__main__":
    main()
```