Faculty of Engineering & Technology

Electrical & Computer Engineering Department

INFORMATION AND CODING THEORY – ENEE5304

Course Assignment

Prepared by:

Mahmoud Hamdan 1201134

Section: 2

Yazeed Hamdan 1201133

Section: 1

Instructor: Dr. Wael Hashlamoun

Date: 26/12/2024 Birzeit

Table of Contents

Table of Figures

1. Introduction

Efficient data compression is a critical area in the field of information theory and coding. As the volume of digital data continues to grow, developing methods to minimize storage requirements and transmission costs becomes increasingly important. Huffman coding, a widely used lossless data compression algorithm, provides an effective solution by assigning shorter codes to more frequently occurring symbols and longer codes to less frequent ones.

This project focuses on implementing Huffman coding to compress the text of the short story To Build A Fire by Jack London. The goal is to simulate the process of creating Huffman codes for characters in the story, analyze the efficiency of the compression, and compare it to the standard ASCII encoding. The work aims to explore fundamental concepts of source coding, including entropy calculation, probability distribution of symbols, and compression ratio.

The project aims to understand the practical applications of Huffman coding, quantify the potential for data compression in a real-world text, and evaluate its performance compared to fixed length encoding schemes. This exploration will also highlight the relationship between information entropy and average codeword length, deepening our understanding of the principles underlying efficient data representation.

2. Method (Theoretical Background)

Huffman coding is an algorithm of lossless data compression that is based on the frequency of symbols in a dataset. It was introduced by David A. Huffman in 1952 and has since become one of the most widely used algorithms for text compression. The algorithm constructs a binary tree, called a Huffman tree, to generate variable-length codes for each symbol, ensuring that no code is a prefix of another. The tree's structure ensures that no code can be a prefix of another, which makes the algorithm uniquely decodable.[1]

The process begins by calculating the frequency of each symbol in the input data. Each symbol's occurrences are counted to determine its frequency, providing the foundation for creating an efficient coding scheme. Symbols with higher frequencies are assigned shorter binary codes, while those that occur less frequently are given longer codes to optimize overall compression.

Once frequencies are determined, all symbols are placed into a priority queue or min-heap, ordered by their frequencies. This queue ensures that symbols with the lowest frequencies are processed first during tree construction. The priority queue acts as a dynamic structure that evolves as symbols are combined into nodes.

Tree construction involves iteratively removing the two symbols with the lowest frequencies from the priority queue. These symbols are combined into a new node, where the combined frequency equals the sum of the two original frequencies. This new node is then reinserted into the priority queue. The process continues until only one node remains, representing the root of the Huffman tree.

Finally, the tree is traversed to assign binary codes to each symbol. Starting from the root, left branches represent binary 0, and right branches represent binary 1. By traversing the tree from root to leaf for each symbol, unique binary codes are generated. These codes adhere to the prefix condition, ensuring that no code is the prefix of another, thereby maintaining the integrity of the data during compression and decompression.[2]

The effectiveness of Huffman coding is measured by comparing the average codeword length to the entropy of the source. Entropy, denoted as *H*, represents the theoretical minimum average number of bits needed to encode a symbol, based on its probability of occurrence. It is calculated using the equation:

$$H = -\sum P(c).\ \log_2 P(c)$$

P(c) represents the probability of each symbol. The average codeword length using Huffman coding should ideally approach the entropy, demonstrating the algorithm's efficiency.

Huffman coding has been widely applied in file compression formats such as ZIP and GZIP and in multimedia standards like JPEG and MP3. Its adaptability and simplicity make it a fundamental technique in data compression.[3]

3. Implementation and Results

3.1 Implementation

The implementation of Huffman coding for the compression of the short story To Build a Fire by Jack London was carried out using Python. The following steps were undertaken:

3.1.1 Preprocessing

The text from *To Build a Fire* by Jack London was converted to lowercase to treat uppercase and lowercase letters as the same symbol, reducing the total number of unique characters. Newline characters (\n) were removed, while spaces were retained and represented as (space) to ensure consistent readability in the output.

3.1.2 Frequency and Probability Calculation

A frequency dictionary was created by counting the occurrences of each character in the text. From this, probabilities were calculated by dividing each character's frequency by the total number of characters in the text. These probabilities formed the basis for the entropy calculation and guided the construction of the Huffman tree.

3.1.3 Huffman Tree Construction

A Huffman tree was constructed using a priority queue (heapq), where characters with lower frequencies were prioritized. By iteratively merging nodes with the smallest frequencies, a hierarchical tree structure was formed. Binary codes were assigned to characters by traversing the tree, with shorter codes allocated to more frequent characters. The prefix-free property of Huffman codes ensured unique and efficient encoding.

3.1.4 Entropy and Compression Efficiency

The entropy of the text was calculated as the theoretical lower bound on the average number of bits per character. The average bits per character were computed using the probabilities and the lengths of the Huffman codes.

3.1.5 Bit Calculations and Compression

The total bits required to encode the text were calculated for both ASCII and Huffman coding. ASCII encoding used a fixed 8 bits per character, while Huffman coding significantly reduced this count. The compression percentage was calculated by

comparing the total bits used by Huffman coding to the bits used by ASCII, showcasing the efficiency of the compression algorithm.

3.2 Results

The number of characters in the story was found with total number of **37705** character, along with their frequency of occurrence, the probabilities of the characters in the story, and the codewords for the characters, all were found and shown in the figure below.

```
-------------------------
Total Characters: 37705
-------------------------

Character | Frequency | Probability | Codeword       | Length
-----------------------------------------------------------------
!         | 3         | 0.00008     | 00011111011011 | 14
"         | 2         | 0.00005     | 00011111011001 | 14
'         | 20        | 0.00053     | 00011111001    | 11
space     | 7048      | 0.18692     | 111            | 3
,         | 436       | 0.01156     | 000110         | 6
-         | 89        | 0.00236     | 000111111      | 9
.         | 414       | 0.01098     | 000100         | 6
:         | 2         | 0.00005     | 00011111011010 | 14
;         | 26        | 0.00069     | 0001111000     | 10
?         | 1         | 0.00003     | 00011111011000 | 14
a         | 2264      | 0.06005     | 1001           | 4
b         | 484       | 0.01284     | 100000         | 6
c         | 779       | 0.02066     | 110110         | 6
d         | 1515      | 0.04018     | 11010          | 5
e         | 3887      | 0.10309     | 010            | 3
f         | 794       | 0.02106     | 00000          | 5
g         | 620       | 0.01644     | 100001         | 6
h         | 2278      | 0.06042     | 1010           | 4
i         | 1983      | 0.05259     | 0110           | 4
j         | 20        | 0.00053     | 00011111010    | 11
k         | 304       | 0.00806     | 1011000        | 7
l         | 1127      | 0.02989     | 10001          | 5
m         | 678       | 0.01798     | 101101         | 6
n         | 2077      | 0.05509     | 0111           | 4
o         | 1971      | 0.05227     | 0011           | 4
p         | 421       | 0.01117     | 000101         | 6
q         | 17        | 0.00045     | 00011111000    | 11
r         | 1481      | 0.03928     | 10111          | 5
s         | 1795      | 0.04761     | 0010           | 4
t         | 2937      | 0.07789     | 1100           | 4
u         | 800       | 0.02122     | 00001          | 5
v         | 179       | 0.00475     | 0001110        | 7
w         | 788       | 0.02090     | 110111         | 6
x         | 34        | 0.00090     | 0001111001     | 10
y         | 356       | 0.00944     | 1011001        | 7
z         | 61        | 0.00162     | 000111101      | 9
—         | 14        | 0.00037     | 000111110111   | 12
-----------------------------------------------------------------
```

Figure 1: Total number of characters and table represents each character information

ASCII code was used, and the number of bits needed to encode the full story was found as **301640** bits. The average number of bits/character for the whole story using the Huffman code equals to **4.21854** bits/character and the entropy found with **4.17205** bits/character. The total number of bits needed to encodes the entire story using Huffman code “Nhuffman” found with **159060** bits, the percentage of compression accomplished by using the Huffman encoding as compared to ASCII code equals **47.27%**

```
Total Bits (NASCII): 301640 bits
Average Bits (Huffman): 4.21854 bits/character
Entropy: 4.17205 bits/character
Total Bits (Huffman): 159060 bits
Compression Percentage: 47.27%
```

Figure 2: Calculated Results from the code output

The table shows some of the results, the probabilities, the lengths of the codewords, and the codewords for selected symbols

```
Table for Selected Symbols:
------------------------------------------------------------
Symbol     Probability      Codeword              Length
------------------------------------------------------------
a          0.06005          1001                  4
b          0.01284          100000                6
c          0.02066          110110                6
d          0.04018          11010                 5
e          0.10309          010                   3
f          0.02106          00000                 5
m          0.01798          101101                6
z          0.00162          000111101             9
(space)    0.18692          111                   3
.          0.01098          000100                6
------------------------------------------------------------
```

Figure 3: Table for selected symbols

4. Conclusion

This project successfully demonstrated the implementation of Huffman coding to compress the text of the short story To Build A Fire by Jack London. By constructing a Huffman tree and generating prefix-free binary codes, the algorithm achieved a significant compression percentage of 47.27% compared to ASCII encoding. The entropy of the text, calculated as 4.17 bits/character, was closely aligned with the average Huffman code length of 4.22 bits/character, confirming the efficiency of the compression process and its adherence to theoretical expectations. The project emphasized the practical advantages of Huffman coding in reducing data size while maintaining data accuracy, illustrating its applicability in real-world scenarios such as file compression and multimedia storage. This work highlights Huffman coding as a key technique for optimizing storage and transmission in modern data systems.

5. References

[1] https://www.dremio.com/wiki/huffman-coding/

Accessed 26/12/2024 at 13:25

[2] https://www.vaia.com/en-us/explanations/computer-science/data-representation-in-computer-science/huffman-coding/

Accessed 26/12/2024 at 13:45

[3] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Accessed 27/12/2024 at 16:55

6. Appendix

The full python code used:

```
import heapq
import math
import docx2txt

# Read text from a .docx file
def read_text_from_docx(file_name):
    text = docx2txt.process(file_name)
    return text.lower().replace("\n", "")  # Convert to lowercase and
remove newlines

# Build a frequency dictionary
def calculate_frequency(text):
    frequency = {}
    for char in text:
        if char == ' ':
            char = '(space)'
        if char not in frequency:
            frequency[char] = 0
        frequency[char] += 1
    return frequency

# Calculate probabilities
def calculate_probabilities(frequency, total_characters):
    probabilities = {char: freq / total_characters for char, freq in
frequency.items()}
    return probabilities

# Calculate entropy
def calculate_entropy(probabilities):
    entropy = -sum(prob * math.log2(prob) for prob in
probabilities.values())
    return entropy

# Build Huffman tree
class Node:
    def __init__(self, char=None, frequency=0, left=None, right=None):
        self.char = char
        self.frequency = frequency
        self.left = left
        self.right = right

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

def build_huffman_tree(frequency):
```

```
    heap = [Node(char, freq) for char, freq in frequency.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(frequency=left.frequency + right.frequency,
left=left, right=right)
        heapq.heappush(heap, merged)

    return heap[0]

# Generate Huffman codes
def generate_huffman_codes(node, code="", codes=None):
    if codes is None:
        codes = {}
    if node.char is not None:
        codes[node.char] = code
    if node.left:
        generate_huffman_codes(node.left, code + "0", codes)
    if node.right:
        generate_huffman_codes(node.right, code + "1", codes)
    return codes

# Calculate average bits per character
def calculate_average_bits(probabilities, codes):
    average_bits = sum(probabilities[char] * len(code) for char, code in
codes.items())
    return average_bits

# Calculate total bits for Huffman and ASCII
def calculate_total_bits(frequency, codes):
    nhuffman = sum(frequency[char] * len(code) for char, code in
codes.items())
    nascii = sum(frequency.values()) * 8  # ASCII uses 8 bits per character
    return nascii, nhuffman

# Main function
if __name__ == "__main__":
    file_name = " "The_Path"
    text = read_text_from_docx(file_name)

    # Step 1: Calculate frequency and total characters
    frequency = calculate_frequency(text)
    total_characters = sum(frequency.values())

    # Step 2: Calculate probabilities and entropy
    probabilities = calculate_probabilities(frequency, total_characters)
```

```
    entropy = calculate_entropy(probabilities)

    # Step 3: Build Huffman tree and generate codes
    huffman_tree = build_huffman_tree(frequency)
    codes = generate_huffman_codes(huffman_tree)

    # Step 4: Calculate average bits and total bits
    average_bits = calculate_average_bits(probabilities, codes)
    nascii, nhuffman = calculate_total_bits(frequency, codes)

    # Step 5: Calculate compression percentage
    compression_percentage = (nascii - nhuffman) / nascii * 100

    # Display results
    print("\n\n-------------------------")
    print("Total Characters:", total_characters)
    print("-------------------------")
    # Display a unified table of results
    print("\nCharacter | Frequency | Probability | Codeword       |
Length")
    print("-----------------------------------------------------------------
--")
    for char, code in sorted(codes.items()):
        char_display = char if char != "(space)" else "space"
        print(f"{char_display:<10} | {frequency[char]:<10} |
{probabilities[char]:.5f}     | {code:<12} | {len(code):<6}")
    print("-----------------------------------------------------------------
--")
    print("Total Bits (NASCII): {:.0f} bits".format(nascii))
    print("Average Bits (Huffman): {:.5f}
bits/character".format(average_bits))
    print("Entropy: {:.5f} bits/character".format(entropy))
    print("Total Bits (Huffman): {:.0f} bits".format(nhuffman))
    print("Compression Percentage: {:.2f}%".format(compression_percentage))
    # Table for specific symbols
    print("\nTable for Selected Symbols:")
    print("-----------------------------------------------------------------
")
    # Adjust the column widths
    print("{:<10} {:<16} {:<18} {:>10}".format("Symbol", "Probability",
"Codeword", "Length"))
    print("-----------------------------------------------------------------
")
    for char in ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', '(space)', '.']:
        if char in codes:
            print(f"{char:<10} {probabilities[char]:<16.5f}
{codes[char]:<18} {len(codes[char]):>10}")
```

```
    print("------------------------------------------------------------------
\n\n")
```