Department of Electrical & Computer Engineering

INFORMATION AND CODING THEORY

ENEE5304

Assignment: Huffman Code

Prepared by :

Names: Abdelrhman Abed	**ID:**1193191
Ody Shbayeh	**ID:**1201462

Instructor:

Dr. Wael Hashlamoun

Sec: 2

Date: 10/1/2025

Table of Contents

Table of Figures

1.Introduction

A major problem in computer science and information theory is data compression, which aims to reduce the quantity of data while preserving the most information. Huffman coding[1], first presented by David A. Huffman in 1952, is a traditional method of lossless data compression. Using the frequencies or probabilities of a collection of symbols, this technique creates the best prefix-free code. In this paper, we analyze Jack London's short narrative To Build A Fire using Huffman coding. The process consists of the following steps: (1) lowercase text conversion; (2) character frequency analysis; (3) probability computation; (4) Huffman tree construction; and (5) entropy and compression metrics evaluation.

There are many sections in the report. After introducing the problem, we describe the theoretical underpinnings and approach, share our results and analysis, and close out with a conversation. The Appendix contains the Python code needed to implement Huffman coding [2].

2. Method (Theoretical Background)

2.1 Frequency Counting and Probabilities

To make sure there is no difference between capital and lowercase letters, we start by reading the text and changing all of the characters to lowercase. For uniformity, newline characters are also eliminated. use the collections in Python.We determine the frequency of every character in the text using Counter[2]. Let's:

$$\text{Freq}(c)=\text{number of occurrences of character } c,\ N= \text{summation } (\text{Freq}(c))$$

where N is the total number of characters in the text. The probability of each character c is then

$$P(c) = \text{Freq}(c)/N$$

2.2 Entropy

The **entropy** H of the character set gives a theoretical lower bound (in bits) on the average code length[1]:

$$H = - \text{summtion } (P(c) \log2(p(c)))$$

A higher entropy indicates a more uniform distribution, implying we need more bits per character on average.

2.3 Huffman Tree Construction

A **Huffman tree** is built by:

The first step in building a Huffman tree is to create a leaf node for every character, whose frequency matches the character's occurrence in the input data. The structure then guarantees effective access to the node with the lowest frequency when these nodes are added to a min-heap.

The two nodes with the lowest frequencies are continually eliminated from the min-heap as the algorithm runs. The frequency of the new node that results from the merger of these two nodes is equal to the sum of their frequencies. With one node identified as the left child and the other as the right child, this new node becomes the parent of the two removed nodes. The min-heap gets refilled with the new node. Until there is only one node left in the heap, this process is repeated.

The Huffman tree's root is the last node that remains. Starting at the root of the tree, the Huffman codes are derived. The binary digits 0 and 1 are assigned to the left and right branches, respectively, during the traversal. The binary code for the relevant character is defined by the path from the root to the leaf node when it is encountered.

By guaranteeing that no code is a prefix of another, this technique ensures prefix-free codes. This characteristic is essential for allowing compressed data to be decoded without uncertainty.

2.4 Compression Metrics

- ASCII Bits: We assume 8 bits per character.

$$N\ ASCII = 8 * N$$

- Huffman Bits: Once a code is assigned to each character ccc, the number of bits for that character is length(code(c)). The average bits per character is:

$$\text{Summation } P(c) * \text{length(code(c))}.$$

The **total** bits for Huffman is thus:

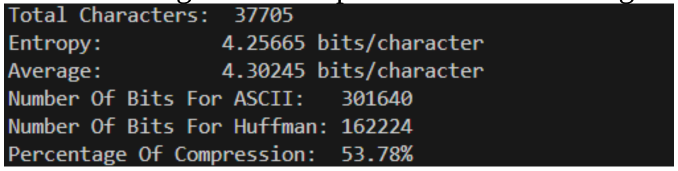
N (Huffman) = N*(average bits per character)

- **Compression** (%): The percentage saved over ASCII encoding is

(1-(N (Huffman)/ N ASCII) *100%)

3. Results and Analysis

The application of Huffman coding to the text produced the following outcomes:



```
Total Characters:  37705
Entropy:           4.25665 bits/character
Average:           4.30245 bits/character
Number Of Bits For ASCII:   301640
Number Of Bits For Huffman: 162224
Percentage Of Compression:  53.78%
```

Figure 1:Huffman Coding Results

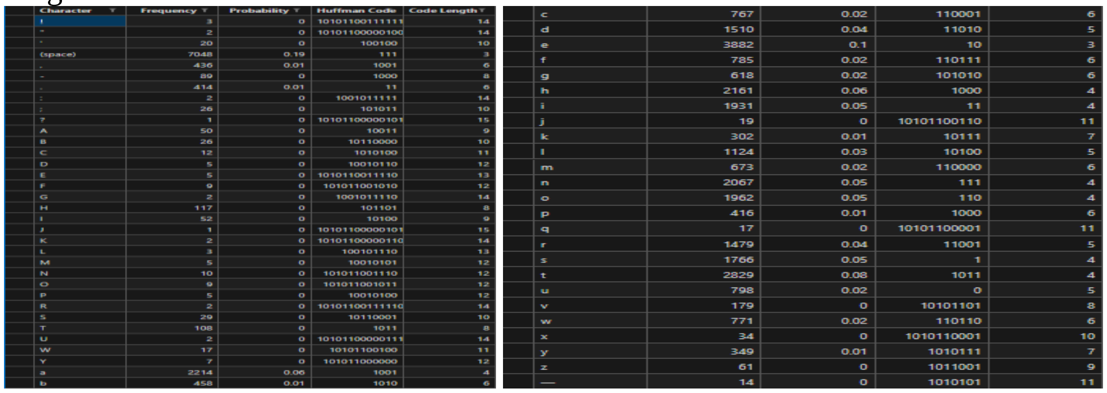
These findings highlight the effectiveness of Huffman coding in compressing data. The compression ratio indicates that the size of the Huffman-encoded data is significantly smaller compared to ASCII encoding. Additionally, the average bits per character using Huffman coding closely approach the theoretical entropy, reflecting an efficient compression process near the lower bound.

Analysis of Huffman Codes

The Huffman encoding generated codes of different lengths based on character frequencies. For instance, frequent characters, such as spaces, were assigned shorter codes like '111', while less common characters received longer codes. This variable-length encoding is the cornerstone of Huffman coding's ability to optimize data representation based on actual character distributions.

Comparison with ASCII

When comparing ASCII and Huffman encoding, it is evident that Huffman coding achieves superior compression. ASCII's fixed-length encoding assigns the same number of bits to all characters, regardless of their frequency, resulting in a larger representation. In contrast, Huffman coding adapts to the frequency distribution of characters, making it more efficient for compressing texts with uneven character usage.



Character	Frequency	Probability	Huffman Code	Code Length
!	3	0	10101100111111	14
"	2	0	10101100000100	14
'	20	0	100100	10
(space)	7048	0.19	111	3
,	436	0.01	1001	6
-	89	0	1000	8
.	414	0.01	11	6
:	2	0	1001011111	14
;	26	0	101011	10
?	1	0	10101100000101	15
A	50	0	10011	9
B	26	0	10110000	10
C	12	0	1010100	11
D	5	0	10010110	12
E	5	0	1010110011110	13
F	9	0	101011001010	12
G	2	0	1001011110	14
H	117	0	101101	8
I	52	0	10100	9
J	1	0	10101100000101	15
K	2	0	10101100000110	14
L	3	0	100101110	13
M	5	0	10010101	12
N	10	0	101011001110	12
O	9	0	101011001011	12
P	5	0	10010100	12
R	2	0	10101100111110	14
S	29	0	10110001	10
T	108	0	1011	8
U	2	0	10101100000111	14
W	17	0	10101100100	11
Y	7	0	101011000000	12
a	2214	0.06	1001	4
b	458	0.01	1010	6
c	767	0.02	110001	6
d	1510	0.04	11010	5
e	3882	0.1	10	3
f	785	0.02	110111	6
g	618	0.02	101010	6
h	2161	0.06	1000	4
i	1931	0.05	11	4
j	19	0	10101100110	11
k	302	0.01	10111	7
l	1124	0.03	10100	5
m	673	0.02	110000	6
n	2067	0.05	111	4
o	1962	0.05	110	4
p	416	0.01	1000	6
q	17	0	10101100001	11
r	1479	0.04	11001	5
s	1766	0.05	1	4
t	2829	0.08	1011	4
u	798	0.02	0	5
v	179	0	10101101	8
w	771	0.02	110110	6
x	34	0	1010110001	10
y	349	0.01	1010111	7
z	61	0	1011001	9
—	14	0	1010101	11

Figure 2:Symbol, Frequency, Probability, Huffman Code, Code Length

4. Conclusions

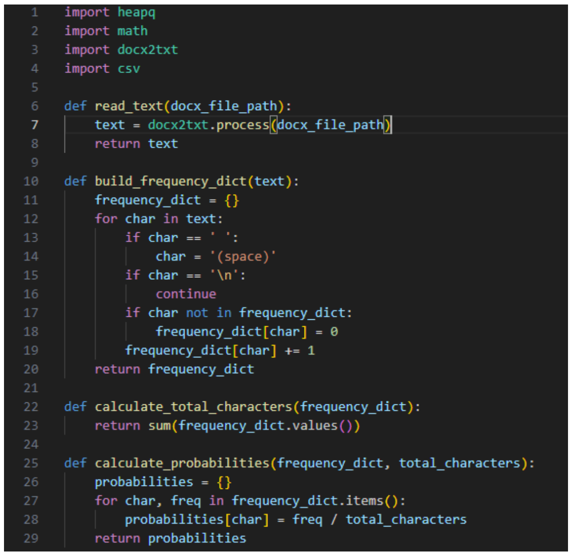
A straightforward but efficient technique for accomplishing lossless text compression is Huffman coding. It creates a binary tree by examining the frequency of characters in a dataset. In order to optimize data representation, this tree gives longer binary codes to less common characters and shorter codes to more common ones. By contrasting the bit usage of its prefix-free codes with that of fixed-length encodings like ASCII, one can evaluate the effectiveness of Huffman coding. Metrics that represent the theoretical bounds of compression for the dataset, such as the compression ratio and entropy, can also be used to gauge its effectiveness. Huffman coding is not always the best option, even though it works very well for many kinds of data. Huffman coding is frequently outperformed by modern compression algorithms by using methods like multi-pass processing, dictionary-based encoding, and contextual modeling [2]. Notwithstanding these developments, Huffman coding is still a fundamental illustration of ideal prefix-free encoding and continues to serve as a key concept in understanding compression methods.

References

1. M. Boutsikas, “On Huffman Coding for Large Alphabets,” *Information Processing Letters*, vol. 173, 2022.
2. K. Faloutsos, “A Practical Guide to Huffman Coding,” *IEEE Access*, vol. 8, pp. 168–179, 2020.

Appendix

This code is written in Python and covers the integration of all the requirements:



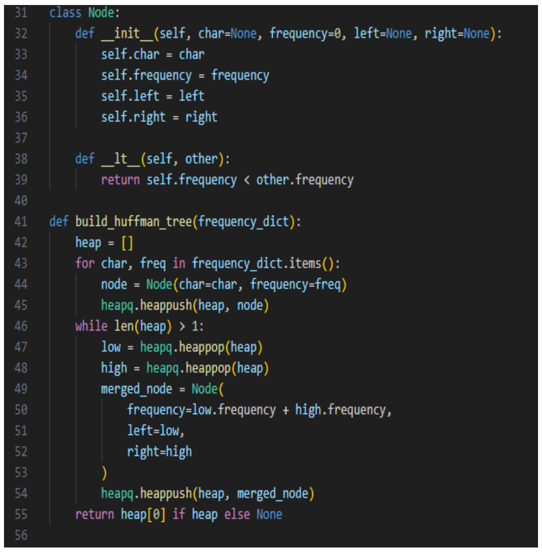
```python
import heapq
import math
import docx2txt
import csv

def read_text(docx_file_path):
    text = docx2txt.process(docx_file_path)
    return text

def build_frequency_dict(text):
    frequency_dict = {}
    for char in text:
        if char == ' ':
            char = '(space)'
        if char == '\n':
            continue
        if char not in frequency_dict:
            frequency_dict[char] = 0
        frequency_dict[char] += 1
    return frequency_dict

def calculate_total_characters(frequency_dict):
    return sum(frequency_dict.values())

def calculate_probabilities(frequency_dict, total_characters):
    probabilities = {}
    for char, freq in frequency_dict.items():
        probabilities[char] = freq / total_characters
    return probabilities
```

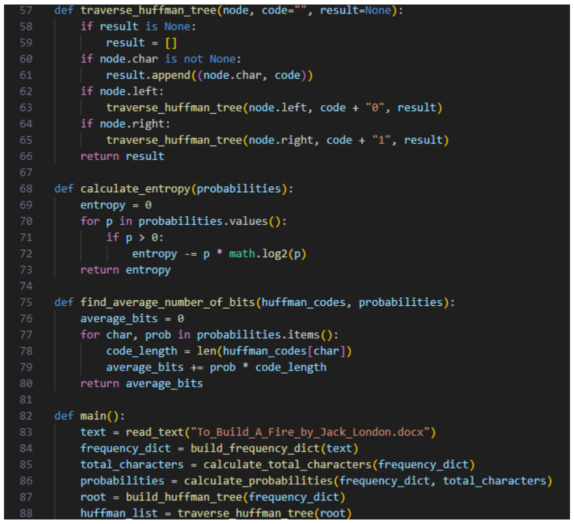


```python
class Node:
    def __init__(self, char=None, frequency=0, left=None, right=None):
        self.char = char
        self.frequency = frequency
        self.left = left
        self.right = right

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

def build_huffman_tree(frequency_dict):
    heap = []
    for char, freq in frequency_dict.items():
        node = Node(char=char, frequency=freq)
        heapq.heappush(heap, node)
    while len(heap) > 1:
        low = heapq.heappop(heap)
        high = heapq.heappop(heap)
        merged_node = Node(
            frequency=low.frequency + high.frequency,
            left=low,
            right=high
        )
        heapq.heappush(heap, merged_node)
    return heap[0] if heap else None

```

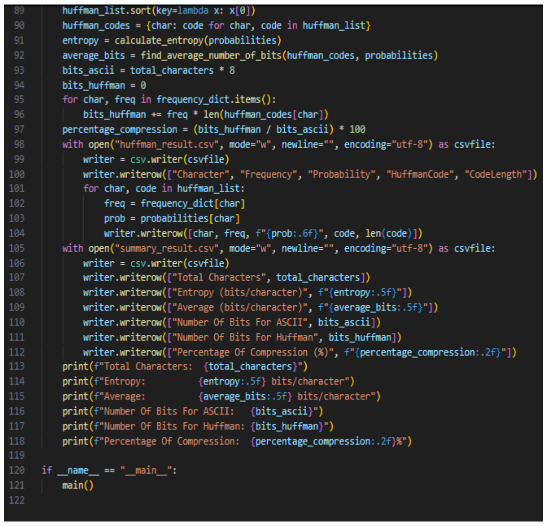


```python
def traverse_huffman_tree(node, code="", result=None):
    if result is None:
        result = []
    if node.char is not None:
        result.append((node.char, code))
    if node.left:
        traverse_huffman_tree(node.left, code + "0", result)
    if node.right:
        traverse_huffman_tree(node.right, code + "1", result)
    return result

def calculate_entropy(probabilities):
    entropy = 0
    for p in probabilities.values():
        if p > 0:
            entropy -= p * math.log2(p)
    return entropy

def find_average_number_of_bits(huffman_codes, probabilities):
    average_bits = 0
    for char, prob in probabilities.items():
        code_length = len(huffman_codes[char])
        average_bits += prob * code_length
    return average_bits

def main():
    text = read_text("To_Build_A_Fire_by_Jack_London.docx")
    frequency_dict = build_frequency_dict(text)
    total_characters = calculate_total_characters(frequency_dict)
    probabilities = calculate_probabilities(frequency_dict, total_characters)
    root = build_huffman_tree(frequency_dict)
    huffman_list = traverse_huffman_tree(root)
```



```python
    huffman_list.sort(key=lambda x: x[0])
    huffman_codes = {char: code for char, code in huffman_list}
    entropy = calculate_entropy(probabilities)
    average_bits = find_average_number_of_bits(huffman_codes, probabilities)
    bits_ascii = total_characters * 8
    bits_huffman = 0
    for char, freq in frequency_dict.items():
        bits_huffman += freq * len(huffman_codes[char])
    percentage_compression = (bits_huffman / bits_ascii) * 100
    with open("huffman_result.csv", mode="w", newline="", encoding="utf-8") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["Character", "Frequency", "Probability", "HuffmanCode", "CodeLength"])
        for char, code in huffman_list:
            freq = frequency_dict[char]
            prob = probabilities[char]
            writer.writerow([char, freq, f"{prob:.6f}", code, len(code)])
    with open("summary_result.csv", mode="w", newline="", encoding="utf-8") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["Total Characters", total_characters])
        writer.writerow(["Entropy (bits/character)", f"{entropy:.5f}"])
        writer.writerow(["Average (bits/character)", f"{average_bits:.5f}"])
        writer.writerow(["Number Of Bits For ASCII", bits_ascii])
        writer.writerow(["Number Of Bits For Huffman", bits_huffman])
        writer.writerow(["Percentage Of Compression (%)", f"{percentage_compression:.2f}"])
    print(f"Total Characters:  {total_characters}")
    print(f"Entropy:          {entropy:.5f} bits/character")
    print(f"Average:          {average_bits:.5f} bits/character")
    print(f"Number Of Bits For ASCII:   {bits_ascii}")
    print(f"Number Of Bits For Huffman: {bits_huffman}")
    print(f"Percentage Of Compression:  {percentage_compression:.2f}%")

if __name__ == "__main__":
    main()

```