Faculty of Engineering & Technology
Electrical & Computer Engineering Department

INFORMATION AND CODING THEORY ENEE5304

“Course Project”

Prepared by:

Musab Masalmah 1200078
Abdalkarim Eiss 1200015

Instructor: Dr. Wael Hashlamoun

Section: 2

Date: 25-12-2024

Table of Contents

List of Figures

1. Introduction

Coding theory is crucial to information theory, focusing on the efficient encoding, transmission, and storage of data. One of the most important uses of coding theory is data compression, which reduces data redundancy while preserving information. This study looks into Huffman coding, a lossless data compression technique commonly employed in coding theory to enhance data representations.

This project uses Jack London's short novel "To Build a Fire" as a dataset to replicate the Huffman coding process. The analysis will include computing character frequencies, probabilities, and creating the matching Huffman codewords. Additionally, the dataset's entropy will be calculated, and the results will be compared to regular ASCII encoding to assess the compression accomplished. This practical implementation demonstrates the usefulness of coding theory in real-world applications, particularly in lowering the quantity of textual material while maintaining its integrity.

2. Theoretical background

2.1. Overview of Huffman Coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters [1].

2.2. Prefix-Free Property

One important component of Huffman coding is the use of prefix-free codes, which ensures that no code is a prefix to another. This ensures that the encoded data can be decoded uniquely and unambiguously, hence avoiding the chance of decoding errors or ambiguities [1].

2.3. Construction of the Huffman Tree

The Huffman coding algorithm constructs a **Huffman tree** to determine the variable-length codes for the characters:

1. **Initialization**: Each character is represented as a node, and these nodes are sorted by frequency [2].

2. **Tree Construction**: The two nodes with the lowest frequencies are merged into a new node, with the sum of their frequencies. This process continues until only one node remains, forming the root of the tree [2].

3. **Code Assignment**: After the tree is constructed, binary codes are assigned by traversing the tree, with 0 for left branches and 1 for right branches [2].

2.4. Efficiency and Optimality

Huffman coding is ideal for reducing the overall number of bits needed to encode the input. The temporal complexity of creating the tree is O(n log n), where n represents the number of different characters in the input. This makes the algorithm efficient and effective at data compression [3].

3. Results and Analysis

The dataset consists of a total of 37,733 characters. Below are the key results derived from applying the Huffman coding algorithm to this data.

3.1. Character Frequencies

Figure 1 shows the character frequencies that are computed and summarized.

```
Character frequencies:
Counter({' ': 7048, 'e': 3887, 't': 2937, 'h': 2278, 'a': 2264, 'n': 2077, 'i': 1983, 'o': 1971, 's': 1795, 'd': 1515, 'r': 1481, 'l': 1127, 'u': 800, 'f': 794, 'w': 788, 'c':
 779, 'm': 678, 'g': 620, 'b': 484, ',': 436, 'p': 421, '.': 414, 'y': 356, 'k': 304, 'v': 179, '-': 89, 'z': 61, 'x': 34, ';': 26, 'j': 20, "'": 20, 'q': 17, 'â': 14, '€': 14,
 '”': 14, '!': 3, ':': 2, '"': 2, '?': 1})
```

Figure 1: Character Frequencies

- The most frequent character is a space (' ') with a frequency of 7,048 occurrences, which is 18.68% of the total character count.
- Other frequent characters include 'e' (3,887 occurrences, 10.30%) and 't' (2,937 occurrences, 7.78%).

3.2. Character Probabilities

Figure 2 shows the character probabilities that are calculated based on the frequency of each character divided by the total number of characters. The probabilities provide insight into the likelihood of each character appearing in the dataset.

```
Character probabilities:
{'t': 0.0778363766464368, 'o': 0.052235443776005086, ' ': 0.18678610235072748, 'b': 0.012826968436116927, 'u': 0.021201600720854426, 'i': 0.0525534677868179, 'l': 0
 .02986775501550367, 'd': 0.040150531365118064, 'a': 0.06000053004001802, 'f': 0.021042588715448018, 'r': 0.03924946333448175, 'e': 0.10301327750245144, 'y': 0
 .00943471232078022, 'j': 0.0005300400180213606, 'c': 0.020645058701931996, 'k': 0.008056608273924681, 'n': 0.0550446558715183, 'h': 0.06037155805263297, 'g': 0
 .01643124055866218, ',': 0.011554872392865661, 'x': 0.0009010680306363l3, 'w': 0.02088357671004161, 'm': 0.017968356610924125, 's': 0.04757109161741711, '-': 0
 .002358678080195055, 'v': 0.004743858161291l776, 'p': 0.011157342379349641, '.': 0.010971828373042164, "'": 0.0005300400180213606, 'z': 0.00161662205496515, 'â': 0
 .0003710280126149524, '€': 0.0003710280126149524, '”': 0.0003710280126149524, ';': 0.0006890520234277688, 'q': 0.0004505340153181565, '?': 2.650200090106803e-05, '!':
 7.95060027032041e-05, ':': 5.300400180213606e-05, '"': 5.300400180213606e-05}
```

Figure 2: Character Probabilities

The character probabilities are as follows:

- The highest probability is for the space character (' ') with a probability of 0.1868.
- The lowest probability is for special characters such as ?, '!', and : with extremely low probabilities of 0.000027, 0.000080, and 0.000053, respectively.

3.3. Huffman Codes

Figure 3 shows the Huffman codes that are assigned to each character based on their probabilities. The space character (' ') has the Huffman code 111, which is relatively short, reflecting its high frequency.

- Characters with lower frequencies such as ? and '!' have longer codes to ensure optimal compression (e.g., ? has the Huffman code 1000000101000).

	Character	Frequency	Probability	Huffman Code
0	t	2937	0.077836	1100
1	o	1971	0.052235	0011
2		7048	0.186786	111
3	b	484	0.012827	000111
4	u	800	0.021202	00001
5	i	1983	0.052553	0110
6	l	1127	0.029868	10001
7	d	1515	0.040151	11010
8	a	2264	0.060001	1001
9	f	794	0.021043	00000
10	r	1481	0.039249	10111
11	e	3887	0.103013	010
12	y	356	0.009435	1011001
13	j	20	0.000530	1000000100
14	c	779	0.020645	110110
15	k	304	0.008057	1000001
16	n	2077	0.055045	0111
17	h	2278	0.060372	1010
18	g	620	0.016431	100001
19	,	436	0.011555	000110
20	x	34	0.000901	1011000010
21	w	788	0.020884	110111
22	m	678	0.017968	101101
23	s	1795	0.047571	0010
24	-	89	0.002359	10000000
25	v	179	0.004744	10110001
26	p	421	0.011157	000101
27	.	414	0.010972	000100
28	'	20	0.000530	10110000111

20	x	34	0.000901	1011000010
21	w	788	0.020884	110111
22	m	678	0.017968	101101
23	s	1795	0.047571	0010
24	-	89	0.002359	10000000
25	v	179	0.004744	10110001
26	p	421	0.011157	000101
27	.	414	0.010972	000100
28	'	20	0.000530	10110000111
29	z	61	0.001617	101100000
30	â	14	0.000371	10000001011
31	€	14	0.000371	10000001111
32	”	14	0.000371	10000001110
33	;	26	0.000689	1000000110
34	q	17	0.000451	10110000110
35	?	1	0.000027	1000000101000
36	!	3	0.000080	1000000101011
37	:	2	0.000053	1000000101010
38	"	2	0.000053	1000000101001

Figure 3: Huffman Codes

3.4. Entropy

Figure 4 shows the entropy of the dataset which is calculated to be **4.1785 bits per character**, which represents the theoretical minimum number of bits required to encode each character without loss of information. This value suggests that the dataset has a reasonable amount of redundancy, which is expected for natural language text.

```
Entropy: 4.1785 bits/character
```

Figure 4: Entropy

3.5. Average Bits per Character

Figure 5 shows the average number of bits per character when Huffman coding is **4.2248 bits** is applied, which is slightly higher than the entropy. This slight increase in bits is due to the overhead introduced by encoding all characters into variable-length codes.

```
Average bits/character with Huffman: 4.2248
```

Figure 5: Average Bits per Character

3.6. Total Bits: ASCII vs Huffman

- **Total bits using ASCII**: 301,864 bits
- **Total bits using Huffman**: 159,416 bits

Figure 6 shows the total bits that is by ASCII vs these that is by Huffman. The compression achieved using Huffman coding is significant, reducing the total number of bits by approximately **47.19%**. This demonstrates the effectiveness of Huffman coding in reducing file size while maintaining lossless compression.

```
Total bits (ASCII): 301864



Total bits (Huffman): 159416
```

Figure 6: Total Bits: ASCII vs Huffman

3.7. Compression Percentage

Figure 7 shows the compression percentage. It achieved is **47.19%**, which indicates the efficiency of Huffman coding for this particular dataset. This compression percentage is a clear indication of the effectiveness of the algorithm in reducing the file size without any data loss.

```
Compression percentage: 47.19%
```

Figure 7: Compression Percentage

4. Conclusion

In conclusion, the results show how Huffman coding, which assigns shorter codes to more common characters, dramatically reduces the quantity of the data. The algorithm's efficiency in compressing text data is demonstrated by the compression percentage of 47.19%. The optimality of Huffman coding in reducing the number of bits needed for encoding is further highlighted by the entropy of 4.1785 bits per character.

According to these findings, Huffman coding is an effective technique for condensing big datasets, particularly those that have a lot of repetition, like natural language text.

5. References

[1] "GeeksforGeeks," 2024. [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. [Accessed 25 12 2024].

[2] "Javatpoint," 2024. [Online]. Available: https://www.javatpoint.com/huffman-coding-algorithm. [Accessed 25 12 2024].

[3] "Wscubetech," 2024. [Online]. Available: https://www.wscubetech.com/resources/dsa/huffman-code. [Accessed 25 12 2024].

6. Appendix

The code:

```
import heapq
from collections import Counter, defaultdict
import math

# Step 1: Read the text file (story content should be in a file named
"to_build_a_fire.txt")
with
open(r"C:\Users\OMEN\OneDrive\Desktop\Last_Semester\CODING\Project\Story.txt",
"r") as file:
    story = file.read().lower()


# Step 2: Calculate character frequency
frequency = Counter(c for c in story if c.isprintable() and c != '\n')  # Ignore
non-printable characters
total_chars = sum(frequency.values())

# Step 3: Calculate probabilities
probabilities = {char: freq / total_chars for char, freq in frequency.items()}

# Step 4: Compute entropy
entropy = -sum(p * math.log2(p) for p in probabilities.values())

# Step 5: Huffman encoding
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

def build_huffman_tree(frequencies):
```

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

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = HuffmanNode(None, node1.freq + node2.freq)
        merged.left = node1
        merged.right = node2
        heapq.heappush(heap, merged)

    return heap[0]

def generate_huffman_codes(tree):
    codes = {}

    def traverse(node, code):
        if node:
            if node.char is not None:
                codes[node.char] = code
            traverse(node.left, code + "0")
            traverse(node.right, code + "1")

    traverse(tree, "")
    return codes

huffman_tree = build_huffman_tree(frequency)
huffman_codes = generate_huffman_codes(huffman_tree)

# Step 6: Calculate average number of bits per character using Huffman
average_bits_per_char = sum(probabilities[char] * len(huffman_codes[char]) for
char in huffman_codes)

# Step 7: Calculate total bits using ASCII and Huffman
nascii = total_chars * 8  # 8 bits per character in ASCII
nhuffman = sum(frequency[char] * len(huffman_codes[char]) for char in
huffman_codes)

# Step 8: Compression percentage
compression_percentage = 100 * (1 - (nhuffman / nascii))

# Display results
print(f"Total characters: {total_chars} \n\n\n")
print(f"Character frequencies:\n{frequency}\n\n\n")
print(f"Character probabilities:\n{probabilities}\n\n\n")
print(f"Entropy: {entropy:.4f} bits/character\n\n\n")
print(f"Huffman Codes:\n{huffman_codes}\n\n\n")
print(f"Average bits/character with Huffman: {average_bits_per_char:.4f}\n\n\n")
print(f"Total bits (ASCII): {nascii}\n\n\n")
print(f"Total bits (Huffman): {nhuffman}\n\n\n")
print(f"Compression percentage: {compression_percentage:.2f}%\n\n\n")

# Tabulate results
import pandas as pd

table_data = {
    "Character": list(frequency.keys()),
```

```
    "Frequency": list(frequency.values()),
    "Probability": [probabilities[char] for char in frequency.keys()],
    "Huffman Code": [huffman_codes[char] for char in frequency.keys()]
}
df = pd.DataFrame(table_data)
print(df)
```