Electrical and Computer Systems Engineering Department

INFORMATION AND CODING THEORY (ENEE5304)

Course Project on Source Coding

Students' Names: Christina Saba - 1201255

Pierre Backleh - 1201296

Instructor: Dr. Wael Hashlamoun

Section: 1

Date: Jan. 10th 2025

List of Tables

Table of Contents

Introduction

Data compression also known as source coding or bit-rate reduction is a technique for fast data transfer and storage with two main types: lossless and lossy compression [1]. Huffman Coding is a popular and efficient data compression algorithm which works by assigning variable-length binary codes to characters based on their frequencies of occurrence. This course project applies Huffman coding to compress the text of the short story "To Build A Fire" by Jack London by analyzing the characters' frequencies, computing the entropy, generating Huffman codes, and evaluating the efficiency of the compression achieved compared to standard ASCII encoding.

Theoretical Background

Huffman Coding is a pre-fix free and optimal method developed by David Huffman to reduce data size without losing any details. The principle is based on assigning shorter codes to more frequent data symbols and longer codes to less frequent ones. It is a way to construct a code with small average codeword length as it works well for text and fax transmissions. Yet when dealing with data that has evenly distributed symbols, Huffman Coding may not achieve significant compression and always requires knowledge of the frequency occurrence of each character. [2]

- **How does Huffman coding work?**

If a string is supposed to be sent over a network, and each character occupies 8 bits then total of 8*(string length) bits are required for sending, however using Huffman encoding, it first creates a tree based on the characters frequencies then generate code for each character. Then for decoding the same tree is utilized making sure no ambiguity is occurred and any code associated with a character should not be present in the prefix of any other code. The steps in this algorithm are:

1. Calculate the frequency of each character in the string and find each one's probability.
2. Arrange the source symbols in a decreasing order of probability known as Sorting Stage.

3. The last two symbols of the lowest probability are assigned a 0&1 referred as Splitting Stage.
4. A new probability with the sum of the last two symbols tied together is formed known as Merge Stage.
5. The steps 2-4 are repeated until two symbols are ended up with and they are assigned digits 0&1.
6. Codeword Generation → The code of each source symbol is found by working backward and tracing the sequence of 0's and 1's assigned to that symbol and its successors.

Methodology

The story used is standardized and preprocessed by converting all the text into lowercase and remove all newlines. Then each character's occurrence is analyzed then the probabilities are computed by dividing each one's frequency by the total number of characters in the text which are then used for entropy and Huffman encoding calculations. The entropy was calculated to determine the theoretical minimum average code length using the formula: $H = -\sum \text{Pi} * \log2(\text{Pi})$

Huffman tree is constructed using a priority queue (min-heap) where two nodes with the lowest probabilities were merged and their probabilities are summed then the process is repeated then binary codes are generated where the encoding was achieved by replacing each character with its corresponding Huffman code, generating a compressed binary string.

Then the compression analysis was measured by computing the ASCII encoding length assuming each character used 8 bits, and Huffman encoding length by summing the product of each character's frequency and its Huffman code length as in the formula below:

$$Compression\ Rate = \left(1 - \frac{Huffman\ Encoding\ Length}{ASCII\ Length\ (Original)}\right) \times 100\%$$

Results and Analysis

- Total Number of Characters: **37676**

Table 1: Symbols, frequencies, probabilities, codewords, and length

Symbol	Frequency	Probability	codeword	Length
d	1513	0.040158	11010	5
a	2262	0.060038	1001	4
y	355	0.009422	1011001	7
(space)	7043	0.186936	111	3
h	2278	0.060463	1010	4
b	482	0.012793	100000	6
r	1480	0.039282	10111	5
o	1968	0.052235	0011	4
k	303	0.008042	1011000	7
e	3886	0.103143	010	3
n	2075	0.055075	0111	4
c	778	0.020650	110110	6
l	1125	0.029860	10001	5
g	620	0.016456	100001	6
,	436	0.011572	000110	6
x	34	0.000902	0001111001	10
i	1981	0.052580	0110	4
w	788	0.020915	110111	6
t	2936	0.077928	1100	4
m	678	0.017996	101101	6
u	799	0.021207	00001	5
s	1795	0.047643	0010	4
f	793	0.021048	00000	5
-	89	0.002362	000111111	9
v	179	0.004751	0001110	7
p	421	0.011174	000101	6
.	414	0.010988	000100	6
'	20	0.000531	00011111010	11
j	19	0.000504	00011111001	11
z	61	0.001619	000111101	9
—	14	0.000372	000111110111	12
;	26	0.000690	0001111000	10
q	17	0.000451	00011111000	11
?	1	0.000027	00011111011000	14
!	3	0.000080	00011111011011	14
:	2	0.000053	00011111011010	14
“	2	0.000053	00011111011001	14

- Entropy of the Alphabet: **4.17168266 bits/character**
- Average Bits/Character (Huffman): **4.218176027**
- NASCII (Number of Bits using ASCII): **301408 bits**
- Nhuffman (Total Bits using Huffman): **158924 bits**
- Compression Percentage: **47.2727996602%**

From the results above, the uneven distribution of character frequencies is observed, where the story contains 37676 characters excluding the new-lines and the most frequent character is the space with 18.7% assigned with low Huffman code, and the least is '?', '!' with the longest Huffman codes up to 14 bits and the shorter codes are assigned to more frequent characters. The calculated entropy of 4.1717 bits per character reflects the theoretical limit for optimal compression and as seen the actual average bits per character using Huffman coding was 4.2182, closely matching the entropy, indicating high efficiency in the coding process. In addition, Huffman encoding significantly reduced the total bits required for storage from 301,408 bits (ASCII) to 158,924 bits.

Conclusion

In conclusion, when applying Huffman Encoding to the story, the data size is reduced highlighting the efficiency of this algorithm. By analyzing character frequencies, calculating entropy, and generating Huffman codes, we achieved a compression percentage of approximately 47.27% compared to standard ASCII encoding. This represents a significant reduction in storage requirements, achieved without any loss of information. Furthermore, the compression achieved not only reduces storage requirements but also enhances data transmission efficiency, making it an ideal solution for text-based data.

References

[1] "What is Data Compression and How Does It Work," SEAGATE. [Online]. Available: https://www.seagate.com/em/en/blog/what-is-data-compression/

[2] "Huffman Coding," dremio. [Online]. Available: https://www.dremio.com/wiki/huffman-coding/

Appendix

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


# Node structure for Huffman tree
class Node(namedtuple("Node", ["char", "freq", "left", "right"])):
    def __lt__(self, other):
        return self.freq < other.freq


# Function to build the Huffman tree
def build_huffman_tree(frequencies):
    heap = [Node(char, freq, None, None) for char, freq in frequencies.items()]
    heapq.heapify(heap)

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

    return heap[0]


# Function to generate Huffman codes
def generate_huffman_codes(tree):
    codes = {}

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

    _generate_codes(tree, "")
    return codes
```

```
# Function to calculate entropy
def calculate_entropy(frequencies, total_chars):
    entropy = 0
    for freq in frequencies.values():
        prob = freq / total_chars
        entropy -= prob * math.log2(prob)
    return entropy


# Function to calculate encoding lengths
def calculate_lengths(frequencies, codes):
    huffman_length = sum(frequencies[char] * len(code) for char, code in codes.items())
    return huffman_length


# Function to calculate average bits per character
def calculate_avg_bits_per_char(huffman_length, total_chars):
    return huffman_length / total_chars


# Function to calculate compression percentage
def calculate_compression(ascii_length, huffman_length):
    return ((ascii_length - huffman_length) / ascii_length) * 100


# Load the text file and preprocess
file_path = "To_Build_A_Fire.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text = file.read().lower().replace("\n", "")  # Convert to lowercase and remove newlines

# Count all characters, including spaces and punctuation
frequencies = Counter(text)
total_chars = sum(frequencies.values())

# Display character frequencies
print("Total Characters:", total_chars)
print("\nCharacter Frequency Count:")
for char, freq in frequencies.items():
    print(f"'{char}': {freq}")

# Step 2: Build Huffman tree
huffman_tree = build_huffman_tree(frequencies)

# Step 3: Generate Huffman codes
```

```
huffman_codes = generate_huffman_codes(huffman_tree)

# Step 4: Calculate entropy
entropy = calculate_entropy(frequencies, total_chars)

# Step 5: Calculate ASCII and Huffman encoding lengths
ascii_length = total_chars * 8
huffman_length = calculate_lengths(frequencies, huffman_codes)

# Step 6: Calculate average bits per character using Huffman coding
avg_bits_per_char = calculate_avg_bits_per_char(huffman_length, total_chars)

# Step 7: Calculate compression percentage
compression_percentage = calculate_compression(ascii_length, huffman_length)

# Display results
print("\nResults:")
print("Entropy (bits per character):", entropy)
print("ASCII Encoding Length (bits):", ascii_length)
print("Huffman Encoding Length (bits):", huffman_length)
print("Average Bits per Character (Huffman):", avg_bits_per_char)
print("Compression Percentage:", compression_percentage)

# Display full table of character frequencies and codes
table = [(char, freq, f"{freq / total_chars:.6f}", huffman_codes[char], len(huffman_codes[char])) for char,
freq in
        frequencies.items()]
print("\nCharacter Frequencies and Codes:")
print("Char | Frequency | Probability | Huffman Code | Code Length")
for row in table:
    print(f"{repr(row[0])}    | {row[1]}       | {row[2]}      | {row[3]}        | {row[4]}")
```