FACULTY OF ENGINEERING & TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

INFORMATION AND CODING THEORY

ENCS5304

Course Project on Huffman Code

Prepared by:

Nasser Awwad 1191484

Mazen Batrawi 1190102

Instructor: Dr. Wael Hashlamoun

Section 1

BIRZEIT

8 / January / 2024

Table of Contents

Table of Figures

I. Introduction

The objective of this project is to employ Huffman coding to encode an English story called "To Build A Fire" by Jack London. Huffman coding is an optimal prefix code algorithm used in compressing files for transmission that is invented by David A. Huffman in 1952, which uses statistical coding and produces a small average codeword length. The algorithm is represented using a Python code, that assigns variable-length codes to the characters of the story based on their frequencies, which reduces the overall number of bits needed for data representation. The story will not only be compressed, but other important details such as the average number of bits per code-word, compression percentage, and code lengths for specific characters will also be provided by the program. The details and the efficiency of the Huffman coding technique will be explored in this project.

II. Theoretical Background

Huffman coding is a variable-length prefix coding algorithm that uses statistical coding to compress data efficiently by assigning shorter codes to more frequent symbols and longer codes to less frequent ones, which leads to a significant reduction in the size of the data, making it easier to store and transmit. The process involves several steps, including creating a frequency table of the symbols in the data, sorting the symbols descending based on their frequencies. Then, building a Huffman tree based on the sorted symbols where the two symbols with the lowest frequencies are assigned a 0 and 1, and combined to create a new node that has the sum of their frequencies, which is repeated iteratively until it ends up. Finally, the code for 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.

III. Results & Analysis

The initial step of implementing the Huffman coding is conducting the frequencies of the symbols in the story and figure 1 shows some of the frequencies that were calculated.

Symbol	Frequency
a	2264
b	484
c	779
d	1515
e	3887
f	794
g	620

Figure 1: frequencies of the symbols

Furthermore, a character count was performed and the total number of characters is 37705 and the entropy of the alphabets only is equal to 4.1462 as shown in figure 2.

Total	37705
Entropy of Alphabets	4.1462

Figure 2: Total count of characters and Entropy of the alphabets

After the calculation of the Huffman code for each character, a representation of some of the characters' new binary codes is displayed in figure 3. It can be observed that fewer than 8 bits (ASCII) were utilized to represent some of the characters. Moreover, the codes presented are prefix-free.

The sum of probabilities equals to 1, which asserts that the data didn't suffer from any loss. If ASCII code was employed for representing the story, a requirement of (37705 characters * 8 bits) per character would have been necessary, which equals 301640 bits. However, with the utilization of Huffman coding, only 159060 bits were needed, which is nearly the half of the ASCII representation. The compression ratio, indicative of this reduction, is calculated which is equal to 52.732%.

Furthermore, the entropy was calculated which is equal to 4.17 and the average number of bits per character was computed to be 4.219 bits per character, which means that further optimizations can be explored in the encoding of characters, as the value of 4.219 does not align with the minimum entropy value of 4.17.

Symbol	Probability	Codeword	Codeword Length
a	0.06	0110	4
b	0.0128	011111	6
c	0.0207	001001	6
d	0.0402	00101	5
e	0.1031	101	3
f	0.0211	11111	5
m	0.018	010010	6
z	0.0016	111000010	9
space	0.1869	000	3
.	0.011	111011	6

```
Sum of probabilities = 1.0
If ASCII code is used, the number of bits needed to encode the full story = 301640
The average number of bits/character  = 4.219 Bits/Character
The entropy = 4.17 Bits/Character
The total number of bits needed to encode the entire story using Huffman code = 159060
Compression ratio = 52.732 %
```

Figure 3: Huffman Coding Results

IV. Conclusion

In conclusion, the Huffman coding process successfully reduced the overall number of bits needed for data representation in the English short story "To Build A Fire". The average number of bits for each symbol is too close to the entropy and the significant difference between ASCII and Huffman coding emphasizes the significant compression achieved and how efficient is the algorithm. The code-word lengths vary based on symbol frequencies where shorter codes to more frequent symbols and longer codes to less frequent ones, allocating fewer bits to more probable symbols.

V. References

[1] “Huffman Coding.” Wikipedia, Wikimedia Foundation, 2 Dec. 2023, https://en.wikipedia.org/wiki/Huffman_coding. Accessed 06 Jan. 2024.

[2] “Huffman Coding: Greedy Algo-3.” GeeksforGeeks, 11 Sept. 2023, www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. Accessed 01 Jan. 2024.

[3] Course Slides

VI. Appendix

```
import math
from functools import cmp_to_key

import docx2txt
from tabulate import tabulate

file_path = "To+Build+A+Fire+by+Jack+London.docx"
text = docx2txt.process(file_path)
text = text.replace('\n', '')
text = text.lower()
codes = dict()


class Node:
    def __init__(self, symbol, probability: int, left=None,
right=None):
        self.symbol = symbol
        self.probability = probability
        self.left = left
        self.right = right
        self.code = ''


def get_code(char, value=''):
    temp = value + str(char.code)
    if char.left:
        get_code(char.left, temp)
    if char.right:
        get_code(char.right, temp)
    if not char.left and not char.right:
        codes[char.symbol] = temp


def result(message_to_encode):
    encoded_string = []
    for c in message_to_encode:
        encoded_string.append(codes[c])

    string = ''.join([str(item) for item in encoded_string])
    return string


def huffman(unique_symbols, frequency):
    nodes = []
    for element in unique_symbols:
        nodes.append(Node(element, int(frequency[element])))

    if len(nodes) == 1:
        nodes[0].code = 0
```

```
    while len(nodes) > 1:
        nodes = sorted(nodes, key=lambda x: x.probability)
        right = nodes[0]
        left = nodes[1]
        left.code = 0
        right.code = 1
        new_node = Node(left.symbol + right.symbol, left.probability +
right.probability, left, right)
        nodes.remove(left)
        nodes.remove(right)
        nodes.append(new_node)

    get_code(nodes[0])
    return result(unique_symbols)


def compare(item1, item2):
    def order_key(item):
        if item[0].isalpha():
            return 0, item[0]
        else:
            return 1, item[0]

    return (order_key(item1) > order_key(item2)) - (order_key(item1) <
order_key(item2))


unique_symbols = set(text)
NASCII = len(text) * 8
count = {}

for symbol in unique_symbols:
    count[symbol] = text.count(symbol)

table_data = []
for symbol in unique_symbols:
    if symbol != ' ':
        table_data.append([symbol, count[symbol]])

table_data = sorted(table_data, key=cmp_to_key(compare))
table_data.append(['space', count[' ']])
table_data.append(['Total', len(text)])

alphabet_entropy = 0
total_alphabets = 0

for symbol in unique_symbols:
    if 'a' <= symbol <= 'z':
        total_alphabets += count[symbol]

for symbol in unique_symbols:
    if 'a' <= symbol <= 'z':
```

```
        p = count[symbol] / total_alphabets
        alphabet_entropy -= p * math.log(p, 2)

table_data.append(['Entropy of Alphabets', str(round(alphabet_entropy,
4))])
headers = ["Symbol", "Frequency"]
print(tabulate(table_data, headers, tablefmt="grid"))

encoded_message = huffman(unique_symbols, count)
average, entropy, sum_prob, Nhuffman = 0, 0, 0, 0
symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'z', ' ', '.']
table_data = []

for symbol in symbols:
    p = count[symbol] / len(text)
    length_of_codeword = len(codes[symbol])
    if symbol == ' ':
        table_data.append(['space', round(p, 4), codes[symbol],
length_of_codeword])
    else:
        table_data.append([symbol, round(p, 4), codes[symbol],
length_of_codeword])

headers = ["Symbol", "Probability", "Codeword", "Codeword Length"]
print(tabulate(table_data, headers, tablefmt="grid"))

for symbol in unique_symbols:
    p = count[symbol] / len(text)
    length_of_codeword = len(codes[symbol])
    Nhuffman += length_of_codeword * count[symbol]
    sum_prob += p
    average += p * len(codes[symbol])
    entropy -= p * math.log(p, 2)

print('Sum of probabilities = ' + str(round(sum_prob, 4)))
print("If ASCII code is used, the number of bits needed to encode the
full story = " + str(NASCII))
print("The average number of bits/character  = " + str(round(average,
3)) + " Bits/Character")
print("The entropy = " + str(round(entropy, 2)) + " Bits/Character")
print("The total number of bits needed to encode the entire story
using Huffman code = " + str(Nhuffman))
print("Compression ratio = " + str(round((Nhuffman / NASCII) * 100,
3)) + " %")
```