BIRZEIT UNIVERSITY

Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENEE5304

Projects Report

Prepared by : Tareq Shannak

ID Number : 1181404

Instructor : Dr. Wael Hashlamoun

Section : 1

Date : 8 December 2021

Table of Contents

1. Project One – Huffman Coding

1.1. Introduction

In this project, we will implement a program that encode a story in word document using Huffman code and shows us a summary for the encoding process by printing the average number of bits per code-word, the entropy, percentage of compression and probabilities, code-word lengths of certain characters.

1.2. Theoretical Background

Huffman coding depends on the probabilities of each symbol in the message that sender sends to the receiver. After the frequencies have been calculated, the symbols will ordered descending according to the frequencies in the sort stage. The two symbols with least frequencies will be merged, their frequencies will be added and each branch will have a different binary digit. The process will repeat until there is no symbols to merge, the code-word for each symbol is the binary digits on the branches from the last node we merged back to the original symbol.

The basic idea in Huffman coding is to use fewer bits for more frequently occurring characters and used for lossless data compression.

1.3. Results

The code is in appendix[1]. The output shows the average number bits for each symbol, the entropy, number of bits in ASCII coding, number of bits in Huffman coding, percentage of compression and the code-words of certain characters. The code has some commented commands that prints things may ensure that the code is correctly works.

```
Average = 4.229 Bits/Character
Entropy = 4.189 Bits/Character
For ASCII coding,   # of bits = 138152
For Huffman coding, # of bits = 331
Percentage of Compression = 0.24%
Symbol  Probability Length Of Codeword  Codeword
'A'         0.068           4           0101
'B'         0.013           6           101110
'C'         0.015           6           011010
'D'         0.038           5           00111
'E'         0.095           4           0000
'F'         0.017           6           010011
'G'         0.018           6           010010
'H'         0.053           4           1010
' '         0.188           2           11
'.'         0.009           7           0011010

Process finished with exit code 0
```

1.4. Conclusions

We can notice that the average number of bits for each symbol is too close to the entropy and the huge difference between ASCII coding and Huffman coding. The code-words have different lengths according to the frequencies for each symbol, the most probability symbol to be in the message the least bits to represent. Also, the codes are prefix-free so the code is instantaneously decodable.

1.5. References

- https://en.wikipedia.org/wiki/Huffman_coding
- https://www.youtube.com/watch?v=0kNXhFIEd_w

2. Project Two – Lempel-Ziv Encoding of Binary Data

2.1. Introduction

In this project, we will implement a program that generates a random sequence of binary data such that the probability of 1s is 95% and parses the data using LZ encoding and assign a number to each phrase. The program should outputs a table that shows the size of encoded sequence, compression ratio and number of bits per code-word for different sequence lengths.

2.2. Theoretical Background

The idea of the compression algorithm is as the input data is being processed, a dictionary keeps a correspondence between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed. Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.

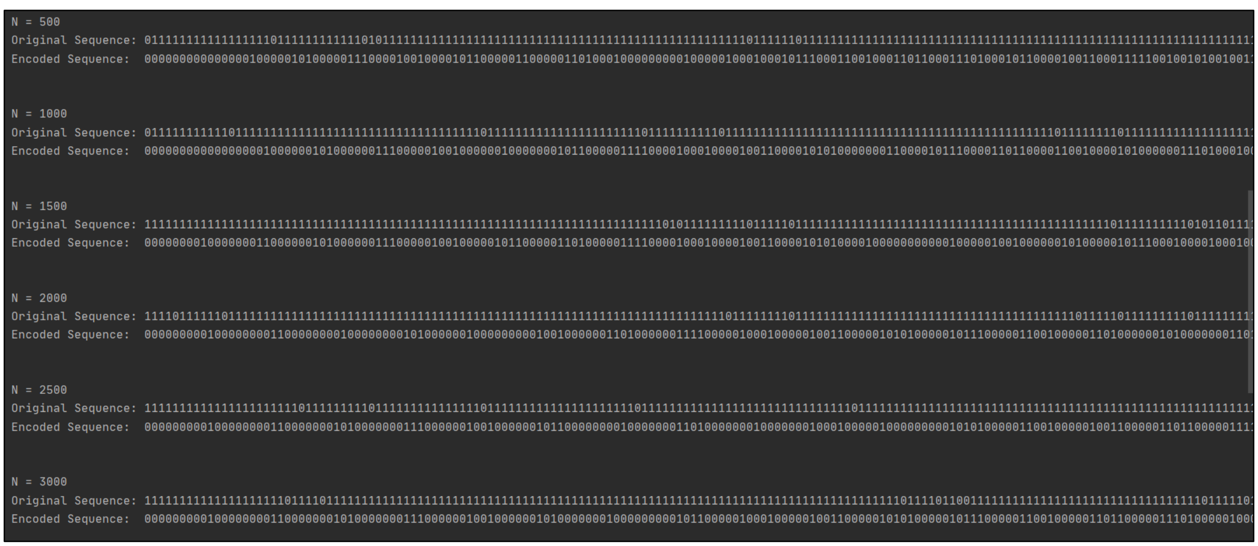
First, a dictionary is initialized to contain the phrases. The algorithm scans the input string and find the longest section that does not exist in the dictionary and move it from the input string to the dictionary, it will have an index. After all input string scanned, we can calculate the number of bits to represent each phrase. The phrases that consist of an old phrase and an additional bit will represent by the index of the old phrase and the additional bit, so it will compress the data and the compression will increase as long as the input string becomes bigger.

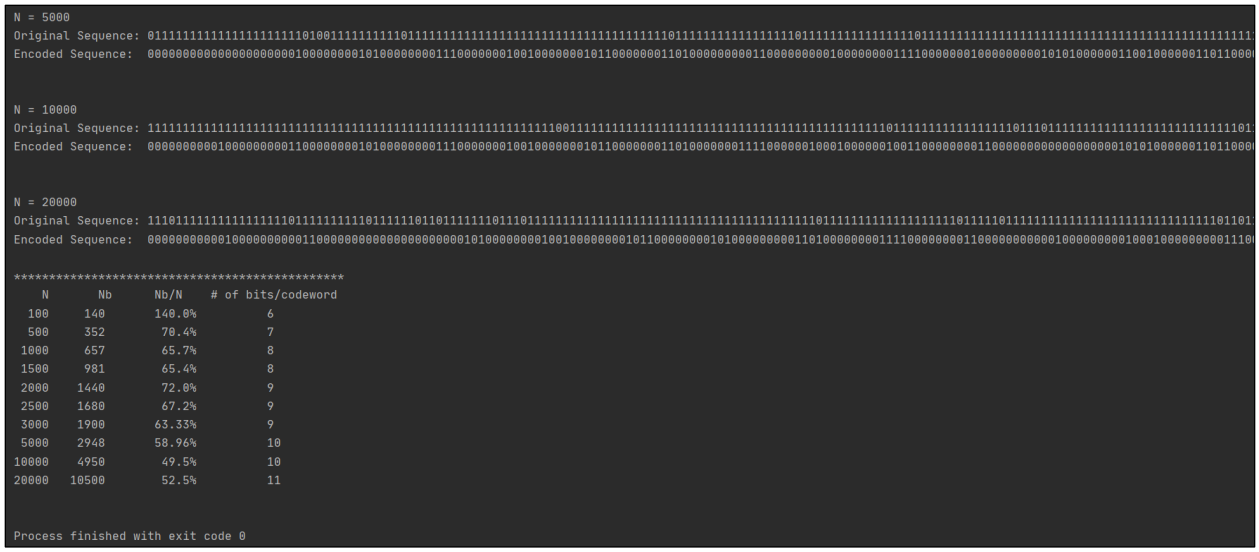
2.3. Results

The code is in appendix[2]. The program outputs the original and encoded sequences for different original sequence lengths, but for the first sequence length (N=100) it also outputs the dictionary contents and the code-words just to make sure that the process works correctly. In the last figure it outputs a table that shows the compression ratio and the number of bits per code-word for the different original sequence lengths.

```
N = 100
Original Sequence: 0111110101111101011111111111111111111111111111101111111110101111111111111101100111111111111111111111
Encoded Sequence:  00000000000001000010100001100000100000011100010010000001100011010010011001010100101110011001001000100110000001011001101100100100001000010001 1

Dictionary Location     Dictionary Contents     Codeword
      1 00001                     0             000000 0
      2 00010                     1             000000 1
      3 00011                    11             000010 1
      4 00100                   110             000011 0
      5 00101                    10             000010 0
      6 00110                   111             000011 1
      7 00111                  1101             000100 1
      8 01000                    01             000001 1
      9 01001                  1111             000110 1
     10 01010                 11111             001001 1
     11 01011                111111             001010 1
     12 01100               1111111             001011 1
     13 01101              11111111             001100 1
     14 01110                   011             001000 1
     15 01111              11111110             001100 0
     16 10000                   101             000101 1
     17 10001             111111111             001101 1
     18 10010                 11110             001001 0
     19 10011                  1100             000100 0
     20 10100            1111111111             010001 1
```





2.4. Conclusions

We can see that when the length = 100 bits, the encoded sequence length is more than the original length so there is no benefit for this encoding. As the original length increases the compression ratio decreases. As shown at N = 20000, the compression ratio equals to 52.5%. In short sentences, a saving can hardly be noticed. But in a long text, many phrases of longer lengths become more frequent, and as such these long phrases will be encoded into smaller number of bits.

The LZ code-words are prefix-free and the lengths of code-words are equals to a number that increases as the input string increases.

2.5. References

- https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
- https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/
- https://www.youtube.com/watch?v=Dn-91_Vu_aM

3. Appendix

3.1. Appendix[1]

```
import math

import docx2txt

codes = dict()


class Character:
    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 calculate_code(char, value=''):
    temp = value + str(char.code)

    if char.left:
        calculate_code(char.left, temp)
    if char.right:
        calculate_code(char.right, temp)

    if not char.left and not char.right:
        codes[char.symbol] = temp

    return codes


def output_encoded(data, coding):
    encoding_output = []
    for c in data:
        # print(coding[c], end='')
        encoding_output.append(coding[c])
    string = ''.join([str(item) for item in encoding output])
    return string


def encoding(symbols_list, frequency):
    nodes = []
    for element in symbols_list:
        nodes.append(Character(element, int(frequency[symbols_list.index(element)])))

    # for node in nodes:
    #     if node.symbol == '\n':
    #         print("'\\n' " + str(node.probability))
    #     else:
    #         print("'" + node.symbol + "' " + str(node.probability))

    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 = Character(left.symbol + right.symbol, left.probability + right.probability,
left, right)

        nodes.remove(left)
        nodes.remove(right)
        nodes.append(new_node)
```

```
    huffman_encoding = calculate_code(nodes[0])
    # print(huffman encoding)
    return output_encoded(symbols_list, huffman_encoding)


my_text = docx2txt.process("Shooting+an+elephant+by+George+Orwell+recovered.docx")
# print(my_text)
list of symbols = []
count = []
sum of characters = 0
for i in my_text:
    sum_of_characters += 1
    if i.upper() not in list_of_symbols:
        list of symbols.append(i.upper())
        count.append(1)
    else:
        count[list_of_symbols.index(i.upper())] += 1
ascii_bits = sum_of_characters * 8
average = 0
entropy = 0
encoded_output = encoding(list_of_symbols, count)

for j in list_of_symbols:
    p = count[list_of_symbols.index(j)] / sum_of_characters
    average += p * len(codes[j])
    entropy += - p * math.log(p, 2)

print("\nAverage = " + str(round(average, 3)) + " Bits/Character")
print("Entropy = " + str(round(entropy, 3)) + " Bits/Character")
# print(len(list_of_symbols))


print("For ASCII coding,   # of bits = " + str(ascii bits))
huffman_bits = 0
for i in encoded_output:
    huffman_bits += 1
print("For Huffman coding, # of bits = " + str(huffman_bits))

print("Percentage of Compression = " + str(round((huffman bits / ascii bits) * 100, 3)) + "%")

print("Symbol\tProbability\tLength Of Codeword\tCodeword")
list_selected_symbols = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', ' ', '.']
for j in list_selected_symbols:
    p = count[list_of_symbols.index(j)] / sum_of_characters
    print("'" + j + "'\t\t\t" + str(round(p, 3)) + "\t\t\t" + str(len(codes[j])) + "\t\t\t" +
codes[j])
```

3.2. Appendix[2]

```
import math
import random


def generate_sequence(limit):
    string = ''
    for i in range(0, limit):
        x = random.random() * 100
        if x < 95:
            string = string + '1'
        else:
            string = string + '0'
    return string


def parse_data(sequence):
    k = 1
    temp = ''
    for i in sequence:
        if temp in lz_dict.values() or temp == '':
```

```
            temp = temp + i
            continue
        lz_dict[k] = temp
        k += 1
        temp = i


def find phrases(num of bits):
    temp = {}
    for present value in lz dict.values():
        if len(present_value) == 1:
            temp[present_value] = '0'.rjust(num_of_bits, '0') + present_value[0]
        else:
            for past key, past value in lz dict.items():
                if present_value[:len(present_value) - 1] == past_value:
                    temp[present value] = str(bin(past key).replace("0b", "")).rjust(num of bits,
'0') \
                                          + present_value[len(present_value) - 1]
    return temp


summary output = "*******************************************\n\tN\t\tNb\t\tNb/N\t# of
bits/codeword\n"
# print("\tN\t\tNb\t\tNb/N\t# of bits per codeword")
for N in [100, 500, 1000, 1500, 2000, 2500, 3000, 5000, 10000, 20000]:
    print("\nN = " + str(N))
    lz dict = {}
    seq = generate_sequence(N)
    parse_data(seq)
    bits = math.ceil(math.log2(len(lz_dict.items()))) + 1

    # print(lz_dict)
    codewords = find phrases(bits)
    # print(codewords)
    print("Original Sequence: " + seq)
    encoded_sequence = ''.join([str(item) for item in codewords.values()])
    print("Encoded Sequence:  " + encoded_sequence + "\n")
    if N == 100:
        print("Dictionary Location\t\tDictionary Contents\t\tCodeword")
        for key, value in lz_dict.items():
            print("\t" + str(key).rjust(3) + " " + str(bin(key).replace("0b", "")).rjust(bits -
1, '0') + "\t\t"
                  + str(value).rjust(15) + "\t\t\t\t" + codewords[value][:len(codewords[value]) -
1] + " "
                  + codewords[value][len(codewords[value]) - 1])

    summary_output += str(N).rjust(5) + "\t" + str(len(encoded_sequence)).rjust(5) + "\t\t" +
str(
        round((len(encoded_sequence) / N) * 100, 2)).rjust(5) + "%\t\t\t" + str(bits) + "\n"
print(summary_output)
```