Faculty Of Engineering and Technology

Electrical and Computer Engineering Department

Information and Coding Theory ENEE5304

Project Report

Sondos Shahin 1200166

Instructor: Dr. Wael Hashlamoun

Sec. 1

Abstract

The aim of this project is first to generate a variable length random sequence from the symbols (a, b, c, d), given that each symbol has a different probability. In addition to calculating the entropy of the source.

Then, a Lempel-Ziv encoding program was implemented to parse data, and give a number for each resulting phrase. Both the number of bits needed to encode the sequence and the number of bits per symbol were found, in addition to the compression ratio relative to the ASCII code.

Moreover, a Huffman encoding program was implemented to find the codeword for each symbol, and to find the average number of bits per symbol.

Finally, a comparison between the Lempel-Ziv and the Huffman encoding methods is done.

Table of contents

Table of figures

List of tables

Theory

Lempel-Ziv encoding algorithm is considered one of the fastest compression algorithms. It does not require a priori knowledge of the input data, unlike other algorithms where certain characteristics of data need to be known before compression [1].

Lempel-Ziv compression works by reading a sequence of symbols, then grouping the symbols into different length phrases, and converting these strings into fixed length codes. Since the codes take up less space than the strings they replace, compression will be achieved [2].

Each phrase consists of a previously occurring phrase, followed by the new source output.

On the other hand, Huffman encoding algorithm assigns variable-length codes to input characters. Lengths of the assigned codes are based on the frequencies of corresponding characters, so that symbols with higher probabilities have shorter codewords [3].

In order to use Huffman encoding technique, the probabilities of the source symbols need to be known before starting the encoding.

Results and Analysis

Lempel-Ziv

In figure 1, the results show a 30-symbol randomly generated sequence, that was encoded using the Lempel-Ziv algorithm. The sequence was parsed into phrases, and each phrase was given a number. The resultant codewords are shown in the figure, in addition to the number of bits needed to encode a single symbol, which includes the number of bits used to represent the previously occurring phrases in the dictionary plus the 8-bits used to represent the new ACSII character at the end of the phrase. Finally, the number of bits required to encode the whole sequence, which depends on the number of phrases in the dictionary, and the encoded sequence itself were shown in the figure.

```
sequence is :  aababacdaaddbaaabdababababbaa
source entropy  =  1.6854752972273346
1 a
2 ab
3 aba
4 c
5 d
6 aa
7 dd
8 b
9 aaa
10 bd
11 abab
12 abb
13 abba
codewords:  {'a': '0000a', 'ab': '0001b', 'aba': '0010a', 'c': '0000c', 'd': '0000d', 'aa': '0001a', 'dd':
'0101d', 'b': '0000b', 'aaa': '0110a', 'bd': '1000d', 'abab': '0011b', 'abb': '0010b', 'abba': '1100a'}
number of bits per symbol =  5.6
number of bits to encode the sequence =  168
Encoded Sequence: 0000a0001b0010a0000c0000d0001a0101d0000b0110a1000d0011b0010b1100a
```

Figure 1- Lempel-Ziv results

Then, different length sequences were generated, and the compression ratio for the number of bits used for encoding each sequence using Lempel-Ziv algorithm relative to the ACSII code was found. Figure 2 illustrates the results.

Nb is the number of bits used to encode the sequence using Lempel-Ziv algorithm.

sequence length	Nb	compression ratio	bits per symbol
800	3088	0.4825	3.86
100	464	0.58	4.64
1000	3536	0.442	3.536
200	736	0.46	3.68
400	1520	0.475	3.8
2000	6834	0.427125	3.417
50	153	0.3825	3.06
20	85	0.53125	4.25

Figure 2- Lempel-Ziv compression ratio

Then, the average values of the number of bits to encode each sequence (Navg), and the number of bits per symbol, in addition to the compression ratio were found by repeating the previous step 5 times and taking the average of the resultant values. Figure 3 illustrates the results.

sequence length	Navg	compression ratio	bits per symbol
800	2873	0.44040625	3.59125
100	306	0.4335	3.06
1000	3485	0.43435	3.485
200	697	0.45049999999999996	3.485
400	1394	0.45049999999999996	3.485
2000	6579	0.4175625	3.2895
50	170	0.425	3.4
20	68	0.425	3.4

Figure 3- Lempel-Ziv avg compression ratio

Huffman

For this part, a Huffman encoding program was implemented to find a codeword for each source symbol. The codewords depend on the probability of each symbol. Since the symbol ‘a’ has the highest probability, the program assigned the shortest codeword to it, followed by symbol ‘b’ which has the second highest probability, and hence was assigned the second shortest codeword. Then both ‘c’ and ‘d’ have the same probability so both were given the same codeword length.

The average number of bits per codeword was found. Then for a 100-symbol sequence, the number of bits to encode the sequence using Huffman and using ACSII were found. Figure 4 shows the results.

```
Huffman codewords for each character:
a: 0
c: 100
d: 101
b: 11
Average number of bits per codeword: 1.7000
for a 100 symbol sequence:
number of bits for encoding using ASCII =  800
number of bits for encoding using Huffman =  170.00000000000003
compression ratio using Huffman =  0.21250000000000002
```

Figure 4- Huffman encoding

Comparison

After analyzing the results from all the previous sections, the following results were concluded.

Sequence length = 100 symbol	Encoded sequence size	Compression ratio	Bits per symbol
Lempel-Ziv	504	0.622	5.04
Huffman	170	0.21	1.7
ASCII	800	1	8

Table 1- comparison table

Huffman encoding achieves the most reduction in encoded size, requiring only 170 bits, and a compression ratio of 0.21 and an average of 1.7 bits per symbol. Lempel-Ziv encoding, while not as efficient as Huffman coding for this sequence, still encode much better than ASCII encoding. The Lempel-Ziv algorithm reduces the encoded size to 504 bits, with a compression ratio of 0.622 and an average of 5.04 bits per symbol. Finally, ASCII encoding, which assigns a fixed 8 bits per symbol, results in an encoded size of 800 bits for the sequence, leading to a compression ratio of 1.0 and an average of 8 bits per symbol.

Conclusion

By the end of this project, a clear understanding about the encoding algorithms was gained. And the importance and efficiency of using encoding techniques such as Lempel-Ziv and Huffman algorithms instead of using ACSII encoding in data encoding and transmission was understood.

Huffman encoding algorithm performs a good compression in the cases where the probabilities of source symbols are known. Whereas Lempel-Ziv performs highest compression when the encoded sequence length increases, since the repetition of patterns become more likely. Both algorithms perform a better reduction in the size of encoded sequences compared with encoding using ASCII.

References

[1] https://www.studysmarter.co.uk/explanations/computer-science/data-representation-in-computer-science/lempel-ziv-welch/ Accessed 9/6/2024 at 14:00.

[2] https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/ Accessed 9/6/2024 at 14:00.

[3] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/ Accessed 11/6/2024 at 19:00.

Appendix

Lempel-Ziv

lempel_ziv.py > ...

```python
from random import random
import numpy as np
import math

def generate_seq(prob_table, length):
    #fprob_table is the input probability dictionary
    seq_list = ''
    s = ''
    entropy=0
    while len(seq_list) < 1:
        for k, v in prob_table.items():
            if len(s) == length:
                seq_list = seq_list + s
                s = ''
                break
            rn = random()
            if rn <=  v:
                s += k
    return seq_list

def calculate_entropy(probabilities):
    return -sum( p * math.log2(p) for p in probabilities)

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
        #print(k, lz_dict[k])
        k +=1
        temp = i
    return lz_dict, k


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

def lempel_ziv(sequence):
    lz_dict , count = parse_data(sequence)
    num_bits = math.ceil(math.log2(len(lz_dict.items())))
    codewords = find_phrases(num_bits)
    return num_bits, codewords, count


def summarize_info(sequence_lengths,prob_table):
    print("sequence length  \t Nb  \t compression ratio  \t bits per  symbol")
    for length in sequence_lengths:
        sequence = generate_seq(prob_table,length)
        lz_dict = {}
        num_bits, codewords, count= lempel_ziv(sequence)
        seq_bits = (num_bits+8)*count
        compression_ratio = seq_bits / (length*8)
        print(length , " \t\t\t", seq_bits, "\t\t ", compression_ratio,  " \t\t ", seq_bits/length )

```

```python
def summarize_avg(sequence_lengths,prob_table):
    print("sequence length  \t Navg  \t compression ratio  \t bits per  symbol")
    for length in sequence_lengths:
      avg_bits=0
      for i in range (5):
         sequence = generate_seq(prob_table,length)
         lz_dict = {}
         num_bits, codewords, count= lempel_ziv(sequence)
         seq_bits = (num_bits+8)*count
         avg_bits += seq_bits
      avg_bits= avg_bits / 5
      compression_ratio = avg_bits / (length*8)
      print(length , " \t\t\t", seq_bits, "\t\t ", compression_ratio,  " \t\t ", seq_bits/length)


prob_table = {'a': 0.5, 'b': 0.3, 'c': 0.1, 'd': 0.1}
sequence = generate_seq(prob_table,30)
print("sequence is : " ,sequence)
entropy = calculate_entropy([0.5,0.3,0.1,0.1])
print("source entropy  = ",entropy)
lz_dict = {}
num_bits, codewords, count= lempel_ziv(sequence)
sequence_length= (num_bits+8)*count
print("codewords: ",codewords)
print("number of bits per symbol = ", sequence_length/30)
print("number of bits to encode the sequence = ", sequence_length)
encoded_sequence = ''.join([str(item) for item in codewords.values()])
print("Encoded Sequence: " + encoded_sequence + "\n")

sequence_lengths = {20,50,100,200,400,800,1000,2000}
summarize_info(sequence_lengths,prob_table)
summarize_avg(sequence_lengths,prob_table)

```

Huffman

huffman.py > ...

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

# Function to build the Huffman Tree
def build_huffman_tree(char_prob):
    # Create a list of nodes
    nodes = [Node(char, prob) for char, prob in char_prob.items()]
    # Continue until the list contains only one node
    while len(nodes) > 1:
        # Sort the list of nodes by frequency
        nodes = sorted(nodes, key=lambda x: x.freq)
        # Get the two nodes with the smallest frequencies
        left = nodes.pop(0)
        right = nodes.pop(0)
        # Create a new internal node with these two nodes as children
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        # Add the merged node back to the list
        nodes.append(merged)
    # The remaining node is the root of the Huffman Tree
    return nodes[0]

# Function to generate Huffman codes from the Huffman Tree
def generate_codes(node, prefix="", codebook={}):
    if node is not None:
        # If this is a leaf node, it contains a character
        if node.char is not None:
            codebook[node.char] = prefix
        else:
            generate_codes(node.left, prefix + "0", codebook)        # Traverse the left subtree with the prefix '0'
            generate_codes(node.right, prefix + "1", codebook)       # Traverse the right subtree with the prefix '1'
    return codebook

def calculate_avg_bits(huffman_codes, char_prob):       # Function to calculate the average number of bits per codeword
    avg_bits = 0
    for char in huffman_codes:
        prob = char_prob[char]
        code_length = len(huffman_codes[char])
        avg_bits += prob * code_length
    return avg_bits

char_prob = {
    'a': 0.5,
    'b': 0.3,
    'c': 0.1,
    'd': 0.1,
}
huffman_tree = build_huffman_tree(char_prob)
huffman_codes = generate_codes(huffman_tree)
print("Huffman codewords for each character:")
for char, code in huffman_codes.items():
    print(f"{char}: {code}")
avg_bits_per_codeword = calculate_avg_bits(huffman_codes, char_prob)
print(f"Average number of bits per codeword: {avg_bits_per_codeword:.4f}")
print("for a 100 symbol sequence: ")
print("number of bits for encoding using ASCII = " , 100*8)
print("number of bits for encoding using Huffman = " , 100*avg_bits_per_codeword)
print("compression ratio using Huffman = ", (100*avg_bits_per_codeword) / (8 * 100))

```