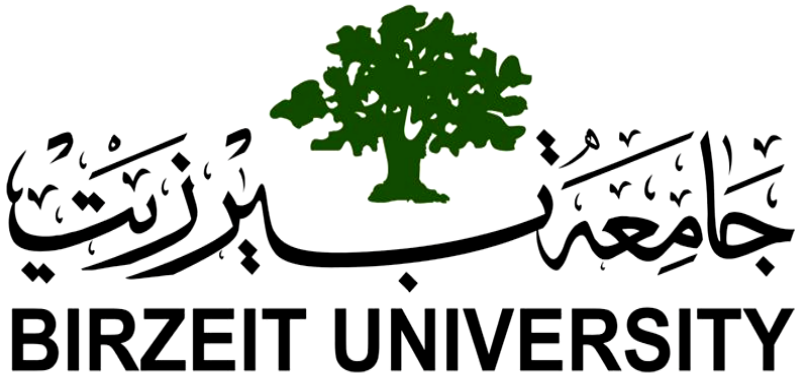

Faculty of Engineering & Technology
Electrical & Computer Engineering Department
INFORMATION AND CODING THEORY
ENEE5304
Course Assignment

Lempel-Ziv Encoding of Random Symbols

Prepared by:

Mohammad Makhamreh **1200227**

Sondos Farrah **1200905**

Instructor: **Dr. Wael Hashlamon**

Date: 14th June 2024

Abstract

This project examines Lempel-Ziv (LZ) encoding for random symbol sequences using Python. Symbols 'a', 'b', 'c', and 'd' are generated with probabilities 0.5, 0.3, 0.1, and 0.1, respectively. The source entropy is calculated, and LZ encoding is applied to sequences of varying lengths (20 to 2000 symbols). We measure the encoded sequence size, compression ratio, and bits per symbol. The performance of LZ encoding is compared with Huffman coding for a sequence length of 100 symbols. Results are summarized in tables, demonstrating the efficiency of each method and the practical benefits of data compression techniques. Python code is utilized throughout the project for simulation and analysis.

Table Of Content

List Of Figures

List Of Tables

1. Introduction

The problem addressed in this project is the efficient compression of data sequences generated from a discrete set of symbols with known probabilities. Specifically, we focus on sequences composed of the symbols 'a', 'b', 'c', and 'd' with respective probabilities of 0.5, 0.3, 0.1, and 0.1. The goal is to apply and compare two lossless data compression techniques—Lempel-Ziv (LZ) encoding and Huffman coding—to these sequences, assessing their performance in terms of compression ratio and bits per symbol. By implementing these encoding methods using Python, we aim to understand their practical effectiveness and provide insights into their respective strengths and suitability for different types of data.

2. Theory

2.1 Source Coding and Entropy

Source coding, also known as data compression, is a method used to reduce the size of data representation. The objective is to represent the information using fewer bits than the original format without losing any information. The efficiency of source coding is often evaluated using entropy, a measure of the average information content per symbol produced by a stochastic source of data. The entropy H of a discrete random variable X with possible values x1, x2, ..., xn and corresponding probabilities P(x1), P(x2),..., P(xn) is given by [1]:

$$H(S) = -\sum_{i} p(\mathrm{x}_i) \cdot log_2 p\ (\mathrm{x}_i) \qquad 1.1$$

In this project, the symbols 'a', 'b', 'c', and 'd' are generated with probabilities 0.5, 0.3, 0.1, and 0.1, respectively. The entropy of this source provides a theoretical lower bound on the average number of bits needed per symbol [1].

2.2 Lempel-Ziv (LZ) Encoding

Lempel-Ziv encoding is a dictionary-based compression algorithm that does not require prior knowledge of the source's probability distribution. Instead, it builds a dictionary of phrases dynamically as it processes the input sequence. The basic idea is to parse the input data into substrings (phrases) that have not been encountered before and assign unique codes to them. This method is particularly effective for sources with repeating patterns [2].

For a given sequence length N, the LZ encoding algorithm performs the following steps:

1. The algorithm starts with an empty dictionary.
2. Then, the algorithm reads the sequence character by character.
3. For each character, it checks if it is in the dictionary or not.

4. If the character is not in the dictionary, the algorithm adds it to the dictionary and outputs the character.
5. If the character is in the dictionary, the algorithm outputs the index of the dictionary entry that contains the character.
6. The algorithm repeats steps 2-5 until it reaches the end of the sequence.

2.3 Huffman Coding

Huffman coding is a variable-length prefix coding algorithm that assigns shorter codes to more frequent symbols and longer codes to less frequent symbols. It constructs an optimal binary tree based on the probabilities of the symbols, ensuring that the average code length is minimized [3].

For a set of symbols with known probabilities, the Huffman coding algorithm performs the following steps:

1. Determine the frequencies for each character in the given text.
2. Sort the symbols in descending order based on their frequencies.
3. Merge the two symbols with the lowest frequencies by creating a new branch with a distinct binary digit and summing their frequencies.
4. The binary digits from the previous node, where merging occurred, are appended to the original symbols, forming the codeword for each symbol.
5. Repeat the merging process until no more symbols remain to be merged.
6. By following these steps, Huffman coding achieves efficient compression by assigning shorter codes to more frequent characters and longer codes to less frequent characters.

3. Result and Analysis

3.1 Source Entropy

A random sequence of symbols a,b,c and d generated with probabilities 0.5, 0.3, 0.1 and 0.1 respectively, then the the source entropy calculated according to these probabilities as the entropy equation:$H(S) = -\sum p_i \cdot log_{2(}p_{i)}$

This equation implemented in the code attached in the appendix, and the source entropy was 1.685.

```
"D:\BZU\Machine Learning\pythonProject\venv\Scripts\python.exe" "D:\BZU\Machine Learning\pythonProject\main.py"
Entropy:
 1.6854752972273346

Process finished with exit code 0
```

Figure 1: Source Entropy

3.2 Parse the symbol sequence of N=30 and finding the Nb and compression ratio.

The attached code in the appendix, generate a random sequence for N=30 and the dictionary review the codeword for each new phrase, then it calculate the number of phrases which is 14 for this random sequence, the bits per phrase which is relates to the 8bits of the ASSCI and it was 12 bits/phrase. However, the Nb calculated by multiplying the number of phrases by the bits per phrase which is 168 and the compression ratio was 0.7.

```
N = 30
------
Generated sequence: abababbaabbbcbabaacacaabbbaabc
Encoded sequence: 0000a0000b0001b0011b0001a0010b0010c0010a1000a0000c0001c0101b0110a0011c
-------------------------------------------------------------------------------------------
Dictionary Location     Dictionary Contents Codeword
       1 001                          a           <0,a>
       2 010                          b           <0,b>
       3 011                         ab           <1,b>
       4 100                        abb           <3,b>
       5 101                         aa           <1,a>
       6 110                         bb           <2,b>
       7 111                         bc           <2,c>
      8 1000                         ba           <2,a>
      9 1001                        baa           <8,a>
     10 1010                          c           <0,c>
     11 1011                         ac           <1,c>
     12 1100                        aab           <5,b>
     13 1101                        bba           <6,a>
     14 1110                        abc           <3,c>
-------------------------------------------------------------------------------
# of phrases: 14
# of bits per phrase: 12 -> [4 + 8]
Nb: 168 -> [12 * 14]
Nb/N*8: 0.7
```

Figure 2 : Symbol sequence of N=30

Table 1 : Symbol sequence of N=30

Sequence Length N	Size of encoded sequence (N_B)	Compression ration $N_B/(8*N)$	Number of bits per symbol (N_B/N)
30	168	0.7	5.6

3.3 Generating 5 sequences for each N, and finding the average Nb

The code in the appendix, was applied for 5 times and each the average value of N_B calculated and stored in the table.

Figure 3: Output of the first repetition of the generated sequence for all N

And the values for each run was as shown:

N	Nb
20	132
50	286
100	546
200	990
400	1755
800	3232
1000	4032
2000	7582

N	Nb
20	132
50	273
100	490
200	990
400	1710
800	3280
1000	3984
2000	7565

N	Nb
20	120
50	299
100	518
200	975
400	1740
800	3280
1000	3968
2000	7599

N	Nb
20	132
50	273
100	518
200	1005
400	1710
800	3264
1000	3920
2000	7599

N	Nb
20	132
50	273
100	546
200	1005
400	1710
800	3280
1000	3984
2000	7531

Figure 4: the values of Nb for all N in 5 times

Table 2 : Symbol sequence of all N

Sequence length N	Size of encoded sequence (N_{av})	Compression ratio N_{av} / (8*N)	Number of bits per symbol (N_{av}/N)
20	130	0.8125	6.5
50	281	0.7025	5.62
100	524	0.655	5.24
200	993	0.620625	4.965
400	1725	0.539063	4.3125
800	3267	0.510469	4.08375
1000	3978	0.49725	3.978
2000	7575	0.473438	3.7875

As we saw, its clear that the compression ratio decreases by increasing the sequence length.

3.4 Huffman coding

The code in the appendix took the probabilities of characters a,b,c and d and assign a codewrodsto them as shown.

```
"D:\BZU\Machine Learning\pythonProject\venv\Script
Huffman Codewords:
a: 0
c: 100
d: 101
b: 11
```

Figure 5: Codewords of characters by Huffman

Then, we find the average number of bits per character by calculating the weighted average of the codeword using the probability of each character.

Average bits per character = (0.5*1) + (0.3 * 2) + (0.1 *3) + (0.1 * 3) = 1.7

```
Average bits per character (Huffman): 1.7
```

Figure 6: Average bits per character in Huffman by python code

Then, we encode 100 randomly generated symbols such that each char requires 8 bits for encoding, by multiplying the number of symbols by the number if bits/symbol it will give us 800.

```
['c', 'a', 'b', 'a', 'd', 'a', 'b', 'c', 'b', 'a', 'a', 'a', 'c', 'd', 'a', 'b', 'a', 'a', 'a', 'a', 'a', 'b', 'a', 'd', 'b', 'a', 'c', 'b', 'a', 'b', 'a', 'a', 'a', 'a', 'b', 'c
Number of bits needed using ASCII code: 800
```

Figure 7: # Of bits required to encode 100 symbol

```
Size of encoded sequence (NB): 185 bits ( 24 bytes)
Compression ratio (NB/(8*N)): 0.23125
Number of bits per symbol (NB/N): 1.85
```

Figure 8: Huffman Results

Table 3 : Limpel Ziv VS Huffman

Sequence Length N	Size of encoded sequence NB	Compression Ratio Nb/8N	Number of bits per symbol
Limpel Ziv (100)	524	65.5%	5.24
Huffman (100)	185 bits	23.12%	1.85

For a sequence length of 100, the size of the encoded sequence (NB) is 185 bits for Huffman coding and 524 bits for Lempel-Ziv coding, indicating that Huffman coding achieves a smaller encoded sequence size. The compression ratio (NB/(8*N)) is 0.2312 (or 23.12%) for Huffman coding, meaning that the compressed data is approximately 23.12% of the original size, whereas the compression ratio for Lempel-Ziv coding is 0.655 (or 65.5%), indicating a different level of compression efficiency. The number of bits per symbol (NB/N) is 1.85 for Huffman coding and 5.24 for Lempel-Ziv coding, suggesting that Huffman coding requires fewer bits per symbol and thus results in more efficient encoding compared to Lempel-Ziv coding.

4. Conclusion

This project explored and implemented Lempel-Ziv (LZ) and Huffman coding techniques using Python to compress random symbol sequences. LZ encoding effectively reduced data size by dynamically building a dictionary and encoding sequences with repeating patterns. Meanwhile, Huffman coding achieved a slightly higher compression ratio compared to LZ for a 100-symbol sequence, leveraging known symbol probabilities efficiently.

The results, presented in tables, highlighted the practical advantages of both methods in digital communication and storage. This project demonstrated the synergy of theoretical knowledge and practical implementation in tackling real-world data compression challenges.

5. References

[1] 15 6 2024. [Online]. Available: https://en.wikipedia.org/wiki/Shannon%27s_source_coding_theorem.

[2] 15 6 2024. [Online]. Available: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/.

[3] 15 6 2024. [Online]. Available: https://www.programiz.com/dsa/huffman-coding.

6. Appendix

6.1 Entropy Code:

```python
import math


def calculate_entropy(probability):
    return -sum([prob * math.log2(prob) for prob in probability])


probabilities = [0.5, 0.3, 0.1, 0.1]
entropy = calculate_entropy(probabilities)
entropy_str = "Entropy: \n " + str(entropy)
print(entropy_str)
```

6.2 Parse the symbol sequence of N=30 and finding the Nb and compression ratio.

```python
from LimpelZiv import *

symbols = ['a', 'b', 'c', 'd']
probabilities = [0.4, 0.3, 0.2, 0.1]
summary_output = "*********************************************\n\tN\t  Nb\t\tNb/8*N\t\t\tNb/N\n"
Ns = [20, 50, 100, 200, 400, 800, 1000, 2000]
for N in Ns:
    lz_dict = {}
    generated_sequence = generate_sequence(symbols, probabilities, N)
    parse_sequence(generated_sequence, lz_dict)
    head_bits = math.ceil(math.log2(len(lz_dict.items())))
    codewords = encode_phrases(head_bits, lz_dict)
    encoded_sequence = ''.join([str(item) for item in codewords.values()])
    Nb = calculate_nb(head_bits, 8, len(lz_dict.items()))
    print("N = " + str(N) + "\n" + '-' * len("N = " + str(N)))
    print(
        "Generated sequence: " + generated_sequence + '\n' + "Encoded sequence: " + encoded_sequence + '\n' + '-' * len(
            "Encoded sequence: " + encoded_sequence))
    print("# of phrases: " + str(len(lz_dict.items())))
    print("# of bits per phrase: " + str(head_bits + 8) + ' -> [' + str(head_bits) + ' + 8]')
    print("Nb: " + str(Nb) + ' -> [' + str(head_bits + 8) + ' * ' + str(len(lz_dict.items())) + ']')
    print("-"*len("Nb: " + str(Nb) + ' -> [' + str(head_bits + 8) + ' * ' + str(len(lz_dict.items())) + ']') + '\n')
    summary_output += str(N).rjust(5) + "\t" + str(Nb).rjust(5) + "\t\t" + str(
        "{:.2f}".format(round((Nb / (N * 8)) * 100, 2))) + "%\t\t\t" + str("{:.2f}".format(Nb / N)) + "\n"
print(summary_output)
```

6.3 Huffman Coding. Code #1

```
import heapq
import random

class Node:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None
```

```
    def __lt__(self, other):
        return self.frequency < other.frequency

def build_huffman_tree(symbols, probabilities):
    # Step 2: Build the Huffman tree
    heap = []
    for symbol, probability in zip(symbols, probabilities):
        node = Node(symbol, probability)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged_frequency = node1.frequency + node2.frequency
        merged_node = Node(None, merged_frequency)
        merged_node.left = node1
        merged_node.right = node2
        heapq.heappush(heap, merged_node)

    return heap[0]

def build_codewords(node, current_code, codewords):
    if node.symbol:
        codewords[node.symbol] = current_code
        return

    build_codewords(node.left, current_code + "0", codewords)
    build_codewords(node.right, current_code + "1", codewords)

# Step 1: Define the symbols and their respective probabilities
symbols = ['a', 'b', 'c', 'd']
probabilities = [0.5, 0.3, 0.1, 0.1]

# Step 2: Build the Huffman tree
huffman_tree = build_huffman_tree(symbols, probabilities)

# Step 3: Build the codewords dictionary
codewords = {}
build_codewords(huffman_tree, "", codewords)

# Step 4: Print the codewords
print("Huffman Codewords:")
for symbol in codewords:
    print(symbol + ": " + codewords[symbol])

# Step 5: Calculate average bits per character
average_bits_per_character = sum(probabilities[symbols.index(symbol)] * len(codewords[symbol]) for symbol in
codewords)

print("Average bits per character (Huffman):", average_bits_per_character)
```

```
# Step 6: Generate 100 random symbols
random_symbols = random.choices(symbols, probabilities, k=100)
print(random_symbols)
# Step 7: Calculate the number of bits needed using ASCII code
ascii_bits = 8 * len(random_symbols)

print("Number of bits needed using ASCII code:", ascii_bits)

# Step 8: Encode the sequence using Huffman coding
encoded_sequence = ''.join(codewords[symbol] for symbol in random_symbols)

# Step 9: Calculate the size of the encoded sequence (NB)
encoded_bits = len(encoded_sequence)
encoded_bytes = (encoded_bits + 7) // 8  #The addition of 7 in the expression (encoded_bits + 7) // 8 is to ensure
that any remaining bits, after dividing by 8, are properly accounted for.

print("Size of encoded sequence (NB):", encoded_bits, "bits (", encoded_bytes, "bytes)")

# Step 10: Calculate the compression ratio NB/(8*N)
compression_ratio = encoded_bits / (8 * len(random_symbols))

print("Compression ratio (NB/(8*N)):", compression_ratio)

# Step 11: Calculate the number of bits per symbol (NB/N)
bits_per_symbol = encoded_bits / len(random_symbols)

print("Number of bits per symbol (NB/N):", bits_per_symbol)
```

6.4 Huffman coding. Code #2:

```
import heapq
import random
from LimpelZiv import *


class Node:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.frequency < other.frequency

def build_huffman_tree(symbols, probabilities):
    # Step 2: Build the Huffman tree
    heap = []
    for symbol, probability in zip(symbols, probabilities):
        node = Node(symbol, probability)
        heapq.heappush(heap, node)

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged_frequency = node1.frequency + node2.frequency
```

```
        merged_node = Node(None, merged_frequency)
        merged_node.left = node1
        merged_node.right = node2
        heapq.heappush(heap, merged_node)

    return heap[0]

def build_codewords(node, current_code, codewords):
    if node.symbol:
        codewords[node.symbol] = current_code
        return

    build_codewords(node.left, current_code + "0", codewords)
    build_codewords(node.right, current_code + "1", codewords)

print("Huffman part" + "\n" + "-"*len("Huffman part") + "\n")

# Step 1: Define the symbols and their respective probabilities
symbols = ['a', 'b', 'c', 'd']
probabilities = [0.5, 0.3, 0.1, 0.1]

# Step 2: Build the Huffman tree
huffman_tree = build_huffman_tree(symbols, probabilities)

# Step 3: Build the codewords dictionary
codewords = {}
build_codewords(huffman_tree, "", codewords)

# Step 4: Print the codewords
print("Huffman Codewords:")
for symbol in codewords:
    print(symbol + ": " + codewords[symbol])

# Step 5: Calculate average bits per character
average_bits_per_character = sum(probabilities[symbols.index(symbol)] * len(codewords[symbol]) for symbol in codewords)

print("Average bits per character (Huffman):", average_bits_per_character)

# Step 6: Generate 100 random symbols
random_symbols = random.choices(symbols, probabilities, k=100)
print(random_symbols)
# Step 7: Calculate the number of bits needed using ASCII code
ascii_bits = 8 * len(random_symbols)

print("Number of bits needed using ASCII code:", ascii_bits)

# Step 8: Encode the sequence using Huffman coding
encoded_sequence = ''.join(codewords[symbol] for symbol in random_symbols)

# Step 9: Calculate the size of the encoded sequence (NB)
encoded_bits = len(encoded_sequence)
encoded_bytes = (encoded_bits + 7) // 8  #The addition of 7 in the expression (encoded_bits + 7) // 8 is to ensure that any
remaining bits, after dividing by 8, are properly accounted for.

print("Size of encoded sequence (NB):", encoded_bits, "bits (", encoded_bytes, "bytes)")

# Step 10: Calculate the compression ratio NB/(8*N)
compression_ratio = encoded_bits / (8 * len(random_symbols))
```

```
print("Compression ratio (NB/(8*N)):", compression_ratio)

# Step 11: Calculate the number of bits per symbol (NB/N)
bits_per_symbol = encoded_bits / len(random_symbols)
print("Number of bits per symbol (NB/N):", bits_per_symbol)
print("\nLimpel-ziv part" + "\n" + "-"*len("Limpel-ziv part") + "\n")
N = 100
lz_dict = {}
generated_sequence = random_symbols
parse_sequence(generated_sequence, lz_dict)
head_bits = math.ceil(math.log2(len(lz_dict.items())))
codewords = encode_phrases(head_bits, lz_dict)
encoded_sequence = ''.join([str(item) for item in codewords.values()])
Nb = calculate_nb(head_bits, 8, len(lz_dict.items()))
print("Size of encoded sequence (NB):", Nb, "bits")
print("Compression ratio (NB/(8*N)):", str("{:.2f}".format(round((Nb / (N * 8)) * 100, 2))))
print("Number of bits per symbol (NB/N):", str("{:.2f}".format(Nb / N)))
```