University of Bergen

Information theory
INF242

First mandatory exercise

Student :
Arthur Gautier

September 25, 2023

Contents

1 Lempel-Ziv-Welch

1.1 Compression

1.1.1 Build the dictionary

We start by building a dictionary using the code of the ASCII table. We include the higher and lower case letters of the latin alphabet. We also add the higher and lower letters exclusive to the Norwegian language.

```
file = open("folktale.txt", "r", encoding="utf-8")
data = file.read()

# Preparing the dictionary :
minuscule = [chr(i+97) for i in range(0,26)]
majuscule = [chr(i+65) for i in range(0,26)]
norwegian_letters_and_space = [chr(198), chr(216), chr(197), chr(230),
chr(248), chr(229), chr(32)]
characters = minuscule + majuscule + norwegian_letters_and_space
dictionary_size = len(characters)
dictionary = {characters[i] : i for i in range(0,dictionary_size)}
```

1.1.2 Implementation

We start by initialize variables :

```
str = "" # to build sequences of characters
output = [] # store the compressed output
```

Then we iterate on each symbol of the data. We start by create a *new_symbol* by concatenating the current *str* with the current *symbol*. We check if this *new_symbol* is contained in the dictionary. If it is the case, it means that we can potentially add more characters to the characters to it and continue searching for a longer sequences. That's why we update *str* to *new_symbol*.

If *new_symbol* is not in the dictionary, we start by output the code for *str* to the *output*. We then add the *new_symbol* to the dictionary with a new index (*dictionary_size* that we immediately increase to 1 for the next potential new symbol). Finally, we update *str* with the current *symbol*.

```
for symbol in data:
    new_symbol = str + symbol
    if new_symbol in dictionary:
        str = new_symbol
    else:
        output.append(dictionary[str])
```

```
            dictionary[new_symbol] = dictionary_size
            dictionary_size += 1
            str = symbol
    if str:
        output.append(dictionary[str])

    print(output)
```

1.2 Decompression

We start by initializing two empty strings : *decompressed_ data* (to stock the compressed data) and *str* (to shape the different values of the dictionary).

We start by iterate on each code in the compressed data. If the code is not already in the dictionary, it means that it is a new one : we add this new code to the dictionary (by concatenating *str* with *str[0]*, we extend the dictionary with a new symbol based on the previous sequence).

We then, add the sequence linked to *code* in *decompressed_ data* : we build the original data based on the LZW code.

If the *str* variable is not empty, we add a sequence to the dictionary. The new entry is formed by concatenating *str* with the first character of the sequence linked to *code*.

Then we update *str* with the sequence linked to *code* value : it's going to be used as a prefix for the next *code*.

```
decompressed_data = ""
str = ""
next_code = dictionary_size
compressed_data = output

for code in compressed_data:
    if code not in dictionary:
        print(code)
        print(len(str))
        dictionary[code] = str + str[0]
    decompressed_data += dictionary[code]
    if len(str) != 0:
        dictionary[next_code] = str + dictionary[code][0]
        next_code += 1
    str = dictionary[code]

print(decompressed_data)
```

1.3 Compression rate

The original file is 8200 bytes. The new file after the LZW compression is 5394 bytes. The compression rate is 66%.

2 Huffman encoding

2.1 Node class

The principle of Huffman coding is based on the creation of a tree structure composed of nodes. We need to create a class for the nodes.

```
class Node:
    def __init__(self, value, letter="", NodeLeft=None, NodeRight=None):
        """constructor"""
        self.value = value # to store the attributed value of the node
        self.letter = letter # to store the attributed letter
        self.NodeLeft = NodeLeft # if it's not a leaf, left node child
        self.NodeRight = NodeRight # if it's not a leaf, right node child

    def __lt__(self, Node2):
        """compare two node values"""
        return self.value < Node2.value
```

2.2 Number of occurrences

We now need to count the number of each value in the data we have.

```
def occurrence(data):
    """count the number of occurrences for each present letters in data"""
    # extract the different values:
    values_to_count = []
    occ = []
    for value in data:
        if value not in values_to_count:
            values_to_count.append(value)
    for value in values_to_count:
        number_occurrences = data.count(value)
        letter = [value, number_occurrences]
        occ.append(letter)
    return occ
```

2.3 Huffman coding

Based on this occurrence table, we can now start to build the Huffman encoding. To do that we will use priority queues using the *heapq* library. A priority queue allows three different operation : add a element to the queue, extract an element with the higher key, test if the queue is empty or not. Priority queue answer to all the need of Huffman encoding implementation.

We begin by placing all the various occurrence values previously calculated into a priority queue represented as nodes. Then, as long as there is more than one node remaining

in the queue, we continue to iteratively combine the two highest nodes into a new parent node in each loop, until only a single root node remains in the queue. We return this root.

```
def build_huffman(data):
    """build the huffman tree"""
    occ = occurrence(data)
    priority_file = []
    for element in occ:
        n = Node(element[1], element[0])
        heappush(priority_file, n)
    while len(priority_file) > 1:
        n1 = heappop(priority_file)
        n2 = heappop(priority_file)
        n3 = Node(n1.value + n2.value)
        n3.NodeLeft = n2
        n3.NodeRight = n1
        heappush(priority_file, n3)
    # Root :
    return heappop(priority_file)
```

Now that the tree structure is built, we will traverse the tree and assign values to the leaves. We recursively traverse the tree from the root, assigning the value 0 to the left node and 1 to the right node. Then we create a dictionary to attribute a Huffman code to each of the original code.

```
def make_huffman_code(data):
    """Create the code dictionary"""
    root_node = build_huffman(data)
    dic = {}
    code_huffman(root_node, dic, "")
    return dic


def code_huffman(n, dictionary, code):
    """recursive function to create the huffman code"""
    if is_leaf(n):
        dictionary[code] = n.letter
    else:
        code_huffman(n.NodeLeft, dictionary, code + '0')
        code_huffman(n.NodeRight, dictionary, code + '1')
```

The other functions then allow us to read the original code and assign, for each element of the code, its attribute referenced in the dictionary. This results in a binary code for the original message.

2.4 Compression rate

The original file is 8200 bytes. The new file after the LZW compression + Huffman encoding is 3279 bytes. The compression rate is 40%.