ELEN-0060: Information and Coding Theory

Project 2 - Source coding, data compression, and channel coding

Maxime Goffart
180521

Olivier Joris
182113

Academic year 2021 - 2022

1 Implementation

1.1 Question 1

Our implementation is based on a data structure representing a Node of the tree which has 4 fields:

- the symbol of the Node which is 'None' if the Node is not a leaf.
- the probability of the Node.
- a pointer to the left child of the Node which is 'None' if the Node is a leaf.
- a pointer to the right child of the Node which is 'None' if the Node is a leaf.

We first create a node for each letter in which we store its symbol, its probability, and 'None' for both children. Then, we build the tree as seen in class: we iteratively link the two nodes with the lowest probabilities until the tree is complete with a node having 1 as probability at root. Each new node born from the birth of two nodes has for probability the sum of the probabilities of these nodes. It also maintains a link to these two nodes becoming its children, and has for symbol 'None'. Finally, we recursively build the Huffman code on the basis of the previously built tree and a choice to represent a left child by a 0 and a right child by a 1.

The output of our code for this question is: `{'A': '000', 'B': '001', 'E': '01', 'C': '100', 'D': '101', 'F': '11'}` which is not the same as the one found during the exercise session. It is not a problem because the Huffman code is not unique. Our code is acceptable and its tree can be observed in the figure 1.

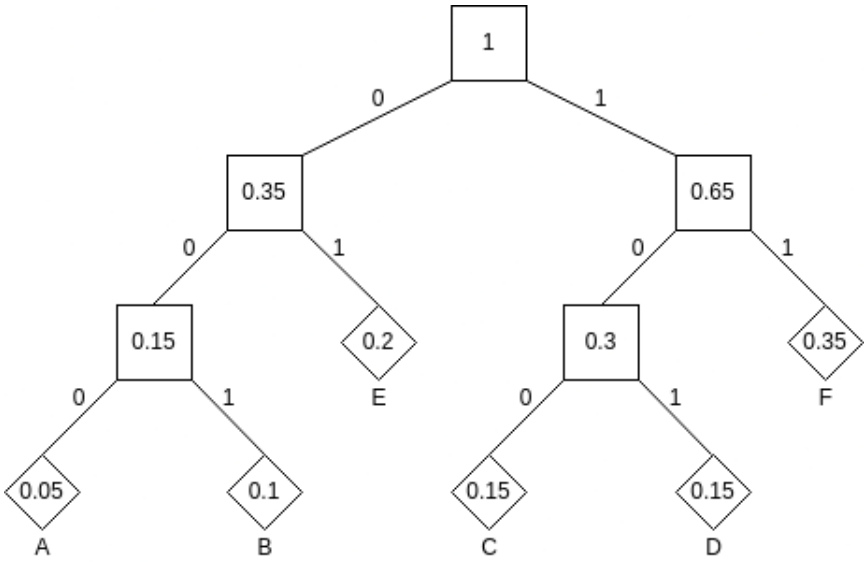

Figure 1: Huffman tree corresponding to the output code

To generate a Huffman code of any (output) alphabet size, several changes would be necessary both in our data structure and in our algorithm:

- Data structure
  - We would need to add a field to store the alphabet size N which was implicitly set to 2 in the binary Huffman code.

  - We would need to store the N children of each node which is not a leaf instead of the two children in the binary Huffman code.

- Algorithm
  - We would need to take the N lowest probability and link them to create new nodes instead of the two nodes with the lowest probability in the binary Huffman code.
  - We would need to assign a given unique symbol of the code to each child of a node when recursively building the Huffman code instead of 1 and 0 in the binary Huffman code where there were only 2 children per node.
  - We might have to deal with problems of number of symbols and size of the alphabet not compatible unlike in the binary Huffman code where it is always possible to merge the two lowest probable nodes.

1.2 Question 2

The example given in the course about state of the art in data compression on slide 50/53 where the sequence 1011010100010 is encoded gives this output for our `LZ_online` function:

- Dictionary: {'': (0, ''), '1': (1, '1'), '0': (2, '00'), '11': (3, '011'), '01': (4, '101'), '010': (5, '1000'), '00': (6, '0100'), '10': (7, '0010')}
- Encoded sequence: 100011101100001000010

1.3 Question 3

The basic version has one main problem which is address coding. It needs to know the size of the dictionary before an address encoding. The on-line version solves this problem by using the current dictionary size to determine the number of bits which is equal to $\lceil \log_2 N \rceil$, where N is the dictionary size. It is thus decreasing the size of the encoded text. This on-line variant is most of the time not very competitive in terms of optimality but is very robust because it does not need assumptions about source behaviour and can thus allow an instantaneous coding without having to browse the dictionary multiple times. The asymptotic performances are reached only when the dictionary starts to become representative: when it contains a significant fraction of sufficiently long typical messages.

1.4 Question 4

Running the example given in Figure 2 with our function and taking a `window_size` of size 7 gives us this output: [(0, 0, 'a'), (0, 0, 'b'), (0, 0, 'r'), (3, 1, 'c'), (2, 1, 'd'), (7, 4, 'd')]. We have chosen to set the size of the look ahead buffer to the size of the window because it allows to have longer matches prefix and the prefix cannot be longer than the this size.

2 Channel coding

2.1 Question 16

In order to implement a function to read and display the given image, we used the methods `imread` and `imshow` provided by OpenCV.

2.2 Question 17

To encode the image signal, we used a fixed-length binary code of 8 bits. We have chosen 8 bits because there are 256 (from 0 to 255) possible values, so we need $\lceil log_2(256) \rceil = 8$.
The code is the binary representation of the grayscale value of each pixel.

2.3 Question 18

By simulating the channel effect on the binary signal of the image, we get the following image:

Figure 2: Image after simulating the channel effect

As we can see in the picture, after simulating the channel effect, there are a lot of small dots[1] that are pixels with different grayscale values compared to their very close neighbors.
This is due to the fact that we are simulating a potential loss bit by bit and we are not using any sort of redundancy. Thus, if one of the most significant bits is modified, it completely changes the grayscale value for the pixel.

2.4 Question 19

In order to compute the Hamming(7,4) code for the binary image signal, we need to add 3 redundancy bits for every 4 bits. The 3 redundancy bits are:

- Bit 1 = $(bit0 + bit1 + bit2)\ mod\ 2$
- Bit 2 = $(bit1 + bit2 + bit3)\ mod\ 2$
- Bit 3 = $(bit0 + bit2 + bit3)\ mod\ 2$

where bit 0, bit 1, bit 2, and bit 3 are, respectively, the first, second, third, and fourth bits for which we want to add redundancy.
By applying this principle on each block of 4 bits from the binary image signal, we get the Hamming(7,4) code for the entire binary image signal.

[1] Zoom in the image to see them better.

2.5 Question 20

If we decode the binary image signal with redundancy, we get the following image:

Figure 3: Image after simulating the channel effect with redundancy

As we can see in the picture, we observe less dots compared to *question 18*. This is because we used redundancy with the Hamming(7,4) code. Yet, we can still observe some dots. This is because Hamming(7,4) code can correct only one error per chunk of bits. In some cases, we might have more than one error per chunk thus the original source is not recoverable.

In order to decode the binary signal with redundancy, we used the syndrome decoding technique, as in the exercise sessions.
To apply this technique, we consider that the source bits were correctly transmitted through the channel. Then, we recompute the 3 parity bits based on the received 4 source bits. Afterwards, we apply a bitwise and between the received parity bits and the re-computed parity bits.
Based on the result of the bitwise operation, multiple cases are possible:

- If 2 or 3 bits are 1 in the result of the bitwise operation, we can deduce that one of the source bit has been incorrectly transmitted. To recover from the error, we can proceed the following way:
  - If bits 0 and 1 of the bitwise and are 1, we need to flip the second source bit.
  - If bits 1 and 2 of the bitwise and are 1, we need to flip the fourth source bit.
  - If bits 0 and 2 of the bitwise and are 1, we need to flip the first source bit.
  - If all the bits of the bitwise and are 1, we need to flip the third source bit.
- If 1 bit is 1 in the result of the bitwise operation, we can deduce that one of the parity bit has been incorrectly transmitted.
- It is possible that 2 or more bits were transmitted incorrectly. In that case, we cannot recover from the error.

2.6 Question 21

In order to reduce the loss of information, we could use Reed-Solomon codes instead of Hamming(7,4) because they allow to correct more than 1 errors in some situations.
Another solution is to replace the type of channel. For instance, replacing copper wire with fiber optic in computer networks. But, it is not also feasible.
There is always a trade-off between the rate and the capability of correcting from errors. Thus, in order to increase the rate, we would need to decrease the redundancy introduced for parity.

3 Source coding and reversible data compression

3.1 Question 5

First, we can compute the marginal probability distribution of all the symbols based on the given Morse text. The distribution is:

Symbol	.	-	_	/
Probability	0.43378	0.28706	0.21452	0.06464

Table 1: Marginal probability distribution of all the symbols

Based on the distribution of probabilities, we can compute the binary Huffman code. We get the following code:

Symbol	.	-	_	/
Huffman code	0	11	101	100

Table 2: Binary Huffman code

By applying the obtained Huffman code to the Morse text, we get the encoded Morse text whose size is 2213141 bits. The original Morse text has a size of 2398580 bits. Thus, we have a compression rate of 1.08379.

3.2 Question 6

We can compute the expected average length of the Huffman code by computing the sum for the 4 symbols of the probability of each symbol times the length of the code associated with the symbol. We get that the expected average length is equal to 1.84538.
By comparing to the empirical average length of the Huffman code, we get:

$$\text{Length of encoded / length of initial text} = 1.84538$$

This value is the same as the one for the expected average length which is logical since the expected average length was computed based on the probabilities of occurrence of each symbol based on the given Morse text.
By comparing to the theoretical bounds, we get that:

$$\frac{H(S)}{log_2(q)} \leq \overline{n} \leq \frac{H(S)}{log_2(q)} + 1 \text{ because } 1.77138 \leq 1.84538 \leq 2.77138 \quad (1)$$

Thus, our code is optimal, because the inequation is satisfied, which was expected because Huffman codes are optimal. But, the obtained code is not absolutely optimal because $\frac{H(S)}{log_2(q)} \neq \overline{n}$.

3.3 Question 7

For the evolution of the empirical average length of the Huffman code with respect to the lengths of the input texts, we get the following plot:

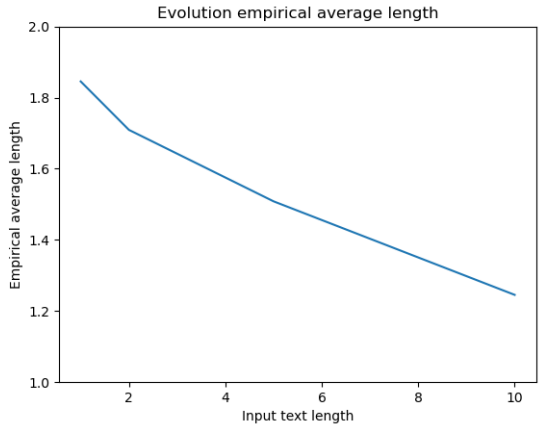

Figure 4: Evolution of empirical average length

As we can observe on the plot, the empirical average length decreases as the length of the input text increases.

3.4 Question 8

By using the Lempel-Ziev algorithm to encode the Morse code, we get an encoded code with a total length of 1866281 bits. The original Morse code has a size of 2398580 bits. Thus, we get a compression rate of $1.28522 = \frac{2398580}{1866281}$.

3.5 Question 14

By computing the Huffman code on the original (27 symbols) text, we get the following code:

Symbol	**a**	**b**	**c**	**d**	**e**	**f**	**g**
Code	1010	100000	100010	11010	001	100011	00000
Symbol	**h**	**i**	**j**	**k**	**l**	**m**	**n**
Code	0111	0100	1100111001	1100110	11000	110010	0110
Symbol	**o**	**p**	**q**	**r**	**s**	**t**	**u**
Code	1001	100001	1100111011	0101	0001	1011	00001
Symbol	**v**	**w**	**x**	**y**	**z**		
Code	11001111	110110	1100111010	110111	1100111000	111	

Table 3: Binary Huffman code for original text

The expected average length of the code is 4.15047 and the experimental length of the encoded text is 1711279 bits.
The original text has a size of 2061550 bits and the encoded text has a size of 1711279 bits. Thus, the compression rate is 1.2046 (= 2061550/1711279).

3.6 Question 15

By comparing the values obtained at the questions 5 and 14, we can conclude that it is better to encode directly the text without first converting it to Morse because the size of the encoded Morse text is 2213141 bits while the size of the encoded text is 1711279 bits. Furthermore, the compression rate obtained is higher in question 14 compared to question 5.
This can be justified theoretically by the fact that using Huffman encoding with 27 symbols instead of 4 symbols allows to benefit more from the frequencies of letters in English. For instance, the letter z in English is way less frequent than the letter e. While using the Morse encoding, the 2 symbols . and - both appear for some of the letters. Thus, we loose the benefit provided by less frequent letters that will get longer codes and more frequent letters will get shorter codes.