ICT Project 2

HOUYON Joachim
SEBATI Ilias

March 2022

1 Q1

Our binary huffman implemenation does the following:

1. Take the two symbols/nodes with the lower probability.
2. Create a node with as sum, the sum of the probabilities of the two symbols/nodes used to create it.
3. Repeat until only one node remains.

The remaining node is the root of the binary tree. You can go over its children and display the code based on the branches you chose.
Reporting of the Exercisce 7-TP2.:

Probability	.05	.10	.15	.15	.2	.35
Huffman code	000	001	100	101	01	11

To make it work with any size alphabet (i.e. size n), we just have to create a node from the n lower probabilities, and finish as above.
Once the root computed, you go over the children of the root, you give the first letter to the first branch (say a), the second letter to the second branch (say b), up to the last branch (the n^{th} one) which will have the last character of the alphabet. You recursively do that for their children.

2 Q2

Sequence is 1011010100010.
Encoded sequence is 100011101100001000010
Dictionnary is:

Sumbol		1	0	11	01	010	00	10
Index	0	1	2	3	4	5	6	7
Adress + Bit		1	00	011	101	1000	0100	0010

3 Q3

The main difference is that the *basic* has to know the final size of the dictionnary to be able to choose the right number of bits for the addresses. Just when this is done, he can encode them. The *online* fashion uses as address the log of the size of the current dictionnary. Hence, he does not have to wait to see the final size of the dictionnary; he can encode on the flight (i.e. online).

Note also that the *online* fashion is more flexible as we can send other symbols even after that the sender believe that there is nothing more to send, and this without modifying the dictionnary of the past elements. While the *basic* would need to recompute the final size and change all the previous symbols.

4 Q4

The *LZ77* algorithm has been applied on the sequence *abracadabra*. By respecting the output format, our algorithm outputs the following encoded sequence as a string: *00a00b00r31c21d74d*

5 Q5

The probability of a symbol is the frequency of that symbol in the text. The table below represents the probability of each symbol and its associated huffman binary code.

Symbol	.	_	-	/
Probability	0.434	0.215	0.287	0.065
Huffman Binary Code	0	101	11	100

The total length of the encoded sequence is the number of bits that is used to encode the morse sequence using the generated huffman binary code. The total length of the encoded sequence is 2 213 141 bits.

The compression rate, expressed in bits, between two sequence s_1 and s_2 is computed as

$$CompressionRate(s_1, s_2) = \frac{log_2(card(s_1))}{log_2(card(s_2))} \frac{|s_1|}{|s_2|}$$

where $card(s)$ denotes the cardinality of a sequence s, i.e. the number of different symbols in s, and $|s|$ is the length of s. Note that, in the next questions, whenever we talk about the compression rate, they will refer to this formula.

In our case, the cardinality of the morse sequence if 4, and 2 for the encoded sequence using Huffman. The compression rate is equal to 1.08, meaning that the length of the encoded sequence is 8% lower than the length of the morse sequence in bits.

6 Q6

The expected average length of the Huffman code is the sum of product between the probability and the length of the binary code of a symbol s, for each symbol in the encoded sequence. The expected average length of our Huffman code is 603 921.505 bits.

The empirical average length is the ratio between the length of the morse sequence and the length of the encoded sequence. This ratio is equal to 1.845.

We can compare this value with the theoretical bound. Huffman produces an optimal code tree and prefix-tree code such that

$$\frac{H(s)}{log_2 q} \leq \bar{n} < \frac{H(s)}{log_2 q} + 1$$

where $\bar{n}$ is the empirical average length, and q the cardinality of the encoded sequence. In our case, $q = 2$.

The theorecical bound says that our empirical average length is in $[1.771, 2.771]$, meaning that our code is not absolutly optimal, tt would have been optimal iff $\bar{n} = 1.771$.

7 Q7

Below is presented the evolution of the empirical average length of the encoded morse for increasing input text lengths.

It is worth noticing that the empirical average length quickly converges to the empirical average length of the whole morse sequence.

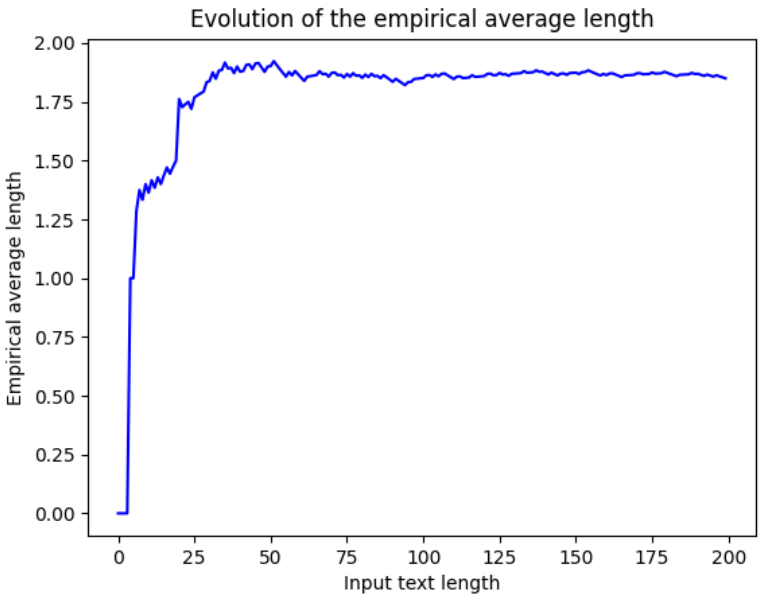

Figure 1: Evolution of the empirical average length of the encoded morse for increasing input text lengths

8 Q8

The encoded morse code using the on-line Lempel-Ziv alrogithm provides an encoded sequence of 1 622 712 bits. The compression rate is equal to 1.478, meaning that the encoded sequence if 47% lower than the morse sequence.

The computation of the number of bits of the encoded sequence is done as such:

1. Count each 0 and 1 symbols from the output of the algorithm.
2. Count the remaining symbols and multiply this quantity by 2. because each symbol of the morse code can be represented with exactly 2 bits.
3. Sum up those two quantitities, it gives the total length of the encoded sequence in bits.

9 Q9

The LZ77 algorithm compresses, with a sliding window size of 7, the morse sequence into an encoded sequence of 3 711 926 bits, giving a compression rate of 0.646. The size of the encoded sequence is higher than the original sequence.

The number of bits n of the encoded sequence is computed using this formula:

$$n = T(2log_2(w+1) + log_2(card(s)))$$

where T is the number of tuples generated by the LZ77 algorithm, w is the window size and s is the original sequence. Notice that if the last tuple does not contain a symbol c because this is the end of the sequence, the number of bits n of the encoded sequence is equal to

$$n = 2Tlog_2(w+1) + (T-1)log_2(card(s)).$$

10 Q10

The first possible combination is to first perform the LZ77 algorithm on the morse sequence, giving a list of tuples. We can consider each tuple as a symbol and apply the Huffman algorithm on these tuples. This combination is decent if the LZ77 algorithm produces many tuples that are equals.

The second possible combination is also to start by performing the LZ77 algorithm on the morse sequence. From this, we know that d and p have the same cardinality, and c has the cardinality of the original sequence. Therefore, we can build 2 huffman codes: one for the symbols of d and p, and one for the symbols of c. We can split a sequence

$d_1p_1c_1d_2p_2c_2...$ into two sequences $d_1p_1d_2p_2...$ and $c_1c_2...$

The encoding is straightforward, the first two symbols are encoded using the first Huffman code (d and p), the third one is encoded with the second Huffman code (c). The next 3 symbols will follow the same pattern. This approach is interessing in the fact that if we were only using a single Huffman code on the whole LZ77 output, it would be less efficient as some symbols will use more bits than in the setting with two Huffman codes. We take profit on the organisation of the symbols in order to optimize the compression of the sequence.

11 Q11

For this, the first idea proposed at *Q10* has been implemented. Using LZ77 with a sliding window of size 7, the combination of LZ77 and Huffman algorithm provides an encoded sequence of 2 769 924 bits, representing a compression rate of 0.866. The encoded sequence is longer than the original sequence.

12 Q12

The figures below presents the total lenghts and compression rates using LZ77 and the implemented combination of LZ77 and Huffman. We can notice that the combination clearly outperforms LZ77 on the morse sequence. Furthermore, the size of the sliding window has a strong impact on the compression rate and the total length. However, higher sliding windows come with a cost in terms of computational time.

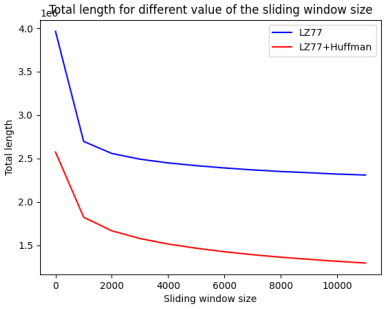

Figure 2: Total length for different values of the sliding window size

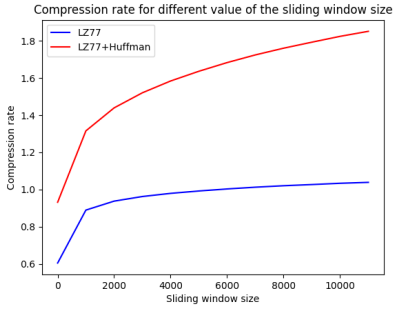

Figure 3: Compression rate for different values of the sliding window size

Compared with the on-line Lempel-Ziv algorithm, the combination performs better as it provides an encoded sequence that is lower, meaning that the compression rate is higher. However, if the compression time is a factor to be taken into consideration, the on-line Lempel-Ziv algorithm could be a fair trade-off.

13 Q13

The LZ77 algorithm can be weak against repetitions that occur at long distances, because the algorithm is dependent of the window size; if the window size is too small, it might not capture these repetitions. On the other hand, if the window size is too big, the computational time will be too high.

The on-line Lempel-Ziv algorithm keeps track of all the patterns it has seen since the beginning of the encoding. Therefore, it can captures any repetitions as it is not limited by the horizon of a buffer. Furthermore, the computational time is acceptable.

Depending the computational capacity and how distant these repetitions are from eachother, The combination of LZ77 and Huffman algorithm or on-line Lempel-Ziv algorithm could be used.

14 Q14

The table below is the huffman binary code for each symbol. The average expected length is equal to 1 112 500 bits, the total length is equal to 1 711 279 bits and the compression rate is 1.146.

15 Q15

In overall, we can agree on the fact that it is more efficient to directly encode the original text with Huffman than encode it first with Morse code before the

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

Huffman encoding. Indeed, applying Huffman on the original text lowers the number of bits by 14%, where applying Huffman on the morse text lowers the number of bits by 8%.

16 Q17

Our binary code has to be uniquely decodable. Using a fixed-length code, the minimum number of bits to use is $\lceil log_2 256 \rceil$ bits. Hence 8 bits.

17 Q18

The image is corrupted. It contains a lot of wrong pixels. The channel flipped a lot of bits. And those bits have not been recovered.

Figure 4: Fixed-length encoding through .01 binary channel

18 Q20

The image is clearer. It has less wrong pixels. It is essentially due to the fact that errors have been recovered. As long as only one bit at worse is corrupted, the hamming was able to recover the initial sequence.

Note that when increasing a lot the probability of flipping a bit in the channel (i.e. p), we see a lower difference (or even none) between the image generated using a fixed-length binary code and the one using the hamming(7, 4). It makes sense as the hamming recovery is assuming that there is no more than one bit per sequence of 7 bits that was flipped.

Figure 5: Hamming(7, 4) encoding through .01 binary channel

19 Q21

Reducing the loss of information would mean adding redundancy. Hence, reducing the cormmunication rate. There is a tradeoff here.

For instance, to reduce the loss of information we could simply send 3 times the same bit (i.e. hamming(3, 1)); this would result in selecting the bit that is present twice. In term of loss of information, this is better than hamming(7, 4) as we must have less than 2 errors on 3 bits to recover (probability = $(1-p)^3 + (1-p)^2 * p = 0.98$), while hamming(7, 4) needed at most 1 error on 7 bits (probability = $(1-p)^7 + (1-p)^6 * p = 0,94$), which is a way stronger assumption (i.e. lower probability). However, it reduces the communication rate as hamming(7, 4) had a $\frac{4}{7}$ communication rate, and hamming(3, 1) had a $\frac{1}{3}$ communication rate.

Redundancy needs to be reduced in order to increase communication rate. We could send the data without any redundancy; a $\frac{1}{1}$ communication rate but a probability p of loss of information. Several other hamming codes exist to increase the communication rate (e.g. Hamming(15, 11), Hamming(31,26), etc)