Information and coding theory

Project 2

Corentin Merle s162662
Jad Akkawi s216641

Data Science and Engineering, bloc 1
Liège Université
School of Engineering
Academic year 2021-2022

1 Implementation

Question 1

The implementation of a Binary Huffman code is as follows :

— Create a node class with 2 children (left and right).
— Create a list of characters with probabilities sorted in reverse.
— Make a node with the last 2 elements of the list as children.
— Replace these 2 elements with this node (value equal to sum of the 2 children's probabilities) in the list and sort list in reverse again.
— Keep doing this until one element remains in the list : its the root node with probability one.
— Finally, explore tree structure of nodes recursively and encode each child until all characters are encoded.

Care to note that in our implementation sometimes the 0 and 1 are swapped. But the compression rate remains the same and the encoding remains uniquely decodable.

The Input code for TP2 EX7 is : {"a" : 0.05, "b" :0.10, "c" :0.15, "d" :0.15, "e" :0.20, "f" :0.35}

The output code for TP2 EX7 is : {"a" : '0111', "b" : '0110',"c" : '001', "d" : '010',"e" : '000', "f" : '1'}

And in order to generate a Huffman code of any alphabet size q, we have to create a node class with q children instead of just 2. And choose the last q elements of the probability list to create a node with those elements as children at each step.

Question 2

The example sequence given in the course was "1011010100010", and the encoding by the on-line Lempel-Ziv algorithm is '100011101100001000010' The online dictionary obtained was :

Symbol		1	0	11	01	010	00	10
index	0	1	2	3	4	5	6	7
address_lastbit		1	00	011	101	1000	0100	0010

Question 3

In the Basic "Lempel-Ziv" Version, the address of the prefix is encoded with a reserved number of bits equal to the logarithm in base 2 of the total amount of encoded entries. This means that we have to can only encode the addresses after we have completely filled the dictionary.

While the Online version of "Lempel-Ziv" relies on the logarithm in base 2 of the current amount of encoded entries (hence the "online" label) This is advantageous in the sense that we can encode each symbol while inserting it in the dictionary, unlike basic version where we have to loop over the dictionary again, after first filling it with symbols and finalizing the number of entries, in order to encode the symbols.

Also, in the online LZ, no need to re-code all symbols in case we need to add new symbols to the dictionary compared to the basic one where the total number of encoded entries could potentially change the amount of bits reserved to represent the address leading to complete recalculation.

Other advantages for the online version is that it helps compress the information even more. On the other hand, this has to be taken into account in the decoding phase. Since decoding symbols without a fixed number of bits can be tricky.

Question 4

When implementing the LZ77 algorithm, we avoided making copies of lists to optimize computation time. So the remaining_input, window, and look_ahead are all concepts created inside the original sequence of symbols and managed locally with indices (inspired by the course of Data Structures and Algorithms that I'm currently undertaking)

For the example in figure 2, the original text is "abracadabrad" and when LZ77 (of window size 7) is applied, the encoded text is 00a00b00r31c21d74d. We can see tuples of 3 digits (1 octal digit (window_size+1)

for offset, 1 octal digit (window_size+1) for length and 1 4-ary digit for symbol) these have to be converted to bits representation when calculating encoded lengths and compression rates.

2 Source coding and reversible (lossless) data compression

Question 5

The marginal probability of the symbols is calculated from measuring the occurrences of each Morse symbol and dividing them with the total number of symbols in the Morse text. And after applying the Huffman algorithm we find the following codes for each symbol :

Symbol heightProbability	. 0.434	- 0.287	_ 0.215	/ 0.065
Binary Huffman Code	'1'	'00'	'010'	'011'

After encoding the Morse text using the binary Huffman code we get an encoded text of length 2213141 bits.

The length of the Morse text is 1199290 symbols, when multiplied with 2 bits for each symbol ($roundup(log_2(4))$ we get the Morse text bits length equal to 2398580 bits.

Compression rate is $\frac{2398580}{2213141} = 1.084$

Question 6

The expected average length for our Huffman code is calculated using the marginal probability of a symbol and its Huffman binary code length. The value found is 1.848 bits.

The empirical length is the ratio of lengths of the encoded Morse text over that of the original Morse text. The value found is 1.845 bits.

The entropy of our marginal probability distribution is calculated also with value equal to 1.773 Shannon or bits.

The theoretical bound to be checked is :

$$\frac{H(S)}{\log_2 q} <= \tilde{n} < \frac{H(S)}{\log_2 q} + 1 \tag{1}$$

q being the cardinality of our Huffman code (binary so it's 2) and where $H(S)$ is the entropy and $\tilde{n}$ is the empirical length, both calculated above.

we have $1.773 <= 1.845 < 2.773$ meaning that our code is optimal but not absolutely optimal since the equality check is not reached.

Question 7

For increasing input text lengths, we used increasing percentages of the original text and recalculated marginal probabilities, resulting binary Huffman code and empirical lengths.

Original text length is 1,199,290 symbols.

The resulting plot is a shown below :

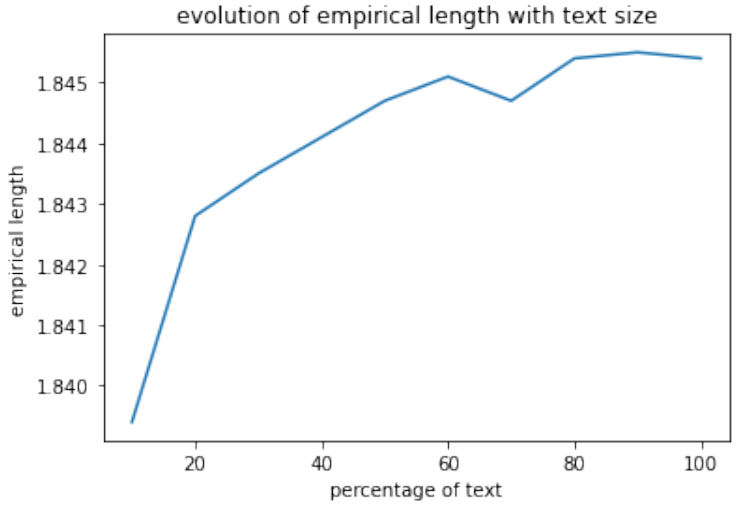

FIGURE 1 – Evolution of the empirical average length of the encoded Morse for increasing percentages of input text length

Question 8

The Lempel-Ziv encoding output is of the form ".1.01_01-000_100.101.000-0011-0100-0000/1010.0101-1000-1101_1000_00011.01000.01011.10000-"

In order to calculate the total encoded length in bits and the compression rate, we have to consider a mapping of the four Morse symbols to 2 bit binary encoding : mapping = {"." : "00", "-" : "01", "_" : "10", "/" : "11"} and encode those symbols as well.

Morse text bits length	2398580
LZ encoded text length before mapping	1530418
LZ encoded text length after mapping	1622722
LZ compression rate	1.478

Question 9

We encoded the text using the LZ77 algorithm with $windowsize = 7$.
The output is of the form "00.11.21_21-33.44_32-31/43.00_42-42_42.22-21/42_62_32.00/41_52.21_21/64-11-41.31.41-54.22-66.42.21/"
Here we did not actually do the mapping of Morse symbols to binary on the LZ77 encoded sequence, but we rather calculated the conversion from decimal system to binary for address and length and from 4-ary to binary for the Morse symbol.
The requested measurements are as follows :

Morse text bits length	2398580
LZ77 encoded text length	3711932
LZ77 compression rate	0.646

Question 10

First method of combining LZ77 and Huffman coding is to start with LZ77 that specialises in detecting and compressing common combination of symbols (words or syllables).
Then when we have the output of the LZ77 which is a long sequence of tuples $(offset, length, nextsymbol)$. We construct the marginal probability distribution of those tuples by running through this sequence and counting occurrences of each tuple. We then encode the tuples using the binary Huffman algorithm.
Another Variation of this method could be to split each tuple into 2 tuples $(offset, length)$ & $(nextsymbol)$

and then calculate occurrences and marginal probabilities and encode each tuple alone using the Huffman algorithm.

The second method could be to start with Huffman coding on the Morse symbols, create the binary code for the whole text and then run LZ77 on it. Huffman specializes in the occurrences of individual symbols rather than combinations, LZ77 then checks for combination of binary bits that are frequently used.

Question 11

We decided to implement the first combination starting with LZ77 and then applying Huffman on the tuples (offset, length, symbol).
In order to distinguish the offset from the length systematically we have reserved blocs of size $roundup(log_{10}(windowsize + 1))$ for each of them. This is noticeable for window sizes 1000 or bigger.(Later questions)
The Huffman dictionary is of the form : {'00.' : 3233, '11.' : 4862, '21_' : 26632, '21-' : 22237, '33.' : 7872, '44_' : 3723, '32-' : 11359, '31/' : 5612, '43.' : 5741, '00_' : 10056, '42-' : 7766, '42_' : 5893,.....
The requested measurements are as follows :

Morse text bits length	2398580
LZ77+Huffman encoded text length	2772030
LZ77 compression rate	0.865

Question 12

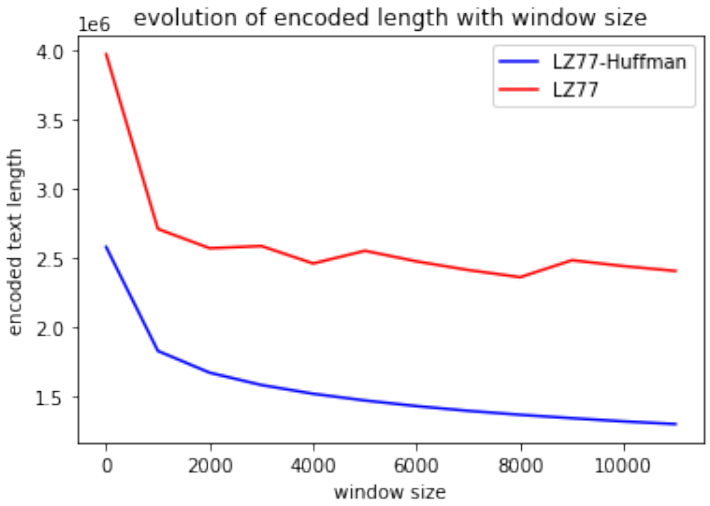

FIGURE 2 – Encoded length for window sizes from 1 to 11000

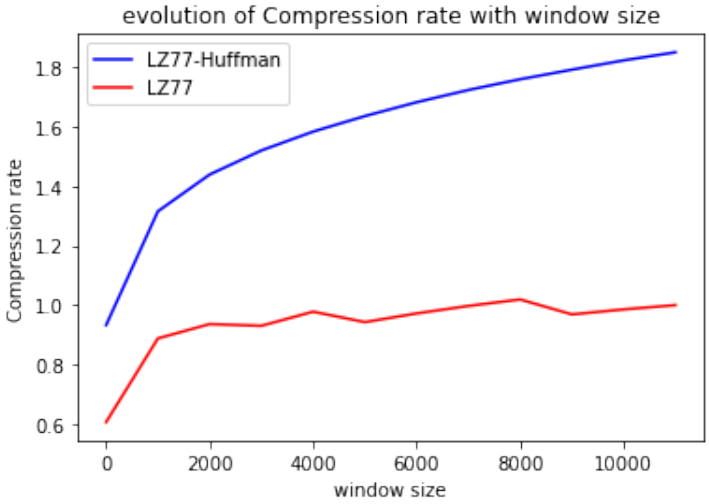

FIGURE 3 – compression rate of encoding for window sizes from 1 to 11000

We can see from the plots that the combination of the 2 algorithms is much better than LZ77 on its own. And that the bigger the window size the better the results. On the other hand the bigger window sizes were taking significant time to be computed.

Comparing with the online Lempel-Ziv, we found compression rates for the LZ77-Huffman combination(maxing at 1.85) that are higher than that of online-LZ (of value 1.478). However the latter has been reached in much less time, which puts us in favor of the online-LZ as best balanced between compression rate and compression time.

Question 13

Since it is typically assumed that repetitions occur at long distances in a text (for instance the name of a character in a book). We can adapt the LZ77-Huffman in a way that the window and the look ahead are not necessarily linked to each other. We can insert a gap between the 2, and check for similarities in occurrences between two distant segments of the texts. This way we don't have to make very big window sizes and we save on computation time. The online-LZ is a good finder of such repetitions since it looks for similarities in all the dictionary.

Question 14

The Huffman dictionary for the text in alphabet is as below :

Code	' '	a	b	c	d	e	f	g	h	i
Symbol	000	0101	011111	011101	00101	110	011100	11111	1000	1011

Code	j	k	l	m	n	o	p	q	r
Symbol	0011000110	0011001	00111	001101	1001	0110	011110	0011000100	1010

Symbol	s	t	u	v	w	x	y	z
Code	1110	0100	11110	00110000	001001	0011000101	001000	0011000111

TABLE 1 – Alphabet Huffman Binary Code

The requested measurements are as follows :

Average expected length	4.15
the experimental length of the encoded text	1711279
compression rate	1.204

Question 15

Since the compression rate of Huffman directly on the alphabet is higher than that on the Morse text, and not just that. The alphabet file length in bits is also smaller than the Morse text in bits. We have a size ratio after Huffman of alphabet over Morse equal to 0.773, meaning the compressed alphabet text is 77% the size of the compressed Morse text so yes it's to directly encode the text with Huffman.

3 Channel coding

Question 17

Since an image signal is quantified with possible values between 0 and 255. We need a bloc of $\log_2(256) = 8$ bits to represent each quantity. This is so we could uniquely encode and decode any picture of fixed size (width*height) with the same amount of bits.

Question 18

Simulating channel effect as noise with probability of 0.01 to flip bits in the binary encoding of the image. After decoding the binary code of the image, the channel effect has lead to change signal value in many pixels inside the picture and the noise is pretty well obvious.

FIGURE 4 – Noisy image after channel effect

Question 20

Introducing redundancy using the Hamming(7,4) Code :

1. We have the binary encoding of the grey signal of all pixels in the image
2. We handle this code by blocs of 4 bits.
3. We calculate the 3 parity checks and insert them right after the bloc of 4 bits.

Decoding image with redundancy using the Hamming(7,4) Code :

1. When decoding we handle the redundant code by blocs of 7 bits.
2. We recalculate the parity checks of the first 4 bits in the bloc
3. If 1 or less bits of difference between the parity check and last 3 bits of the bloc then we consider that the parity check itself is flipped and we take the first 4 bits as they are.
4. If 2 or more bits are different then we have to re-flip a certain bit in the first 4 bits as per hamming algorithm and then we take first 4 bits after re-flipping.
5. Finally we reprocess the signal of the pixels from bits to decimal and we display the image

FIGURE 5 – Less noisy image from channel effect after introducing redundancy

As we can see the image is much less noisy than when sent through channel without redundancy. However some pixels remain wrongly decoded because the channel might have flipped more than one bit in each of the corresponding 2*7-bit blocs for each of those pixels, while the hamming(7,4) code assumes that only one bit has been changed per 7-bit bloc.

Question 21

When trying to reduce loss of information, one tends to increase redundancy. Best way is to send same information an odd number of times and to use the majority vote when decoding, this lowers the loss due to the very low probability of having the majority of equivalent bits flipped. But then the communication rate would plummet $communicationrate = \frac{1}{oddnumberoftimes}$ much lower than $\frac{4}{7}$.
On the other hand, when trying to increase communication rate, one risks to increase the loss of information. There exist less redundant codes with communication rates higher than $\frac{4}{7}$ but more noise will remain after decoding.