Université de Liège

ELEN060-2
Information and Coding Theory

Second Assignment

MVOMO ETO Wilfried - s226625
DELPORTE Guillaume - s191981

Academic year 2022-2023

Contents

1 Implementation

1.1 Huffman Coding

Our implementation is based on list of lists. We had a problem where we could not have symbols more than length one in string format, but it was a problem for the realization of question 6 of the project so we went through a complete overall of the function.
First, we retrieve the symbols and their corresponding probabilities.
Then, we initialize the empty code-words dictionary and a 'huffman_builder' list that we fill to contain some tuples made of the symbol and it's corresponding probability. We encapsulate the symbol into a list to be sure that the symbol can be as long as it needs to be.
After that, the standard Huffman greedy algorithm takes place. We order every tuples in the builder in function of it's probabilities and then retrieve the best candidates as left and right node of our tree builder by using the pop method.
We then check for the left and right node if a code-word is already in the tree and insert accordingly a one or a zero to create the Huffman code. We insert at position 0 because otherwise the code would be 'upside down'.
When the tree is completely built, we simply fill in the code-words dictionary by transforming the list composed of one and zeros into a string.
Verifying our code on Exercise 7, we get this result: Huffman Codewords:

- Symbol: a (prob = 0.05) Codeword: 111
- Symbol: b (prob = 0.10) Codeword: 110
- Symbol: c (prob = 0.15) Codeword: 011
- Symbol: d (prob = 0.15) Codeword: 010
- Symbol: e (prob = 0.2) Codeword: 10
- Symbol: f (prob = 0.35) Codeword: 00

We could modify our algorithm to make sure that the alphabet is of power two when being sent to the function. If it is not the case, we could add some 'padding symbols'. These symbols are just added for the algorithm to be able to run normally.
Then, when the algorithm has successfully created the tree, we would need to remove the padding and modify slightly certain code-words.

1.2 Lempel-Ziv Online

Our implementation start by initializing the dictionary with the null sequence.
We then search for the longest prefix not present in the dictionary and then add it in the dictionary by finding the number of bits in the address to encode by looking at the log2 of the dictionary's size. We also encode the sequence that will be returned by the same occasion.
The example that we were asked to verify our function on is : '1011010100010'. When plugging it into the Lempel-Ziv Online function, we get the answer : '100011101100001000010' and the dictionary at this state is :
{'': (0, ''),
'1': (1, '1'),
'0': (2, '00'),
'11': (3, '011'),
'01': (4, '101'),
'010': (5, '1000'),
'00': (6, '0100'),
'10': (7, '0010')}

1.3 Comparing the Online and Basic version of Lempel-Ziv

To begin with, the basic version of Lempel-Ziv is a dictionary-based compression algorithm. It also needs very long input sequences for the dictionary to be representative and for the algorithm to compress efficiently the text. This can be quite a big drawback if the text that is t encpde is farily small as compression will not be at it's finest. that builds a dictionary of previously encountered strings and replaces them with shorter codes represented by binary dictionary address of the prefix (already in the dictionary) and the last symbol to create a new prefix.
The dictionary is of a fixed size and has enough entries for every prefix needed to encode the sequence. The main advantage is that it is pretty easy to implement and is easy to understand. However, the main problem is that the complete sequence to encode has to be known before hand as the size of the dictionary as to be determined when the sequence is being encoded. This means that streaming data is not suitable for this algorithm.
To come over this problem, engineers have created the online version of Lempel-Ziv. This version of the algorithm removes any problem of address coding because it uses a clever trick were the size of the dictionary is used to determine the number of bits in the address.
This is advantageous as it can encode streaming data because it does not require the entire sequence to be known beforehand. It can also adapt to changes in the input data, as the dictionary is built dynamically.
However, This version is not perfect and also has some drawbacks. One drawback is that the size of the dictionary can grow very large for large input sequences, leading to increased memory usage. Another one is that it is not very competitive in general when compared to other compression algorithm.
It also needs very long input sequences for the dictionary to be representative and for the algorithm to compress efficiently the text.
This can be quite a big drawback if the text that is t encode is fairly small as compression will not be at it's finest.

1.4 Lempel-Ziv 77

The implementation strictly follows the given pseudo-code and we did not add any limitation to the look ahead buffer. The example we had to test was 'abracadabrad' and this led us to the encoded text '00a00b00r31c21d74d'

2 Source coding and reversible (lossless) data compression

2.1 Write a function to read and display both images Give the number of symbols required to represent all possible images in both image representations and the length of both files

The function can be seen in the Jupyter notebook. At first glance the two images look exactly the same, let's see how many symbols are required to represent all possible images in both raw and PNG representations and the length of both files. The number of symbol depends on the size of the image. In our case, the image is of 512*512 pixels.

$$\texttt{\#\_symbols\_for\_one\_image} = 512 * 512 = 262144 \ \texttt{symbols}$$

This means that we need 262144 symbols to represent one image, regardless of the format (the format will compress the size of the file by using various technique but at the end, the number of symbols to represent an image stays the same). We know that in our representation, each symbol is a byte, thus going from 0 to 255.
This means that to represent every possible gray scale 512*512 image, knowing that each pixel can take up to 256 values, is :

$$\texttt{\#\_symbols\_for\_evey\_image} = 256^{\texttt{\#\_symbols\_for\_one\_image}} = 256^{262144}$$

That is a comically large number. By looking at the sizes of each representation, we can determine with no real surprise that the raw image is 262144 bytes long, as no compression whatsoever as been used.
However, for the PNG Format, we can observe that the file is made of 160552 bytes. Thus achieving more or less a 39% reduction in size. That is pretty impressive. The picture that we had to decode is this one:

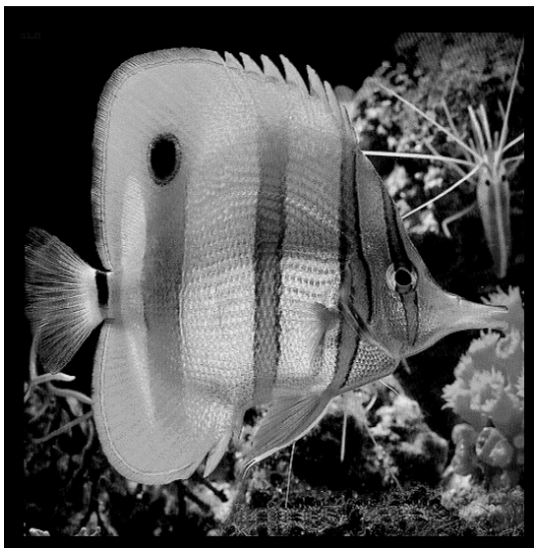


Figure 1: Picture decoded from PNG.txt and pixel.txt

2.2 Estimate the marginal probability distribution of all symbols from the given PNG representation sequence of the image, and determine the corresponding binary Huffman code and the encoded PNG sequence Give the total length of the encoded PNG sequence and the compression rate

The total length of the encoded PNG sequence is 1282749 bits or 160343.625 bytes. When compared to the 1284416 bits or 160552 bytes of the original message, we can see a slight reduction in size and the compression rate that cat be computed by :

$$\texttt{compression\_rate} = 1 - \frac{1282749}{1284416} * 100 = 0.129\%$$

We can therefore see a slight reduction in size compared to the PNG sequence. However, in a real life situation, the code-words dictionary would need to be saved with the file and, as such, we do not think that this technique would be suitable to efficiently compress this type of file. We think that is because PNG extension as already it's compression algorithm baked into it and further compression if complicated as their compression method seems to be pretty efficient.

2.3 Give the expected average length for your Huffman code. Compare this value with (1) the empirical average length, and (2) theoretical bound(s)

By first verifying Kraft's inequality, we can know for sure that there exists a uniquely decodable code associated with the given sequence. We then proceeded to compute the expected average length.
The formula for the expected average length L of a Huffman code with codeword lengths $l_1, l_2, ..., l_n$ and symbol probabilities $p_1, p_2, ..., p_n$ is:

$$L = \sum_{i=1}^{n} p_i * l_i \tag{1}$$

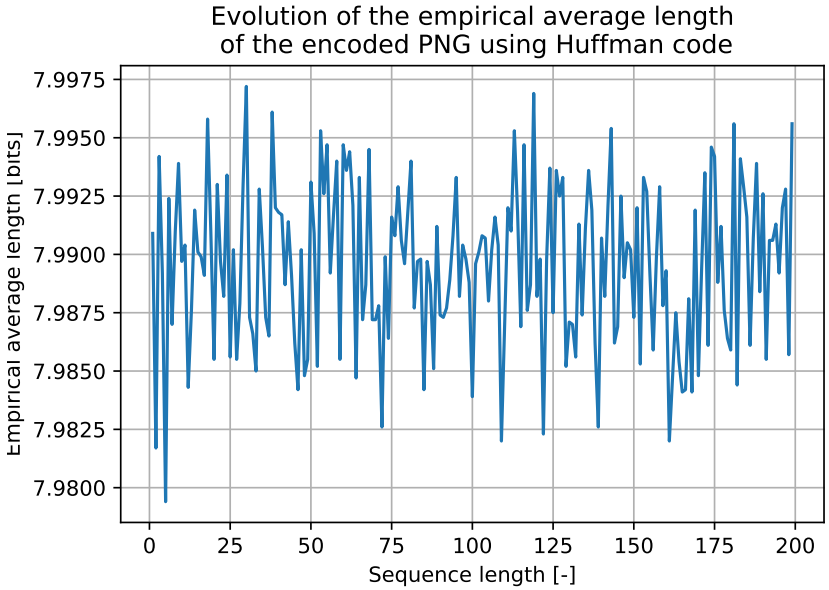
In our case, the theoretical average length is therefore 7.9896 bits. We then used python's random library to create 100 sequences of length 10 by using the marginal distribution we had found earlier. This gave us the empirical average length of 7.992 bits.
We can therefore see that the value found empirically is really close to the theoretical one. This is good news as this means that our calculations seems to hold up together.
We then proceeded to compute the Shannon entropy with Shannon's formula :

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where X is a discrete random variable taking n possible values with probability mass function p_i. The entropy we found is 7.972 bits and this corroborates perfectly our results as the entropy is the minimal bound of bits needed before we loose any information.
This means that in our case we almost are hitting the theoretical limit and that the compression is almost optimal.

2.4 Plot the evolution of the empirical average length of the encoded PNG using your Huffman code for increasing input sequence lengths.

To make this plot and get an average length that had the time to converge, we made each time 1000 repetitions of each sequence length that ranges from 1 to a 200, to be representative.



We can observe that the average length never goes under Shannon entropy and that it most of the time is around the theoretical average of 7.9896 for almost every sequence length, even short ones.

2.5 Encode the PNG sequence using the on-line Lempel-Ziv algorithm. Give the total length of the encoded sequence and the compression rate.

The total length of the encoded sequence is 1072553 characters. However, it is composed of characters that represent 1 bit and some ASCII characters that are represented by 8 bits.
To know the actual size, in bits, of the encoded sequence, we can count how many ASCII characters are in the sequence and then compute the number of bits as such :

$$\texttt{\#\_bits} = (\texttt{\#\_ascii\_char} * 8) + (\texttt{len(encoded\_png\_online)} - \texttt{(\#\_ascii\_char)})$$

This leads us to :

$$\texttt{\#\_bits} = 1536667 \ \texttt{bits} = 192083.375 \ \texttt{bytes}$$

We can therefore compute the compression rate as :

$$\texttt{Compression Rate} \ = 1 - \frac{\#\_bits\_encoded\_sequence}{\#\_bits\_png\_sequence} * 100 = -19.63 \ \texttt{\%}$$

The compression rate is not great. In fact, the encoded sequence is made up of more bits than the actual original sequence.
This is maybe due to the fact that the file as already been compressed to a point where compressing more just leads to a counter productive effect.

2.6 Encode the PNG sequence using the LZ77 algorithm with window_size = 7. Give the total length of the encoded sequence and the compression rate.

The process is similar to the one at the precedent question, we here have determined that, because the window is of size 7, we need 3 bits for each number and 8 bit for the character in each (0-7)(0-7)(a-z) triplet. To compute the number of bits we therefore need to divide the total length of the encoded sequence by 3 and then multiply this result by 14.
This led us to an encoded sequence of 271104.75 bytes and a compression rate of -68.858 %

2.7 Famous data compression algorithms combine the LZ77 algorithm and the Huffman algorithm. Explain two ways of combining those algorithms and discuss the interest of the possible combinations.

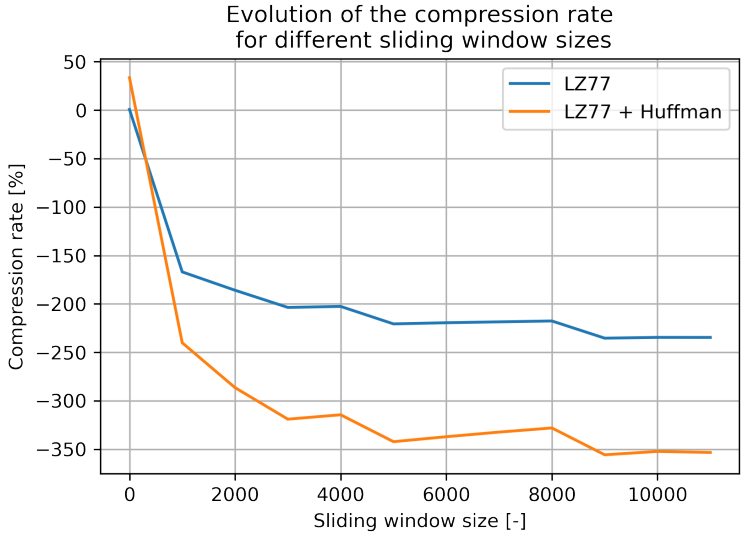
These algorithms can be combined in one of two ways: either by first using LZ77 to identify and compress any repetitive patterns in the data, followed by Huffman coding to encode those patterns, or by first encoding the data using Huffman coding, followed by LZ77 to identify and compress any repetitive patterns in the encoded data.
LZ77 is employed in the first technique to find recurring patterns in the data. The shorter code that results from replacing these patterns can then be further compressed using Huffman coding. This technique has the benefit of enabling more effective repetitive data compression, which can lead to a higher compression ratio.
In the second approach, the data is first encoded using Huffman coding. As a result, the data is represented in a compressed form that is smaller than the original representation. Then, the encoded data is examined for any repetitive patterns and compressed using LZ77.
Combination of LZ77 and Huffman coding are popular because they offer a good balance between speed and compression ratio. While Huffman coding is very effective at symbol encoding in a way that minimizes the total number of bits required, LZ77 is very good at identifying repetitive patterns.

2.8 Encode the PNG using one of the combinations of LZ77 and Huffman algorithms you proposed in the previous question. Give the total length of the encoded PNG sequence and the compression rate.

We chose to first encode the ascii list with huffman and then use LZ77 on the resulting sequence. This led us to a sequence length of 303766.45 bytes and a compression rate of -89.201%

2.9 Report the total lengths and compression rates using (a) LZ77 and (b) the combination of LZ77 and Huffman, to encode the PNG sequence for different values of the sliding window size (use sliding window sizes from 1 to 11000 with a step of 1000). Compare your result with the total length and compression rate obtained using the on-line Lempel-Ziv algorithm. Discuss your results.

The total rates are reported in this plot as we found it to be easy to understand :



We can see that the results are pretty surprising and that the compression rate get worse at each window size. We can also see that the LZ77 + huffman methods leads to even worse performance. This is not what we should expect from such algorithm. Maybe that the compression candidate not good.

2.10 Instead of encoding the PNG sequence, encode directly the pixel values with the binary Huffman algorithm. Give the average expected length, the experimental length of the encoded text and the compression rate.

When encoding directly the pixel values, we find a compression rate of 24.427%. This a significant improvement when compared to the .32% of the PNG sequence encoding. The expected length of the sequence is computed as before and is equal to 6.045 bytes per character. This seams right as it represent a $\approx 20\%$ reduction in size and this is what we get empirically (see the compression rate). The empirical length is 198110.125 bytes which is a nice reduction from the 262144 bytes of the pixel file.

2.11 Compare the values found at the previous question with the ones found in Question 5. In particular, is it better to first encode the text with PNG code before the Huffman encoding or to directly encode the pixel image with Huffman? Discuss.

When we take into account that the size of the PNG sequence is 160552 bytes and that the length of the Huffman pixel sequence encoded is 198110.125 bytes, we can directly see that it is better to use

PNG encoding which is specifically made for image file encoding. Because the huffman encoding on the PNG file brought .32% of compression, it is still better to use Huffman on the compressed PNG File, even if the gain from encoding is minimal.

3 Channel coding

3.1 Implement a function to read the text document and encode the text signal using the binary ASCII fixed-length binary code. Count the number of bits required for the text provided with this assignment (text.txt).

To implement the function asked, we created a function `ascii_to_bin_opener()` that opens a given file and format the ASCII text to a binary string.
In ASCII formatting, each character is represented by an 8 bit number.
We therefore have to multiply by 8 the length of the original text to get the number of bits needed to represent the text document.
We found the length of the text to be 2261, this means that we need 18088 bits to represent the text in binary form.

3.2 Simulate the channel effect on the binary text signal. Then decode the text signal and display the decoded text. What do you notice?

After the simulation, we can notice that the text is not readable anymore. The comprehension is compromised. We wanted to know how many errors had slipped into our text.
With a for loop, we compared the noisy text to the original and found more or less 300 errors. With a text of 2261 characters, this means that $\approx 13\%$ of the text has been corrupted, which is a significant level of corruption for such a small channel noise.
Here is a sample of noisy text :
'THE BOÉ WHO LIVED Mr. ald Mrw& Dursley8 of numâes foUz, Prive4 Drire¬ were pr+ud tɑqay that they weru ðeòfectly normal, thank you repy muc‘."Tbey were t*e lAs Pegple you'd expgct to(‘e involöed an anything stranfe /r m9sterioõs, because they just dkän't hold wiuh s5kh nonsens%. Mv. Dubseey vas the director of e fhrl called Grunnyng3, which made dréhló. He"was a big, fdefy mAn with hardly!any neck, ‘lthOugh lg did havE a very láòge mustache. '
We can see that this text is not comprehensible anymore and this is a perfect example on why we need to implement some redundancy through channel coding. We personnaly would have not thought that such little noise could have so much damage on the resulting text.

3.3 Instead of sending directly through the channel the binary text signal, you will first introduce some redundancy. To do that, implement a function that returns the Hamming (7,4) code for a given sequence of binary symbols. Then, using your function, encode the binary text signal.

To encode the text with Hamming(7,4) mapping, we created a function `hamming_encode` that converts a 4 bit input into it's 7 bit Hamming mapping by doing the dot product with the Hamming

code generator matrix given in the slides. The generator matrix is hard coded as such:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

By doing the dot product with a matrix representing 4 bits we find the corresponding hamming mapping :

$$Hamming\_code = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

This method is pretty strait-forward and we just made a loop that repeat this principle for every 4 bits in our binary encoded ASCII text and append the resulting mapping to an initially empty string.

3.4 Simulate the channel effect on the binary text signal with redundancy. Then decode the binary text signal. Display the decoded text. What do you notice? Explain your decoding procedure.

Let's describe the decoding procedure that we implemented. The goal idea is to compute the syndrome of the message that will tell by itself if the message is corrupted or not. Hamming(7,4) code correction will make us able to find and correct a 1 bit error, however it will only be possible to detect a 2 bit errors but we would not be able to correct it. The Hamming distance is a measure of the difference between two code-words, in the case of Hamming(7,4), it's value is 3. This means that any two valid code-words in the code differ in at least three bit positions. This makes the detection of 1 or 2 bits errors possible but only 1 bit error correcting.
First, we therefore need to compute the syndrome. To be able to do so, we retrieve the 7 bits of the hamming mapping where the first four bits are the data bits and the three last bits are the parity bits. We then use the four data bits and the generator matrix to compute what the parity bits should be like after transmission. To obtain the syndrome, we just add together the received parity bits and the computed parity bits as vectors modulo 2. This gives us the syndrome matrix that normally should look like this when no parity bits violation is found :

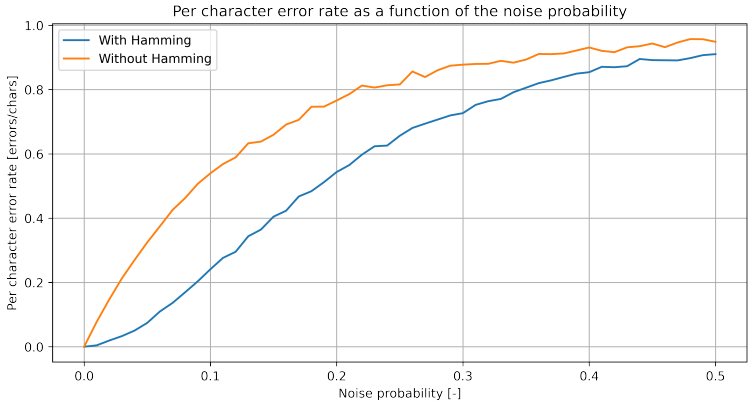
$$syndrome = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

This means that if the syndrome is composed strictly of zeros, the returned data bits by the function should be the one received without any data bits flipped.
However, if that is not the case and the syndrome is different from a null vector, we need to apply some error correction scheme. The syndrome is made of three bits, we therefore need to compute beforehand the 8 possible cases. For each syndrome, one particular bit position is the most probable one that has been flipped.
We therefore implemented a hard-coded dictionary containing every syndrome possible and it's error correcting vector (we will explain just after how we used those) :

syndrome_dict = {
'000': [0, 0, 0, 0, 0, 0, 0],
'001': [0, 0, 0, 0, 0, 0, 1],
'010': [0, 0, 0, 0, 0, 1, 0],
'100': [0, 0, 0, 0, 1, 0, 0],
'011': [0, 0, 0, 1, 0, 0, 0],
'101': [1, 0, 0, 0, 0, 0, 0],
'110': [0, 1, 0, 0, 0, 0, 0],
'111': [0, 0, 1, 0, 0, 0, 0]}
The correction is pretty systematic, we just need to search in the dictionary for the syndrome and then add the error correcting vector to the hamming code that was given as input modulo 2. This will flip the right bit to come back to the more probable state. The null syndrome therefore as a null correcting vector as we do not want the code to be changed before returning it. Now that we have a function to decode the hamming mapping back to bit representation, we just need to loop over the binary text 8 bits by 8 bits and convert it back to text with the use of Python's char() function. The first thing we did after that was to count the number of errors by using the same procedure we used for the text where we did not use Hamming(7,4). We came up to ≈ 30 errors. The text being the same, and therefore the same length, this means a reduction in errors of 90%, which is pretty huge. Here is a small portion of the decoded text :
'THE BOI WHO LIVED Mr. and Mrs. Dursley, of number four, Privet Drive,@were proud to say that they were perfectly normal, thank yoõ very much. They were the last people you'i expect to be involved in anything strange or mysterious, because they just didn't hold with such nonsense. Er. Dursley was the dorector of a firm called Grunnings, which made drills. He was a big, beefy man with hardly any neck, although$he did have a very large mustáche.'
We can see that the text is know comprehensible even if some mistake made it through the error correcting net. This could be due to bits being flipped while close to each other and therefore affecting the Hamming mapping in a way that correction is not possible anymore.

3.5 Given the text document encoded in binary ASCII, implement a Python program to simulate transmission over a binary symmetric channel with a noise probability ranging from 0 to 0.5, with a step of 0.01. Compute the per character error rate for each noise probability, both with and without Hamming (7,4) code. Plot the per character error rate as a function of the noise probability, and compare the performance with and without Hamming (7,4) code.

With the program that we implemented, we came up to this plot : One can observe that when the



noise is not to consequent, the Hamming(7,4) mapping is working like a charm and manage to reduce a fairly significant bit of the errors.
However, it is quickly overwhelmed when the noise become to important. This is surely due to the fact that the more the noise, the closer shifted bits will be clogged together and that makes 2 bits errors (or more) more frequent which lead to a incapacity of the Hamming(7,4) mapping to correct errors.

3.6 How would you proceed to reduce the probability of errors and/or to improve the communication efficiency? Justify.

We could increase the number of bits in the codeword, by using other variants of the Hamming algorithm. This will increase the hamming distance between the code-words and make further error detection possible. However this comes at the expense of more bandwidth requirements.
Another possible solution would be to use more advanced error correcting codes: There are many error correcting codes that are more advanced than the Hamming (7,4) code. Such as Reed-Solomon (the one implemented in CD/DVD technology) codes and Turbo codes. These codes offer even better error correction performance at the expense of increased complexity.
A last way to improve is to apply interleaving. Interleaving is a technique where bits are rearranged in a systematic way before transmission, which helps to spread out errors caused by noise bursts. This can significantly improve error correction performance, especially for bursty errors. However to do such things, we have to know precisely the conditions of the channel used to be able to proceed to the interleaving correctly, when the error bursts are expected.