FACULTY OF APPLIED SCIENCES

INFORMATION AND CODING THEORY

ELEN0060-2

Project 2: Source coding, data compression and channel coding

Authors :
DELCOUR Florian - s181063
MAKEDONSKY Aliocha - s171197

Instructors :
WEHENKEL Louis
Teaching Assistant :
CIOPPA Anthony

April 2022

1 Implementation

Question 1

Here are the main steps of our implementation :

1. A `Node` class to manage the Huffman tree structure.
2. A function `Huffman_code` to apply the Huffman algorithm, so that always merge two nodes that have the lowest probabilities.
3. A function `Huffman_tree` to assign `0` or `1` to each branch of the tree in order to get the code of each symbol.

The output of our implemented algorithm to Exercise 7 TP2 is :

Probability distribution = `{0.05, 0.10, 0.15, 0.15, 0.20, 0.35}`
Associated codes = `{'000', '001', '100', '101', '01', '11'}`

which is not exactly the same as answer as the one provided in TP2. However, we know that there are several optimal Huffman codes when there are multiple choices to merge two nodes with the lowest probabilities. Thus, doing this exercise by hand, we easily see that these associated codes are valid and optimal.

Extension for Huffman code of any alphabet size :

- ❊ Change the `left` and `right` nodes by a `child` list of size `alphabet_size` that will contain nodes with lowest probabilities. Then, instead of merging the two lowest probability nodes, we would merge the nodes in `child` list.

Question 2

Dictionary and encoded sequence for T = `1011010100010` :

Key	Entry index	Binarized address + bit
`' '`	0	`' '`
`'1'`	1	`'1'`
`'0'`	2	`'00'`
`'11'`	3	`'011'`
`'01'`	4	`'101'`
`'010'`	5	`'1000'`
`'00'`	6	`'0100'`
`'10'`	7	`'0100'`

Table 1: Dictionary from LZ on-line algorithm

The associated encoding sequence is U = `100011101100001000010` and corresponds to the solution in the theoretical course.

Question 3

Basic Lempel-Ziv algorithm :

- ❋ Advantages : Can reach the optimal solution asymptotically, i.e with a sufficient large text and under some assumptions.
- ❋ Drawbacks :
  - Source must be stationary and ergotic.
  - Set the length of the binary address without previously knowing the size of the input text. Thus the address length could be too large for a small text size (not efficient nor optimal) or it could be too small for a large text size (full dictionary). So we need a mechanism to prevent the dictionary to be full otherwise the algorithm will be inefficient.

On-line Lempel-Ziv algorithm :

- ❋ Advantages :
  - No assumption about the source (very robust)
  - Adaptive size of the dictionary $\rightarrow$ Gain of memory, optimal
  - Flexibility thanks to the dictionary management allowed by variants
- ❋ Drawbacks :
  - Must have very long input text to reach asymptotic performance
  - Can be less efficient due to the robustness

Question 4

Encoded sequence for T = `abracadabrad` and `window_size` = 7 using `LZ77` algorithm :

U = `00a00b00r31c21d74d`

which is the solution of the provided example.

2 Source coding and reversible (lossless) data compression

Question 5

The marginal probability distribution of all symbols from the Morse text is computed by counting the occurrence of each symbol and divide it by the total number of symbols. Then we can use the Huffman algorithm to get the associated code of each symbol and eventually encode the Morse text. Results are shown in Table 2.

Symbol	Probability	Huffman code
`'.'`	0.4338	`'0'`
`'-'`	0.2871	`'11'`
`'_'`	0.2145	`'101'`
`'/'`	0.0646	`'100'`

Table 2: Probability distribution and Huffman code of Morse symbols

The total length of the encoded Morse text is 2 213 141 bits, considering that we need 2 bits to encode Morse symbols. The compression rate is given by :

$$\text{Compression rate} = \frac{length\_Morse}{length\_encoded} \cdot \frac{log_2\, 4}{log_2\, 2} = 1.0838$$

We observe that there is almost no compression. This can be explained by the fact that Morse symbols are already an coding of the Latin alphabet. Therefore, it is complicated to find more efficient symbols.

Question 6

The expected average length of the Huffman code is given by :

$$\bar{n} = \sum_{i=1}^{Q} P(s_i) \cdot n_i = 1.845$$

where Q is the input alphabet size, $P(s_i)$ is the probability of the symbol s_i and n_i is the length of the associated Huffman code.

The empirical average length is also 1.845 since we compute the Huffman code with empirical probabilities calculated on the same text as the one used to build the code.

The theoretical bounds are given by :

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

$$1.771 \leq \bar{n} < 2.771$$

and we see that the expected average length is included in the bounds. Thus, by the first Shanon theorem, the code is optimal. Indeed, we know that the Huffman code produces an optimal and prefix-free code such that this inequality is respected.

Question 7

On Figure 1, we observe that the empirical average length is growing with the length of Morse text. This can be explained by the fact that one or more Morse symbol are much present in the beginning of the input text and thus they will have a higher probability distribution. This has the effect of reducing the empirical average length because these symbols with higher probability will have a smaller Huffman code. Larger the input Morse text, larger will be the empirical average length until we reach a plateau. Indeed, the probability distribution of all symbols tends to their real values (large numbers law). Thus the empirical average length converges to the expected average length.

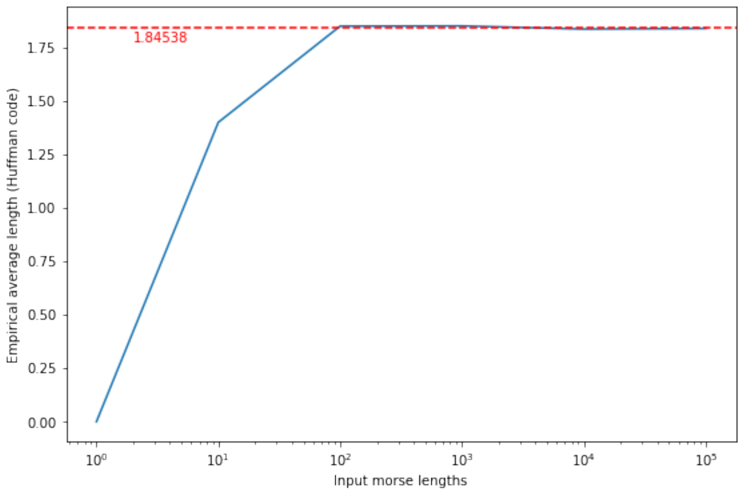

Figure 1: Empirical average length of Huffman coding Morse for increasing input text lengths

Question 8

The encoded sequence of the `LZ on-line` algorithm is a mix of binary symbols and Morse symbols. Thus, we must take into account the different alphabet size and the total length of the encoded sequence is given by :

$$\begin{aligned} \text{length binary symbols} &= 1\ 438\ 097 \cdot log_2\ 2 = 1\ 438\ 097 \text{ bits} \\ \text{length Morse symbols} &= 92\ 304 \cdot log_2\ 4 = 184\ 608 \text{ bits} \\ \text{Total length} &= 1\ 622\ 705 \text{ bits} \end{aligned}$$

The compression rate is given by :

$$\text{Compression rate} = \frac{length\_Morse}{Total\_length} \cdot log_2\ 4 = 1.4781$$

The compression has performed well because the encoded Morse text is about 1.5 times smaller than the original one. The input text length (=2 398 580 bits) is a sufficient length (as seen in Q3) to achieve nice performance.

Question 9

The encoded sequence of the `LZ77` algorithm is a mix of number symbols (ranging from 0 to window size) and Morse symbols. Thus, we must take into account the different alphabet sizes and the total length of the encoded sequence is given by :

$$\begin{aligned}\text{length number symbols} &= 927\ 982 \cdot log_2\ (window\_size + 1) = 2\ 783\ 946 \text{ bits}\\ \text{length Morse symbols} &= 463\ 990 \cdot log_2\ 4 = 927\ 980 \text{ bits}\\ \text{Total length} &= 3\ 711\ 926 \text{ bits}\end{aligned}$$

The compression rate is given by :

$$\text{Compression rate} = \frac{length\_Morse}{Total\_length} \cdot log_2\ 4 = 0.6462$$

In this case, this is a really poor result as the encoded sequence length is larger than before the compression (about 5/3 times larger).

Question 10

Huffman coding is useful to reduce the number of bits/symbols of a sequence, based on their probability distribution. However, this method does not allow to identify repeated occurrences of series of symbols. On the other hand, `LZ77` allows the identification of these repetitions but it does not minimize the number of bits used for the representation.

1. A first way to combine Huffman and LZ77 encodings is to start by encoding the sequence using LZ77 algorithm which would output a coded sequence composed of tuples. Then, we can further compress this output by applying Huffman coding on the remaining Morse symbols given their probability distribution in the LZ77 output sequence.
2. A second way is to first encode the sequence using Huffman coding, then compress further by applying LZ77 algorithm which can detect repetitions.

Question 11

We decided to encode the Morse text with the first way, although we tried both methods but the second one yields poorer results.

The encoded sequence of the `LZ77-Huffman` algorithm (`window_size` = 7) is a mix of number symbols (ranging from 0 to window size) and binary symbols from the Huffman code. Thus, we must take into account the different alphabet size and the total length of the encoded sequence is given by :

$$\begin{aligned}\text{length binary Huffman symbols} &= 927\ 980 \cdot log_2\ (2) = 1\ 855\ 960 \text{ bits}\\ \text{length number symbols} &= 927\ 982 \cdot log_2\ (window\_size + 1) = 2\ 783\ 940 \text{ bits}\\ \text{Total length} &= 3\ 711\ 926 \text{ bits}\end{aligned}$$

The compression rate is given by :

$$\text{Compression rate} = \frac{length\_Morse}{Total\_length} \cdot log_2\ 4 = 0.6462$$

We can observe the same conclusion as in Question 9. Notice that we have the same total length and compression rate as in Q9. This can be explained by the fact that there is not a enough discrepancy in distributions of the remaining Morse symbols, that are encoded using Huffman algorithm. It means that the remaining Morse symbols are all encoded using 2 bits like it was in Question 9 and thus we obtain the same results for **window_size = 7**.

Question 12

Figure 2 shows the evolution of the encoded Morse total length for `LZ77` and `LZ77-Huffman` algorithms and for different sliding window sizes. Figure 3 shows the associated compression rate.

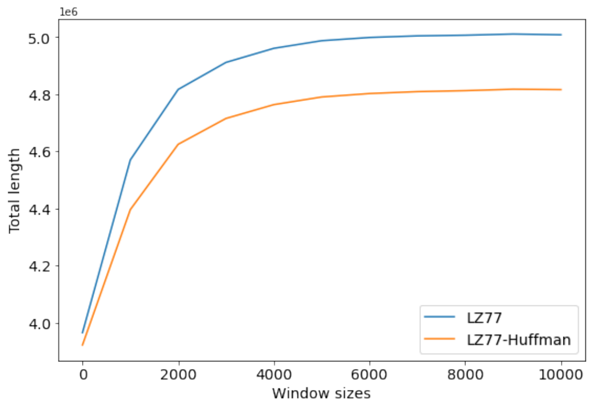

Figure 2: Total length in bits

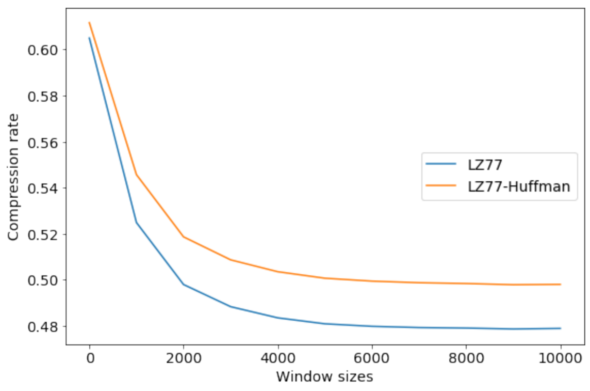

Figure 3: Compression rate

As we can see, both `LZ77` and `LZ77-Huffman` algorithms performs a really poor compression and it becomes even worse as the window size increases. This can be explained by the fact that looking for occurrences in a window size is countered by the space taken by this window size when the text is encoded. Indeed, addresses bits d and p are encoded on a number of bits which depends of the window size.

However, we can notice that `LZ77-Huffman` algorithm performs a slightly better compression. Applying Huffman after `LZ77` allows to compress further the coded sequence because it reduces the number of bits required to encode Morse symbols depending on their probability distributions (these probabilities depend on the window size).

The `LZ on-line` algorithm achieve greater compression compared to these two algorithms (1.4781). Its total length (1 622 705 bits) is at least 3 times smaller and at most 2.5 times smaller which is a noticeable difference.

Question 13

1. `Huffman` : Applying Huffman coding on Morse text does not compress a lot the input sequence as Morse is already a coded text. Indeed, the low alphabet size does not allow Huffman to exploit the disparity in symbols distributions and thus is quite

inefficient. An input text coming from a larger alphabet size (Latin alphabet for example) with various symbols distributions would lead to a more efficient Huffman code. However, it does not output coded text longer than the original one.

2. `LZ77` : Since we know that repetitions occur at long distances in a text, we can forget `LZ77` which should then have gigantic window sizes and thus involve long calculations to find a prefix. According to our results, it always encodes the text with a greater length, as the compression rate is always lower than one.
3. `LZ77-Huffman` : As it starts by applying `LZ77`, we can have a similar conclusion as above. However, our results showed that applying Huffman after `LZ77` on Morse text helps a bit to perform a better compression but not that much because Morse is already a coded text (low alphabet size so low disparity in symbols distributions) leading to inefficiency of Huffman algorithm as seen in question 5. We could be more optimistic if the input text was coming from a bigger alphabet size with more variation of symbols distributions (Latin alphabet for example). In such a case, the Huffman algorithm could be exploited to its full potential.
4. `LZ on-line` : The on-line version of the Lempel-Ziv algorithm is the best data compression algorithm to encode Morse text. Indeed, it’s the one with the best compression rate (1.4781) on Morse text. This type of dictionary really correspond to this kind of data where repetitions occur at long distances. Searching in dictionary is a costless operation in time, it only requires to store the dictionary.

Question 14

Table 3 shows the binary Huffman code obtained from the original text.

Symbol	Code	Symbol	Code	Symbol	Code
’ ’	111	’i’	0100	’r’	0101
’a’	1010	’j’	1100111001	’s’	0001
’b’	100000	’k’	1100110	’t’	1011
’c’	100010	’l’	11000	’u’	00001
’d’	11010	’m’	110010	’v’	11001111
’e’	001	’n’	0110	’w’	110110
’f’	100011	’o’	1001	’x’	1100111010
’g’	00000	’p’	100001	’y’	110111
’h’	0111	’q’	1100111011	’z’	1100111000

Table 3: Binary Huffman code for each symbol of the original text

The expected average length of the Huffman code is given by :

$$\bar{n} = \sum_{i=1}^{Q} P(s_i) \cdot n_i = 4.15047$$

where Q is the input alphabet size, $P(s_i)$ is the probability of the symbol s_i and n_i is the length of the associated Huffman code.

The empirical average length is also 4.15047 since we compute the Huffman code with empirical probabilities calculated on the same text as the one used to build the code.

The compression rate is given by :

$$\text{Compression rate} = \frac{length\_original}{length\_encoded} \cdot \frac{log_2 27}{log_2 2} = 1.1456$$

Question 15

Encoding directly the original text with Huffman lead obviously to a better compression rate (1.1456) than passing by Morse text (1.0838) because Morse is already a coded alphabet. However, encoding directly with Huffman gives a coded sequence of length 1 711 279 bits whereas passing by Morse leads to 2 213 141 bits, which is longer. We observe that directly encoding the original text yields better results. This is caused by the fact that Morse coding acts in a similar manner than Huffman but with less effectiveness. Indeed, it is does not take into account the actual input text. It just gives less symbols to Latin character that often appears in language (comes from England). Unlike Morse, Huffman coding takes into account the probability distribution of input symbols to make a better compression. Thus, Huffman becomes less effective with less input symbols than directly encoding the original alphabet.

3 Channel coding

Question 16

With the `PIL` library it is really easy to load an image. We can use `PIL.Image.open('image.png').convert('L')`. The L mode in which we convert the image is a mode where the image is in 8-bit (to have number between 0 and 255) pixels, black and white. Then the function `img.show()`, where `img` is the image we opened with the previous function, will display our image.

Question 17

We need a fixed-length binary code, then all symbols of the alphabet must have the same size. Considering that each pixel has a value between 0 and 255, each pixel binary representation can be encoded with 8 bits since $\log_2 256 = 8$. So 8 bits are sufficient to represent all numbers between 0 and 255.
We have $0 \rightarrow 00000000$, $1 \rightarrow 00000001$, $2 \rightarrow 00000010$, ..., $254 \rightarrow 11111110$, $255 \rightarrow 11111111$
As the size of the image is 1984×1116 pixels, we need $1984 \times 1116 \times 8 =$17 713 152 bits in total to encode the image.

Question 18

Since we have a binary symmetric channel with a probability error `p` equal to 0.01, the channel works as follows :

- If we send a `'0'`, we have a probability of 0.99 to receive a `'0'`, and a probability of 0.01 to receive a `'1'`.
- If we send a `'1'`, we have a probability of 0.99 to receive a `'1'`, and a probability of 0.01 to receive a `'0'`.

We re-encode the 8-bits sequence received in the corresponding grayscale pixel value between 0 and 255. The noisy image is shown on Figure 4.

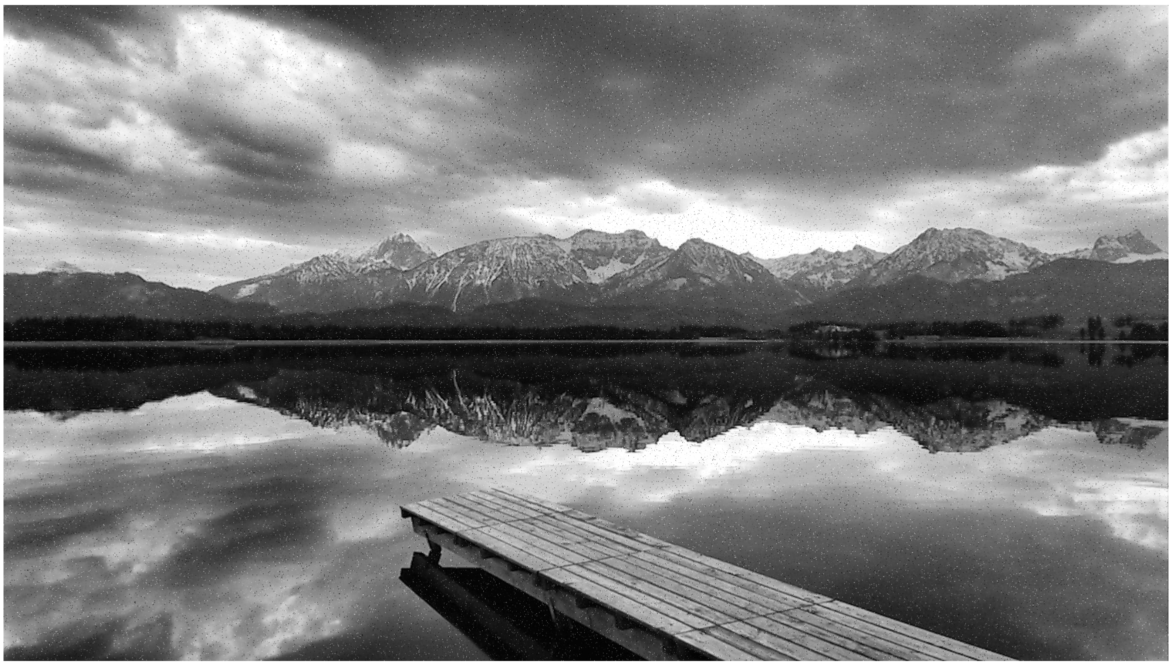

Figure 4: Channel effect on binary image signal

We see that the image remains barely the same but pixels are modified quite everywhere. Indeed the probability of a pixel to be modified is $1-(1-p)^8 = 1-(1-0.01)^8 = 0.077$, so approximately 7.8% of the pixels will be modified.

Question 19

The Hamming (7,4) code of a 4 bits signal $s_1s_2s_3s_4$ is $s_1s_2s_3s_4p_1p_2p_3$ with $p_1 = s_1+s_2+s_3$, $p_2 = s_2+s_3+s_4$ and $p_3 = s_1+s_3+s_4$ in binary.

To use the Hamming (7,4) code, we must divide the binary sequence encoding each grayscale value of the pixels into 2 parts of 4 bits, then we compute the Hamming (7,4) code of each of those 2 parts and we encode each pixel with this 14-bits sequence (4 signal bits - 3 parity bits - 4 signal bits - 3 parity bits). Table 4 shows some examples.

Pixel value	Binary code	Hamming (7,4) code
124	`01111100`	`0111-010-1100-011`
233	`11101001`	`1110-100-1001-110`
87	`01010111`	`0101-101-0111-010`

Table 4: Examples of Hamming (7,4) coding

Question 20

Once we have converted all of our pixels into 14 bits sequence with Hamming (7,4) code, we can make them go through the channel. We compute the Hamming (7,4) code of each of the 4 signal bits sequences received, and we compare the parity bits computed with the one received. If there is 0 different bit, we most probably received the correct signal bits. If there is 1 different bit, the most probable scenario is that the parity bits received contain an error, thus we don't have to correct the signal bits received. But if there is 2 or 3 different bits, then the most probable scenario is that there is an error in the signal bits and we can correct the wrong one with the parity circles :

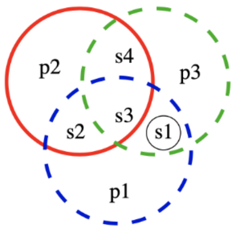

Figure 5: p = parity bit, s = signal bit

Once all the sequences have been computed and potentially corrected, we re-encode the binary image by transforming the 8-bits sequences (we only keep the signal bits) into their corresponding grayscale pixel value. The decoded image is shown in Figure 6.

We observe that there are less modified pixels than with no redundancy but it's still not exactly the original image (but close to it), and it's because an error of more than 1 bit in the 7-bits sequence received can't be corrected with this method.

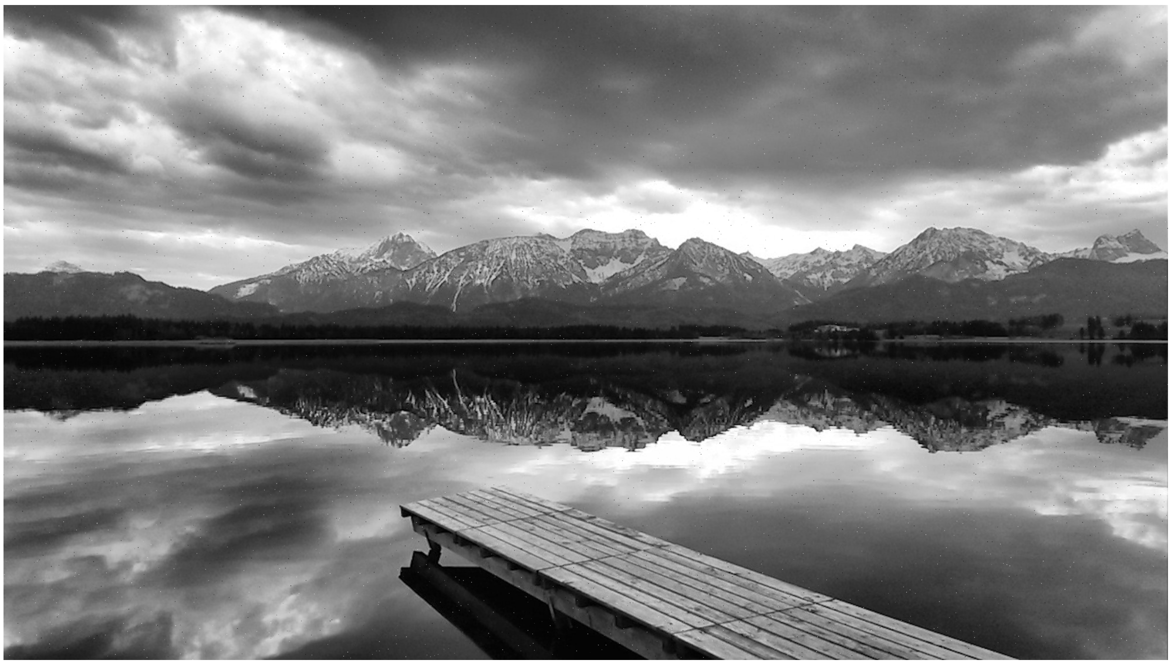

Figure 6: Decoded image using Hamming encoding

Question 21

We could use repetition codes, which repeats each bits three times and so if in a 3-bits sequence we have one bit different from the 2 others, the most probable scenario is that this bit is wrong and that the correct bit represented by this 3-bits sequence is corresponding to the 2 other bits (if we receive 010, the most probable scenario is that the 1 is wrong and that the bit encoded by this sequence is 0).
It should improve our results since having 2 errors in a 3-bits sequence is less probable than having 2 errors in a 7-bits sequence (Hamming (7,4) code).
We could also combine those repetition codes with Hamming (7,4), here is how we would do for a 4-bits signal:

1. Compute the Hamming (7,4) code of those 4 bits, we get a 7-bits sequence
2. Triple every bits of the 7-bits sequence (repetition codes), we have a 21-bits sequence
3. Send the 21-bits through the channel
4. Go on the received sequence 3 bits per 3 bits and keep only the most present value in those 3 bits (if we have 010, the 1 is most probably wrong, thus we keep 0), we recover a 7-bits sequence
5. Use the Hamming (7,4) algorithm on this remaining sequence

This algorithm is way more time consuming than the previous one, but it should reduce much the loss of information.