LIÈGE université
Sciences Appliquées

ELEN0060 - INFORMATION AND CODING THEORY

PROJECT 2 - REPORT

Source coding, data compression and channel coding

Authors:
Antoine DECKERS (s170999)
Antoine DEBOR (s173215)

Lecturer:
Pr. L. WEHENKEL
T.A. :
A. SUTERA

Sunday 2nd May, 2021

Source coding and reversible (lossless) data compression

The python script used to compute and display all the results asked for this section can be found in the file `source_coding.py` in our archive.

1. The function returning a binary Huffman code for a given probability distribution is called `huffman_procedure` and can be found in the file `Huffman.py`. The main steps of the implementation are the following:

   (a) First, the probability distribution is transformed in a list of *Node* objects, each of these having four attributes:
   - *left* and *right*: the children of the node, the left one corresponding by convention to the largest probability value (see *value* here after). They are both equal to None if the current node is a leaf and both *Node* objects otherwise .
   - *value*: the probability value associated to the current node. It is equal to an element of the probability distribution if the node is a leaf and to the sum of the *value* attributes of the children of the current node otherwise.
   - *word*: the index of each probability value in the original probability distribution, used to ease the construction of the returned object. This attribute is None for all nodes that are not leaves.

   (b) This list of Node objects is then sorted by increasing *value* value.

   (c) This sorted list is then used and updated to build a Huffman tree following the following steps:

   While the list contains more than one Node object:

   i. The two *Node* objects with the smallest *value* values are used as children to create a new *Node* object. The new *Node* object's attributes are instantiated as explained before.
   ii. The two merged *Node* objects are removed from the list and the new one is appended.
   iii. The updated list is again sorted by increasing *value* value.

   (d) The remaining *Node* object in the list is the root node and is used to generate the Huffman code. The `huffman_code` function recursively walks back through the generated tree while generating the different code words by calling itself with the children of the considered node as argument. Once the leaves are reached, a dictionary is updated with the *word* attributes of the leaf *Node* objects as keys and the corresponding code words as values.

   (e) The generated dictionary is finally used to build the returned list, containing the code words in the same order as the corresponding probabilities initially given as argument to the function.

   In order to extend this function to any alphabet size n, several changes would have to be done:

   - The *left* and *right* attributes of the *Node* class should be replaced by a array-like object (*e.g.* a list) containing the n children of each node. For the leaves, these would again be instantiated as None.
   - Instead of merging the two nodes corresponding to the smallest probability values, the function should merge the n nodes corresponding to the smallest probability values. These n nodes would then be given as the children of the new node and removed from the list. The new node would then be appended to the list.
   - In the `huffman_code` function, generating the different code words should be done by assigning a different character to each of the n children of each non-leaf node.
   - To cope with probability distributions corresponding to a non-valid number of leaves regarding the size of the alphabet (*e.g.* if $n = 3$ and there are 4 leaves, this will lead to two nodes having to be merged with an alphabet of size $n = 3$ which is not valid), a prior addition of a relevant number of zero-valued leaves should be done to ensure the completeness of the algorithm without compromising its properties.

The implementation has been verified on Exercise 7 of the second list of exercises. The code can be launched and the displayed output is reported in table 1.

probability	code word
0.05	000
0.1	001
0.15	100
0.15	101
0.2	01
0.35	11

Table 1: Huffman code generated for Exercise 7 of the second list of exercises

Although the output is not the same as the one derived during the tutorial, it is a valid Huffman code. This can be seen in fig. 1, where the leaves have not been represented in a particular order for the sake of clarity. However, one can go through the tree and convince oneself that the generated code is indeed valid.

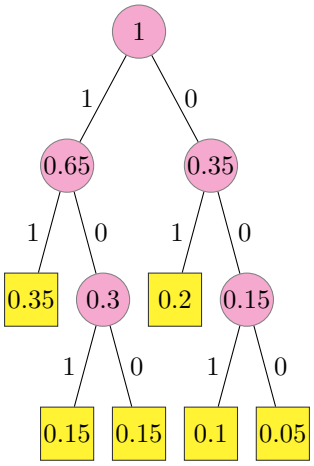

Figure 1: Huffman tree for Exercise 7 of the second list of exercises

2. Given a sequence of symbols, the function returning a dictionary and the encoded sequence using the *on-line* Lempel-Ziv algorithm is called `online_LZ` and can be found in the file `Lempel_Ziv.py`. The example given in slide 50/53 of the theoretical course can be reproduced by launching the code: the generated dictionary is reported in table 2 and the generated bit stream is `100011101100001000010`.

source word	(address, bit)
1	(, 1)
0	(0, 0)
11	(01, 1)
01	(10, 1)
010	(100, 0)
00	(010, 0)
10	(001, 0)

Table 2: Online Lempel-Ziv dictionary

It is noteworthy that a specific handling has been implemented to deal with sequences finishing with a previously-seen source word, such as the genome sequence provided for this project (available in the file `genome.txt`). In such a case, the last tuple is built as usual but not considering the last symbol as part of the already-seen source word. In the dictionary, the corresponding source word is denoted as 'end'.

3. Two versions of the Lempel-Ziv algorithm have been seen in the theoretical course: the basic version and the *on-line* one. Both algorithms are dictionary methods for data compression, *i.e.* they rely on a set of frequently used words to parse the input text. This parsing process consists in encoding the given sequence as a sequence of addresses in the dictionary coupled with additional bits. They rebuild the dictionary on the fly for each text, incrementally as the text is read, *i.e.* they are adaptive methods.

   It can be shown that, for both versions, the average number of bits per symbol asymptotically converges to the entropy rate $\mathcal{H}(\mathcal{S})$ almost surely for messages of any stationary ergodic source, which means that for the size of the encoded sequence going to infinity, the average number of bits per symbol is optimal. Therefore, this method can be described as a "universal" algorithm. The basic version however suffers from one main drawback: address coding. Indeed, the basic version needs to know the size of the dictionary prior to the address coding, and therefore this version does not present an *on-line* character.
   The *on-line* version addresses this problem by using the current dictionary size to determine the number of bits, and therefore shortens the size of the encoded text. Indeed, when encoding the n-th word, one knows the maximal address of the partial dictionary in which one should seek for the prefix ($c(n)$), and can thus encode the address using $\lceil \log_2 c(n) \rceil$ bits. This version thus presents an adaptativity property as it builds a local dictionary.
   As this has been said earlier, the asymptotic performances do not change between both versions, but the *on-line* version turns out to be better than the basic version on the finite-size texts encountered in practice. This is noteworthy as the asymptotic performances are reached only when the dictionary starts to become representative, *i.e.* when it contains a significant fraction of sufficiently long typical messages, which can represent a tremendous amount in terms of source text length. This is a consequence of the fact that these methods are not able to exploit *a priori* knowledge about the source text's nature.
   According to the theoretical slides, the *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. From a practical point of view, these methods still provide good computational performances and good results on numerous types of source text.

4. The function returning the encoded sequence using the *LZ77* algorithm given an input string and a sliding window size l is called `LZ77` and can be found in the file `Lempel_Ziv.py`. It implements the *LZ77* algorithm as described in the statement, but merging the prefix definition and the *offset*, *length of the prefix* and *symbol following the prefix* definitions into a single function, called `get_prefix` and which can be found in the same file. It is not specified in the statement how to deal with strings ending with an already-seen pattern, such as the *cororico* sequence, for instance. Therefore, in such a case, it has been arbitrarily decided to set the *symbol following the prefix* variable of the output tuple as an empty string, as there is no symbol left after the last detected prefix. Another way to handle this would be to add a 'end' signal at the end, similarly to what has been implemented for the *on-line* Lempel-Ziv. Furthermore, note that the size of the look-ahead buffer has been constrained to the size of the slicing window + 1 when calling `get_prefix`, as manipulating a larger sequence does not make any sense since the prefix can not be longer than the window's size.

   The example given in the statement can be reproduced by launching the code, which defines *abracadabrad* as the input sequence and l=7 as the window size. The returned and displayed output is the expected one, *i.e.* (0, 0, 'a'), (0, 0, 'b'), (0, 0, 'r'), (3, 1, 'c'), (2, 1, 'd'), (7, 4, 'd').

Codon (a_i)	Probability (p_i)	Code word ($c(a_i)$)	Length (l_i)
TTG	0.0219	111110	6
ATC	0.0168	100111	6
AGT	0.0137	010110	6
CTA	0.0115	000011	6
ATA	0.0199	110111	6
TGG	0.0142	011011	6
CGT	0.0097	1101001	7
GAG	0.0114	000010	6
GTG	0.0113	000001	6
ACG	0.0096	1100111	7
CCG	0.0082	1001001	7
ACT	0.0143	011100	6
GAA	0.0241	00101	5
TCA	0.0194	110101	6
CAG	0.0137	010111	6
TCG	0.0117	001000	6
TAC	0.0121	010000	6
GCG	0.0072	0111100	7
ACC	0.0105	1110001	7
GTC	0.0097	1101000	7
AGG	0.0104	1110000	7
TCT	0.0177	101100	6
AAG	0.0178	101110	6
AAA	0.0453	0001	4
CGA	0.012	001100	6
GTT	0.0177	101101	6
CGG	0.0079	0111111	7
CTG	0.0138	011001	6
TAT	0.02	110110	6
AGC	0.013	010010	6
TGC	0.0132	010011	6
GGG	0.0078	0111110	7
GAC	0.0093	1100001	7
TTC	0.0242	00111	5
GGC	0.0095	1100110	7
CAA	0.0214	111001	6
GCA	0.0132	010100	6
CTC	0.0121	001101	6
TCC	0.0139	011010	6
GCT	0.0138	011000	6
GCC	0.0088	1100000	7
CCA	0.0143	011101	6
CCC	0.0077	0111101	7
CGC	0.008	1001000	7

Table 3: Marginal probability distribution of all codons in the given genome

5. The marginal probabilities of all codons in the given genome are illustrated on table 3. The total length of the given genome is 958557 and the length of the encoded genome is 1885514. Considering that the size of the source alphabet is 4 (A,C,G,T) and the code alphabet is binary (0,1), we can compute the compression rate (CR) as follows :

$$CR = \frac{genome\ length}{encoded\ genome\ length} * \frac{log_2\ 4}{log_2\ 2} = \frac{958557}{1885514} * log_2\ 4 = 1.0168$$

6. The expected length $L(C, X)$ of a symbol code C for an ensemble X is defined as

$$L(C, X) = \sum_{x \in \mathcal{A}_X} P(x) l(x).$$

Let's consider the ensemble as the set of the 21 codons and the symbol code as the one on table 3. One can therefore compute the expected average length of our Huffman code such as :

$$L(C, X) = \sum_{i=1}^{I} p_i l_i = 5.9 \ bits/symbol$$

The empirical length can be computed as the length of the encoded genome divided by the number of symbols in the genome text. $958557/3 = 319519$ codons were identified in the text which lead to an empirical length equal to :

$$\frac{1885514}{319519} = 5.9 \ bits/symbol$$

One can notice that the empirical average length is equal to the expected average length. Let consider $\mathcal{X}$ as the random variable indicating the codons in our sample. The empirical average length was computed using the empirical measures of the codons in our genome sample, i.e our observations, without using the probability distribution of $\mathcal{X}$. Theses values are associated with the empirical distribution function (EDF) of our sample. Moreover, the average expected length was computed using the probability distribution of $\mathcal{X}$. This distribution is associated to the cumulative distribution function (CDF) of $\mathcal{X}$. EDF is an unbiased estimator of CDF and they have the same asymptotic behaviour. Indeed, by the strong law of large numbers, for an infinitely large sample size, EDF will converge to CDF. Based on the above reasoning and our sample size of 1885514, we can explain the equality of those two quantities.

By definition, a Huffman code should achieve an expected length that satisfies :

$$H(X) \leq L(C, X) < H(X) + 1$$

One can therefore compute the source entropy of the codon alphabet :

$$H(X) \equiv \sum_i p_i \log \frac{1}{p_i} = 5.867 \ bits$$

and use the values derived in the previous equations to conclude that our implementation is correct and satisfy this property.

$$5.867 \leq 5.9 < 6.867$$

One can notice that our code is almost absolutely optimal (it would be the case if the expected length would equally satisfy the lower bound of the inequality).

7. The empirical average length has been computed for an increasing input genome length, starting from an initial length of 57 bases incremented by steps of 300 bases until reaching the total genome length of 958557. This leads us to have 3195 values for the EAL illustrated on fig. 2.

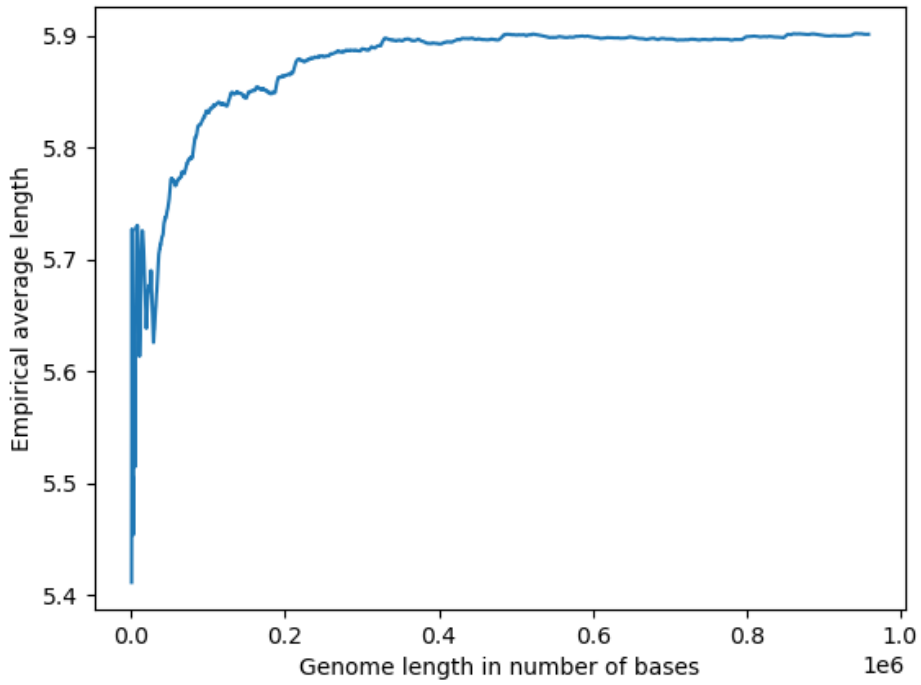

Figure 2: Evolution of the empirical average length for increasing input genome lengths

One can easily observe that, quite naturally, the EAL converges to 5.9. However, the graph is not strictly increasing as this can be seen for low input genome length (IGL) values. This local "sawtooth" behaviour is actually strongly dependent on the Huffman code generated and one would not observe the same graph for another code. Indeed, for small genome lengths, a rare (resp. frequent) codon, corresponding to a longer (resp. shorter) code word, will have a larger effect on this ratio. Therefore, the observed spikes can be explained by the presence of rarer codons in the genome, suddenly increasing the above-mentioned ratio, followed by more frequent ones, decreasing this ratio subsequently. As the length increases, the rarity and thus the impact of a given codon on the average length decreases, which explains the apparent progressive smoothing of the curve.

N.B.: The EAL as a function of the IGL can be fitted using a *logistic function* such as

$$f(IGL) = \frac{K}{1 + a \cdot e^{-r.IGL}}$$

where the coefficient K is equal to the empirical average length derived for the full genome length (5.9), r >0 and a any number.

8. The Huffman algorithm we have seen in the theoretical course and the one implemented in this assignment is the *static* version of the algorithm. One idea would be to use another version, called the *adaptative* Huffman code. This technique permits to dynamically adjust the Huffman tree as the data is being transmitted, having no initial knowledge of the source distribution. This alternative only needs one-pass encoding compare to two for the *static* version. The main disadvantage is that the tree is modified a lot and so the computing time is not optimal. However, the compression rate is the best amongst the different versions of the algorithm. Moreover, as the source does not need to be known in advance, this algorithm can encode data stream in real time (ex. audio and video signal). Using the static version, we had to know a file containing the genome to be encoded while with this version, we

can encode the data as the genome sample is being sequenced. Our source model is not a static tree anymore but a dynamically changing tree.

Amino Acid (a_i)	Probability (p_i)	Code word ($c(a_i)$)	Length (l_i)
Ala	0.0431	11101	5
Arg	0.0644	1001	4
Asn	0.0492	0010	4
Asp	0.0255	01100	5
Cys	0.0304	01110	5
Gln	0.0351	10101	5
Glu	0.0354	11010	5
Gly	0.0415	11100	5
His	0.0292	01101	5
Ile	0.0678	1011	4
Leu	0.1	000	3
Lys	0.0631	1000	4
Met	0.017	011111	6
Phe	0.0678	1100	4
Pro	0.0408	11011	5
Ser	0.0894	1111	4
Thr	0.0514	0100	4
Trp	0.0142	011110	6
Tyr	0.032	10100	5
Val	0.0509	0011	4
Stop	0.0514	0101	4

Table 4: Marginal probability distribution of all amino acids in the given genome

Another idea would be to consider the source as emitting amino acids rather than codons. The fact that several codons can code for the same amino acid in some cases can be taken into account by encoding such codons with the same codeword. By doing that, several codons are merged together which increases the probability of observing the corresponding symbol, as the probabilities of each codons are added up together. Therefore, by following such a method, table 4 contains only 21 rows and the amino acids yields more probable symbols and thus smaller code words, which leads to a smaller encoded genome length and therefore, a bigger compression rate :

$$CR = \frac{genome\ length}{encoded\ genome\ length} * \frac{log_2\ 4}{log_2\ 2} = \frac{958557}{1366176} * log_2\ 4 = 1.403$$

However, such an encoding scheme could not be used for every application. Indeed, by proceeding as explained, the decoding part would not be able to recover the original *codon* stream, but only the *amino acid* one. Therefore, some information is lost during the encoding regarding the original stream. If the input *amino acid* stream is considered as a pre-processed version of the original *codon* stream that has to be used in a system considering amino acids only, the encoding scheme can be considered as valid. However, if the pre-processing step is used to improve the CR while the output target is still a codons stream, then the encoding scheme is not valid.

9. Using the algorithm illustrated in the course's slides for the *on-line* Lempel-Ziv algorithm, one will build a dictionnary with each key representing a sequence of symbols (taken from the following ones : A, C, G, T) and each value a tuple. The first element in this tuple is the binary address of the key and the second element is the last symbol of the key sequence which is in this project, one of the four bases C,G,T,A. Therefore, the two element of this tuple are not encoded in a same alphabet. In order to compute a meaningful compression rate, we decided to binarize the four bases on two bits, each

one corresponding to one element of the set $\{00, 01, 10, 11\}$. Please note that this is only taken into account when computing the CR, *i.e.* the output stream is NOT modified. A last remark is that the commas are not considered as part of the encoded genome's alphabet, because one actually does not need them to decode the encoded stream. Indeed, as a given size of the dictionary yields a given size for an encoded address, one can deduce the total size of each tuple and decode the stream in an *on-line* fashion.
The encoded genome length is 1905755. Considering a binarization of the letters in it to compute the CR, one adds the number of letters in the stream to this length, which yields a value of 2018912. Therefore, the compression rate can be computed as

$$CR = \frac{genome\ length}{encoded\ genome\ length} * \frac{\log_2 4}{\log_2 2} = \frac{958557}{2018912} * \log_2 4 = 0.9496.$$

10. In order to encode the genome using the LZ77 algorithm, one should carefully choose the size of the window (as discussed in question 4, the size of the look-ahead buffer has been constraint to window + 1). One could try to use a window size of 22. Therefore, we could encode the four bases (A,C,G,T) and the length/offset values using an alphabet of size 26, like the English alphabet, while not caring about the commas. Using this method, we obtain a compression rate of :

$$CR = \frac{genome\ length}{encoded\ genome\ length} * \frac{\log_2 4}{\log_2 26} = \frac{958557}{1591115} * \frac{\log_2 4}{\log_2 26} = 0.256.$$

This method could be replicated to any language alphabet. However, the compression rate obtained seems to be quite low.
Another approach would be to binarize the four bases on 2 bit, as shown at the previous question, and limit the size of the window to a power of 2 in order to encode each tuple on a desired amount of bits. For instance, if we want to encode each tuple in the ouput stream on 24 bits (3 bytes), we must limit the size of the widow to $2^{11} bits$ (22 bits for the length/offset and 2 for the symbol). This approach is discussed for different window sizes in question 13.

11. LZ77 and Huffman detect and remove two different kinds of redundancy in data. First of all, the LZ77 algorithm performs well at detecting redundant blocks of consecutive symbols. This algorithm could thus firstly filter the input stream by removing consecutive symbols redundancy into an intermediate sequence (composed of (length,distance,symbol) tuples according to this algorithm). The Huffman algorithm, on its side, is good at compressing the stream using the symbol frequencies alone. The algorithm could then compress this intermediate sequence, with the same performance as if we had not applied LZ77 before. Therefore, by using a combination of the LZ77 and the Huffman algorithm (in that order) we could achieve a better net compression compared to the one we would get if we used these algorithms separately.
In more details, one knows from theory that the LZ77 algorithm asymptotically converges towards the source entropy (asymptotic optimality). This happens because the output of the LZ77 algorithm asymptotically converge to complete randomness. However, in a practical implementation, the size of the dictionnary is of finite length, which means that the distribution of the output of the LZ77 is quite far from being totally random. This non-randomness implies that a further compression of the data is possible. One first way of performing this second compression on the output of the LZ77 algorithm would be to apply LZ77 again. A second way combines LZ77 with the Huffman algorithm and applies Huffman on the output of LZ77, as explained here above. This second way is actually the basic principle behind several algorithms, among which the *deflate* one. The reason why Huffman is used rather than LZ77 is mainly because, in the LZ77 output, symbols are more likely to be less dependent from each other than in the original sequence. Therefore, the LZ77 loses its main advantage of taking dependencies into account and the Huffman algorithm, which is way simpler, is prefered. The most common algorithm using this combination is called the *deflate* algorithm and is implemented in the *ZIP*, *gzip* and *PNG* formats.

Another way of combining these two algorithms would be to apply Huffman first, and then the LZ77 algorithm. However, Huffman considering each input symbol in an independent fashion, the resulting

binary stream could not be exploited as efficiently by the LZ77 algorithm as the original 4-bases sequence. Moreover, a non-negligible drawback of the Huffman algorithm is that, unlike LZ77, it has to know the whole sequence to compute the statistics. Therefore, if the genome sequence had to be compressed in an *on-line* fashion, for instance to avoid memory issues, the Huffman algorithm could not be used in the first place. In such a case, applying LZ77 first would be the best choice.

12. According to the previous question, the best combination of the LZ77 and Huffman algorithms consists in applying LZ77 first, which will then output a list of (distance, length, char) tuples. In order to apply the huffman algorithm on this stream, one should transform these tuples into a sequence of codes belonging to a specific alphabet, as it is done in the *deflate* algorithm for instance. However, for the sake of simplicity, it has been decided to apply the Huffman algorithm considering each tuple of the LZ77's output as a source symbol on its own. The proportion of each tuple in the total stream has thus to be determined before performing the Huffman encoding. This naive method is however not expected to provide tremendous results. Indeed, the different possible tuples all correspond to small frequencies, with only a few of them being consistently 'more probable' than the others, though presenting very small frequencies too. This is illustrated in fig. 3, which presents the distribution of the different possible tuples in the encoded genome. Please note that the x-axis only represents the indices of the tuples in the stored dictionary. Note also that the 'wavelet' behaviour of the graph simply comes from the order according to which the tuples are considered by the `unique` function from *NumPy*.

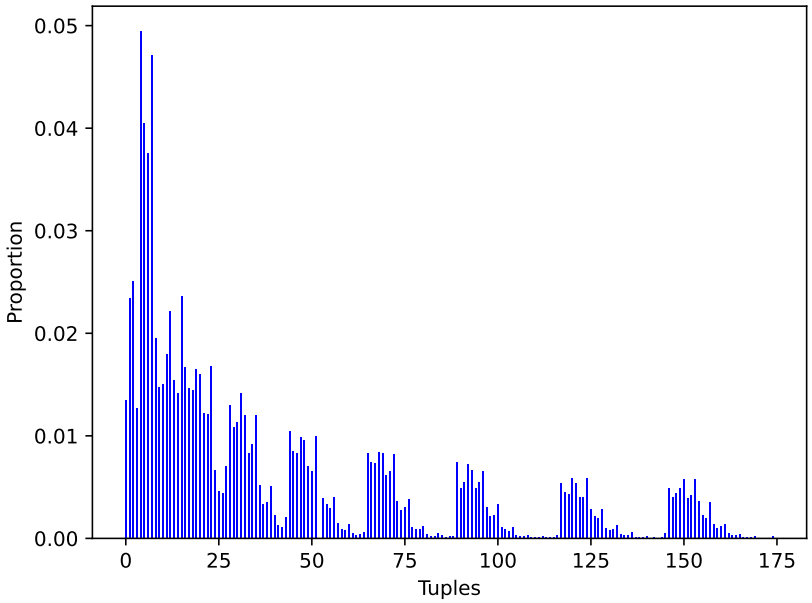

Figure 3: Caption

The implementation of this combination can be found in the file `LZ77Huffman.py` of the submitted archive. By using a window size of 7, it yielded a total length for the encoded genome equal to 2412566 symbols (alphabet size = 2) and a compression rate equal to

$$CR = \frac{958557}{2412566} * \frac{log_2 4}{log_2 2} = 0.7946.$$

As expected, the compression rate is really poor.

Window size	LZ77 Encoded Length	LZ77 CR	LZ77-Huffman Encoded Length	LZ77-Huffman CR
128	2185552	0.877	1350216	1.42
256	1813752	1.057	1069427	1.8
512	1482500	1.293	838330	2.29
1024	1194270	1.6	651274	2.94
2048	949632	2.02	500558	3.83
4096	747630	2.56	378601	5.06
8192	586040	3.27	283024	6.77
16384	452610	4.23	203595	9.41
32768	349728	5.48	145699	13.16
65536	272272	7.04	103377	18.54

Table 5: Data comparison between LZ77 and LZ77-Huffman for different window size

13. The choice of the alphabet used to modify the tuples of the output stream of the LZ77 in order to compute a *meaningfull* compression rate is the one discussed at the end of question 10. The length and compression rate obtained using the LZ77 algorithm and the LZ77-Huffman combination are illustrated on table 5. We can see that for an increasing input window size, both algorithms perform better as the CR is significantly increasing. Moreover, as expected, the combination of LZ77 and Huffman yields to a better CR for every single window size. For small window sizes (128-256), the compression rate obtained are similar to the one obtained using the `on-line` Lempel-Ziv algorithm but, as soon as the window size starts to get bigger, these two algorithms outperform by far the compression rate obtained using the *on-line* Lempel-Ziv algorithm. Note that the more the window size increases, the more the complexity of these algorithms increases and the computing time becomes larger. Therefore, a trade-off has to be made between the compression rate and the computing time of the algorithm.

14. If repetitions do occur at long distance in a genome, a first idea would be to consider the $LZ77$ algorithm as a starting point. Indeed, this algorithm can exploit the fact that repetitions occur at long distance by taking large window sizes. If these are taken too small, the algorithm can not cope with that property and loses its main interest. This algorithm can be completed by applying the Huffman algorithm on top of it, as it is done in the previously mentioned *deflate* algorithm. Even the naive implementation presented here above shows interesting results for large window sizes. On the other hand, the Huffman algorithm alone showed pretty good results in the above, as the expected average length was close to the entropy. Therefore, even if it does not explicitly take advantage of the redundancy of blocks of codons, the Huffman encoding technique remains a good candidate for encoding the genome. The CR is however not as large as the one presented by the LZ77-Huffman combination with large window size which outperforms all the other techniques.

Channel coding

The *Python* script providing the presented results can be found in the file `channel.py` of the submitted archive. This script imports the files `utils.py`, `BSC.py` and `Hamming_7_4.py` of the same archive.

15. The data contained in the file `sound.wav` is extracted using the function `load_wav` from `utils.py`. This function returns both the sample rate and the data from the `.wav` file. The first quantity is equal to 11025 Hz and the data contains 86565 samples, which corresponds to a 7.85 s long audio message. The plot of the sound signal contained in the file `sound.wav` can be seen in fig. 4.

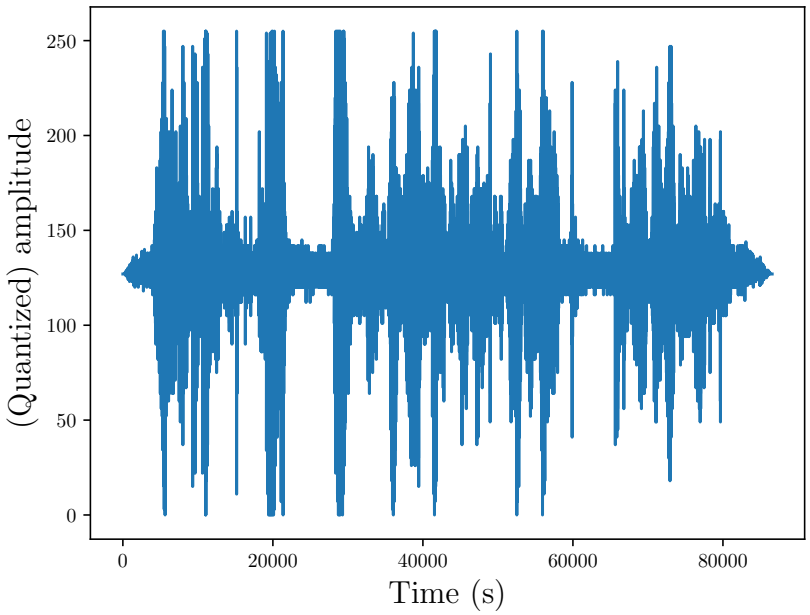

Figure 4: Original sound signal of `sound.wav`

This sound signal can be listened to using the `playsound` function from the *Playsound* package. It corresponds to a quote from *Mission : Impossible II*:

Boss: You mean it's going to be difficult.
Ethan Hunt: Very.
Boss: Well, Mr. Hunt, this is not Mission: Difficult, its Mission: Impossible. Difficult should be a walk in the park for you.

16. Each sample of the quantized signal is a value between 0 and 255. Therefore, in order to encode the signal using a fixed-length binary code, the minimal required number of bits is 8 since $2^8 = 256$. This is the appropriate number of bits because it ensures to have a uniquely decodable code. This encoding is performed using the `binarize` function from `utils.py` and all words are then concatenated to obtain a continuous bit stream stored as a list of strings.

17. The effect of the binary symmetric channel is simulated in the function `BSC` from `BSC.py`. This function simply takes as input a bit stream and a given probability of error p, and randomly flips bits in this stream with probability p. The bit stream generated in the previous point can thus be fed to this function and the resulting noisy bit stream can be decoded using the function `bin_to_dec` from `utils.py` and then saved as a `.wav` file in the file `sound_BSC.wav` using `save_wav` from `utils.py`. For $p = 0.01$, one can thus generate the sound plot of fig. 5.

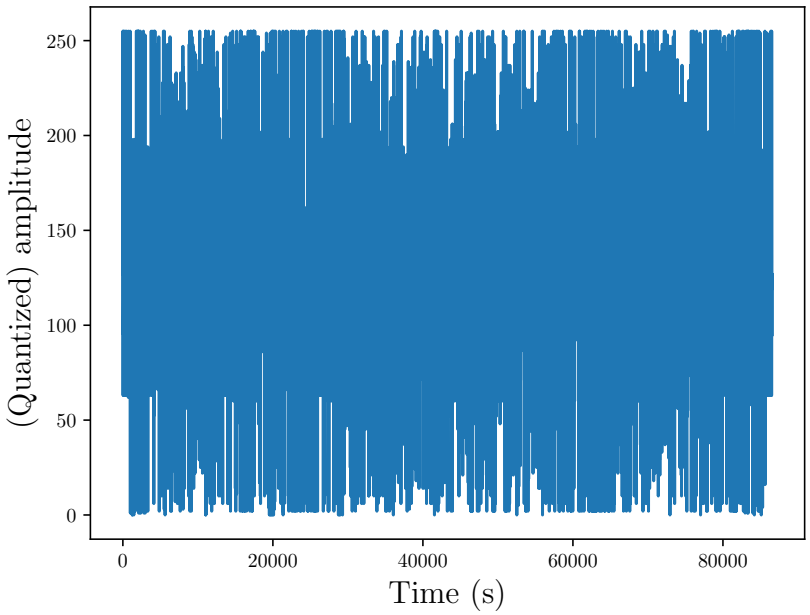

Figure 5: Noisy sound signal of `sound_BSC.wav`

This sound signal can again be listened to using the `playsound` function and corresponds to a noisy and unpleasant-to-hear version of the original quote. This is coherent with the plot from fig. 5, in which one can see the detrimental effect of the binary symmetric channel's non-zero error probability: the original signal is surrounded with a lot of noise, due to randomly flipped bits and resulting in the bad quality of the audio signal.

18. The function returning the Hamming (7,4) code for a given sequence of binary symbols is the function `Hamming_7_4_cod` from the file `Hamming_7_4.py`. It simply adds redundancy to the signal by adding 3 parity bits to each 4-bits long block of the original non-redundant signal following the Hamming (7,4) procedure. The resulting bit stream is thus $\frac{7}{4}$ times longer than the original one. The original binary sound signal stream can thus be fed to this function to add redundancy in order to increase its robustness against noisy channels.

19. The effect of the binary symmetric channel is again simulated in the function `BSC` from `BSC.py`. The resulting noisy bit stream is then decoded using the same procedure as the one followed during the *ELEN0060* tutorials:

    (a) *For each 7-bits long block of the bit stream*:

        i. Decompose the block in two parts: the signal bits <0:3> and the parity bits <4:6>.
        ii. From the signal bits, compute the *expected* code word using the previously mentioned Hamming (7,4) procedure. Extract the corresponding *expected* parity bits <4:6>.
        iii. Compute the syndrom as the modulo-2 bit-to-bit sum of the parity bits and *expected* parity bits.
        iv. Determine the most probable error pattern. This is done using the circles of parity (*i.e.* the Hamming (7,4) Venn diagram), whose graphical representation can be seen in fig. 6.

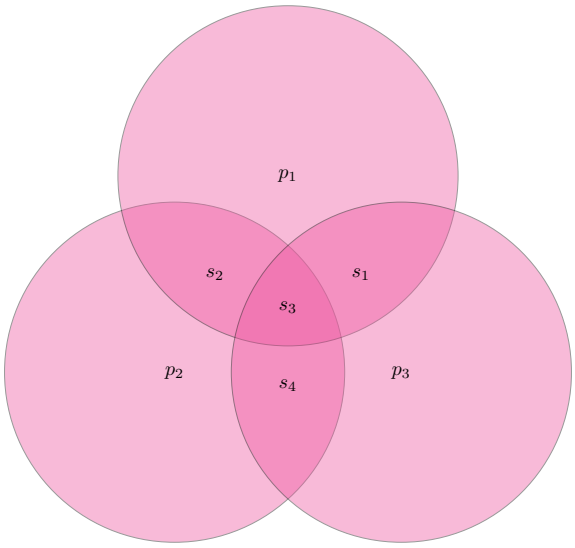

Figure 6: Venn diagram of the Hamming (7,4) code

and is implemented through a predefined dictionary linking each syndrom to its most probable error pattern. Note that, if only one bit of the syndrom is set, the corresponding most probable error is upon a parity bit, and the received signal bits are assumed error-free. This is also the case if the bits of the syndrom are all cleared.

v. Compute the modulo-2 bit-to-bit sum of the signal bits with the most probable error pattern bits, which corresponds to applying a correction to the received code word if needed. This decoded word can finally be appended to the output bit stream.

(b) The decoded output bit stream can be decoded using again the function `bin_to_dec` and then saved as a `.wav` file in the file `sound_BSC_Hamming.wav` using `save_wav`.

This procedure is implemented in the function `Hamming_7_4_dec` from the file `Hamming_7_4.py`. For $p = 0.01$, one can thus generate the sound plot of fig. 7.

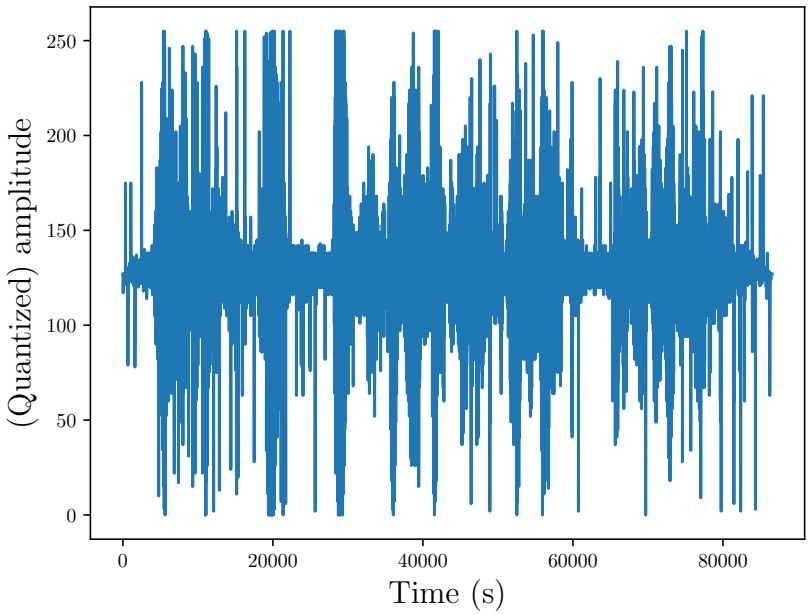

Figure 7: Decoded sound signal (using the Hamming (7,4) code) of `sound_BSC_Hamming.wav`

This sound signal can again be listened to using `playsound` and corresponds to a noisy version of the original quote but in which the noise quantity has been dramatically reduced comparing to the sound of the file `sound_BSC.py`. The background noise does not surround the signal of interest as it was the case before adding redundancy using the Hamming (7,4) code. This is coherent with the plot of fig. 7, which presents a shape similar to the original signal's one, as the quantity of randomly modified values has been decreased.
Still, some noise remains, which is not surprising since the Hamming (7,4) code does not allow to correct multiple errors. Indeed, if several bits are flipped within the same code word, the decoding process is not able to correct these errors and the wrong code word is not corrected.

20. In order to reduce the loss of information only, one could add more redundancy to the signal. For instance, one could combine the repetition strategy with the Hamming code. In such a strategy, a 4-bits long word would be first coded using the Hamming (7,4) code, and the corresponding 7-bits long code word would then be repeated several times. The decoding process would so combine both the Hamming decoding strategy previously described and a majority vote over the repetitions, for each bit of the considered word. However, this technique would degrade the communication rate as more bits would have to be sent to transmit the same quantity of information of interest.
From a hardware point of view, another way to reduce the loss of information would be of course to enhance the quality of the channel used, by improving the physical devices underlying the channel, *e.g.* the electronic components.

    To reduce the loss of information and improve the communication rate, one would select a more advanced coding technique than the Hamming (7,4) code, *e.g.* convolutional or turbo codes. These coding structures implement a more efficient way of adding redundancy while requiring a more complex decoding stage. As it can be seen in the figure presented in slide 31/37 of the data transmission set of slides from the theoretical lectures, except for very low power-to-noise ratios, convolutional and turbo codes present bit error rates way better than repetition codes such as the Hamming (7,4) code. Actually, turbo codes approach the Shannon limit way closer than the two other mentioned coding techniques and, given a certain power-to-noise ratio threshold, one can achieve bit error rates as low as one desires without the need to increase the power injected in the transmission system.