University of Liège

ELEN060-2 - Information and Coding Theory

Master in Computer Science and Engineering & Master in Data Science

Project 2 - Source Coding, Data Compression and Channel Coding

Authors:
Martin Peeters
Maxime Amodei

Professor:
Louis Wehenkel
Teaching Assistant:
Antonio Sutera

May 2, 2021

1 Source Coding and Reversible (lossless) Data Compression

1.1 Question 1

The program can be divided in two parts. The first consists in generating the Huffman tree from a probability distribution, which is represented as a priority queue. Using such a data structure allows to easily add or remove elements. When there is only one node left, the second part begins. From the root node, the Huffman algorithm generates a string for each child that is stored in a dictionary. Recursion provides a very sleek implementation for trees.

Our implementation only works for a binary Huffman code. In order to handle any alphabet size, a few adjustments are required. The structure of a node would be slightly less convenient. One could represent the n children as a list. Thus, some loops would be added because the alphabet size is unknown at compile time. Moreover, when building the tree, we need to group the leaves by n. Similarly, when assigning a digit when visiting the tree, we must consider n branches.

The probability distribution is P(S) = [0.05, 0.10, 0.15, 0.15, 0.2, 0.35]. Let assume that each value is respectively associated to the labels ['A', 'B', 'C', 'D', 'E', 'F']. Then, the program computes the following results:

Label	**Huffman Code**
F	11
E	01
C	101
D	100
B	001
A	000

Because the code obtained is prefix free, we can claim that it is also uniquely decodable.

1.2 Question 2

The Lempel-Ziv encoding produces a sequence of words that are composed of a binary address and a symbol. The binary address points toward the representation of the prefix in the dictionary. The encoding was done using the algorithm provided in the theoretical course. To decode the encoded words in the sequence we first split them in the two parts regarding the address and the symbol, then we can obtain the prefix and re-assemble to original word. The encoding of the message `1011010100010` gives the representation `1`, `00`, `011`, `101`, `1000`, `0100`, `0010` as we obtained in the theoretical course.

1.3 Question 3

The main differences between the offline and online versions is the address coding. The offline version first looks at the whole message before choosing the length of the address in the encoded sequence. In contrast the online version will start with short addresses representation and increase the length as it progresses. The decoding of the first version could be slightly easier but the decoding of the online version isn't too computationally intensive. The main advantage of the online version is the single pass on the data: for very long message doing multiple passes when one does the job is wasteful and this difference can be significant.

1.4 Question 4

The algorithm is implemented following the pseudo code of the statement, on the word `abracadabrad` with a window of size 7 it produces `[(0, 0, 'a'), (0, 0, 'b'), (0, 0, 'r'), (3, 1, 'c'), (2, 1, 'd'), (7, 4, 'd')]`.

1.5 Question 5

Codon	Number	Huffman Code
AAA	14476	0001
TTT	13943	11110
AAT	9995	10001
ATT	9899	10000
TTC	7736	00111
GAA	7692	00101
TTA	7022	111111
TTG	6992	111110
TAA	6898	111011
CAA	6827	111001
ATA	6383	110111
TAT	6376	110110
TCA	6193	110101
TGA	5979	110010
CTT	5975	110001
AAC	5723	101111
AAG	5686	101110
GTT	5669	101101
TCT	5662	101100
CAT	5545	101011
TGT	5494	101010
ATG	5436	101001
ACA	5433	101000
ATC	5384	100111
AGA	5264	100110
GAT	5183	100101
ACT	4565	011101
CCA	4565	011100

TGG	4542	011011
TCC	4439	011010
CTG	4420	011001
GCT	4410	011000
CAG	4386	010111
AGT	4381	010110
GGA	4241	010101
GCA	4230	010100
TGC	4223	010011
AGC	4155	010010
GTA	3893	010001
TAC	3862	010000
CTC	3861	001101
CGA	3851	001100
CAC	3799	001001
TCG	3744	001000
CTA	3684	000011
GAG	3630	000010
GTG	3607	000001
TAG	3549	000000
GGT	3485	1110101
CCT	3406	1110100
ACC	3357	1110001
AGG	3325	1110000
CGT	3102	1101001
GTC	3089	1101000
ACG	3059	1100111
GGC	3050	1100110
GAC	2976	1100001
GCC	2821	1100000
CCG	2609	1001001
CGC	2533	1001000
CGG	2517	0111111

GGG	2500	0111110
CCC	2472	0111101
GCG	2316	0111100

We consider the input T with newline removed but encoded as we received it, i.e. with 8 bits per symbol as it has an UTF-8 encoding with 1 byte per character.

Let T be the genome and C(T) its coding. Then, the length of the encoded sequence is

$$l(C(T)) = 1,885,514 \text{ bits.}$$

The compression rate of C on T is

$$\frac{l(T)}{l(C(T))} = \frac{8 \times 958,557}{1,885,514} = 4.067037423.$$

We consider that each character of the genome (i.e., A, C, G, and T) can be stored with one byte.

In this case the Huffman algorithm is not doing much useful compression: if the input was received in a more condensed format with two bits per symbol, the Huffman algorithm would only benefit from the higher number of some codons (e.g., 'AAA', 'TTT'). This is because the marginal probabilities of the symbols are close to uniform: only a few symbols are rare enough to receive a shorter encoding.

When considering the input in such a condensed format, the compression rate of C on T is

$$\frac{l(T)}{l(C(T))} = \frac{2 \times 958,557}{1,885,514} = 1.016759356 \text{ bits}$$

which is a much more reasonable value. Indeed looking at the distribution of the codons from the long table above there is practically no disparity so the Huffman algorithm doesn't have a lot of potential.

1.6 Question 6

The Expected Average Length can be computed as follows : $EAL = p_{AAA} \times 4 + p_{TTT} \times 5 + ... = 5.9011$. Where p_{XYZ} is the probability of observing the codon XYZ.

The Empirical Average Length is the length of the actual (compressed) encoded message (in bits) over the number of symbols in it. In our case, the Empirical Average Length of the genome is 1885514 / 64 = 29,461.15625.

By the Shannon theorem, the average length is bounded by the source entropy. In our case we have a memory-less stationary source so the source entropy $H(S)$ can be computed as $\sum_{i=1}^{64} P(S_i) \times \log_2 P(S_i) = 5.87$ which isn't fare from the 5.9011 bits/symbol that we could achieve with he Huffman code.

1.7 Question 7

We observe that the curve in Figure 1 is linear. Intuitively, since the Huffman algorithm without past knowledge all symbols are encoded the same, therefore the sum of the lengths of the encoded messages is the length of the sum of the encoded messages.

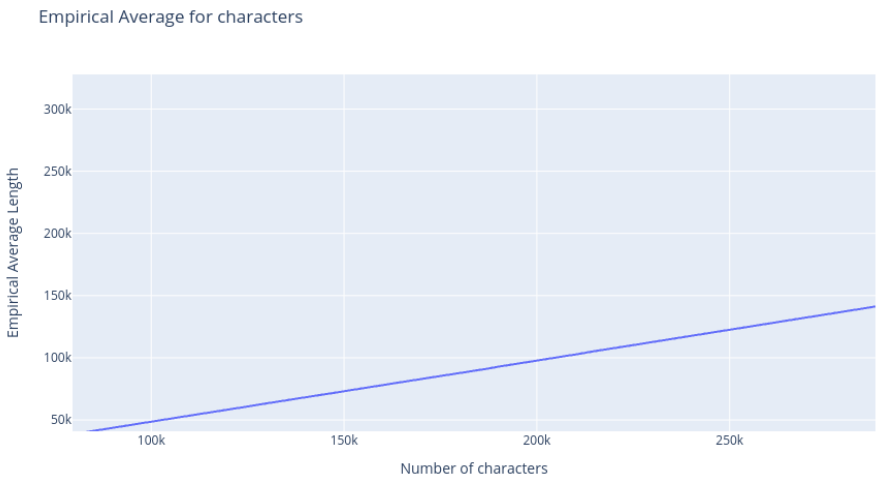


Figure 1: Huffman Increasing Length

1.8 Question 8

A first way we could improve the algorithm is to use the amino acids rather than the codon: some codons produce the same amino acid and therefore the Huffman algorithm could handle less cases. This improvement is problem dependent and since we do not know if the raw bases are import this approach may not work.

Another path for improvement is to group the codons together : by considering group of length n we have a better theoretical bound for the expected average length. In this case since amino acid groups form proteins and some proteins have a higher occurrences in nature it is very possible that some groups of codons appear more frequently than other.

1.9 Question 9

Considering that the symbol are encoded in 8 bits (since the file is encoded in UTF-8) the encoded version of the genome using the Lempel-Ziv algorithm has a total length of 2,697,854 bits. To evaluate the compression rate it is not necessary to take the size of the dictionary into account as it can be recovered easily from the encoded message. We first stripped the newline characters from the text file and counted 958,557 symbols. The encoding of the file gives a symbol rate of 8 bit per symbol so the total length of the input is 8 * 958,557 = 7,668,456bits which is a bit more than twice the length of the compressed genome. To be more precise, the compression rate is 284% while it was around 102% for the Huffman algorithm, this shows that the repetition play a big part if the redundancy of the genome.

1.10 Question 10

Let T be the genome and C(T) its coding. Then, the length of the encoded sequence is

$$l(C(T)) = TripletSize \times TripletsNumber = (3+3+8) \times 407,285 = 5,701,990.$$

The number of bits required to encode a triplet is 14 because both the distance and the length are integers that belong to the set {0, ..., WindowSize}. Since WindowSize = 7, three bits are sufficient to represent such an integer.

Thus, the compression rate of C on T is

$$\frac{l(T)}{l(C(T))} = \frac{8 \times 958,557}{5,701,990} = 1.344873632.$$

1.11 Question 11

Both algorithms are efficient to remove a specific kind of redundancy. Indeed, Huffman takes into account that a recurrent character could be represented with fewer bits. Similarly, a character that is rarely encountered could be longer without impacting the average length.

On the other hand, lZ77 provides a mechanism to reference a substring that has already been parsed. By executing sequentially these algorithms, one could obtain a better compression rate.

In our case it can be useful to consider the bases rather than the codons : there may be some repetition spanning multiple codons and not necessarily starting or ending with these codons. For instance ACG CGC GCG CAA don't have any repetition if grouped by three but taken bases by bases, CG happens 4 times in a row. Taking the bases rather than the codons improves the efficiency of the LZ77 algorithm.

1.12 Question 12

The first stage consists in executing LZ77 as before. Then, Huffman allows to compress the symbols (i.e., A, C, G, and T) embedded in the LZ77-triplets.

Let T be the genome and C(T) its coding. Then, the length of the encoded sequence is

$$l(C(T)) = TripletSize \times TripletsNumber = (3 + 3 + 2) \times 407,285 = 3,258,280.$$

Indeed, applying Huffman on top of LZ77 reduces the number of bits needed to represent the symbols.

Thus, the compression rate of C on T is

$$\frac{l(T)}{l(C(T))} = \frac{8 \times 958,557}{3,258,280} = 2.353528856.$$

The ratio between the compression rates of LZ77 and this combination is

$$\frac{2.353528856}{1.344873632} = 1.75.$$

This means that combining both algorithms is much more efficient than executing solely LZ77.

1.13 Question 13

As a reminder, the encoded message using Lempel-Ziv was 2,697,854 for a compression ratio of 2.84 .

1.14 Question 14

Ta achieve the best compression, we would first use a Huffman coding on the amino acids (rather than on the basis compounds) and use a dictionary base compression method to take care of the redundancy in the coded proteins.

While A, C, T and G are distinct codons it is important that dome combination of those serve the same purpose and it cannot be discovered from the general compression algorithm. This advantage has serious potential to increase the compression ratio.

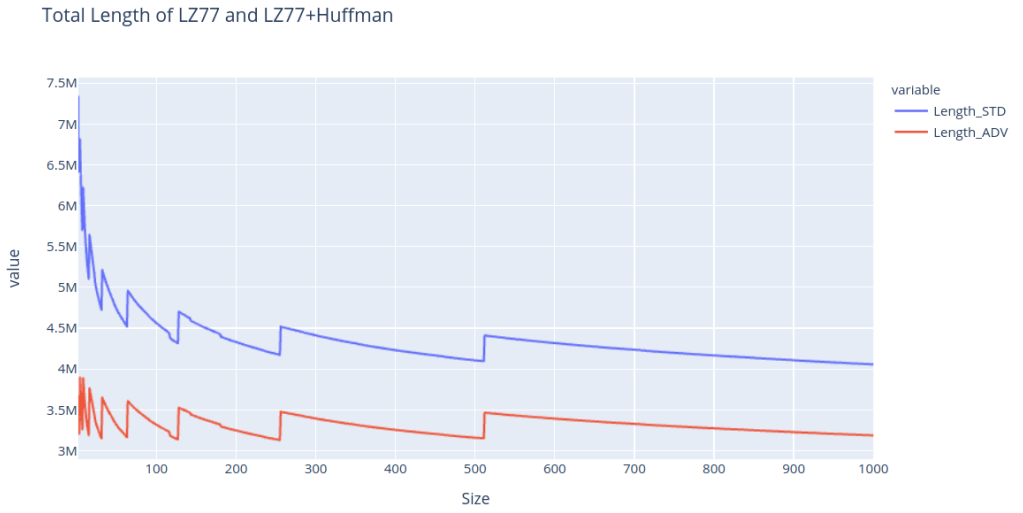
The dictionary approach can be used to take advantage of the repetition of amino acids: some proteins have similar structure and therefore these algorithm can remove these redundancies efficiently.

1.15 Question 15

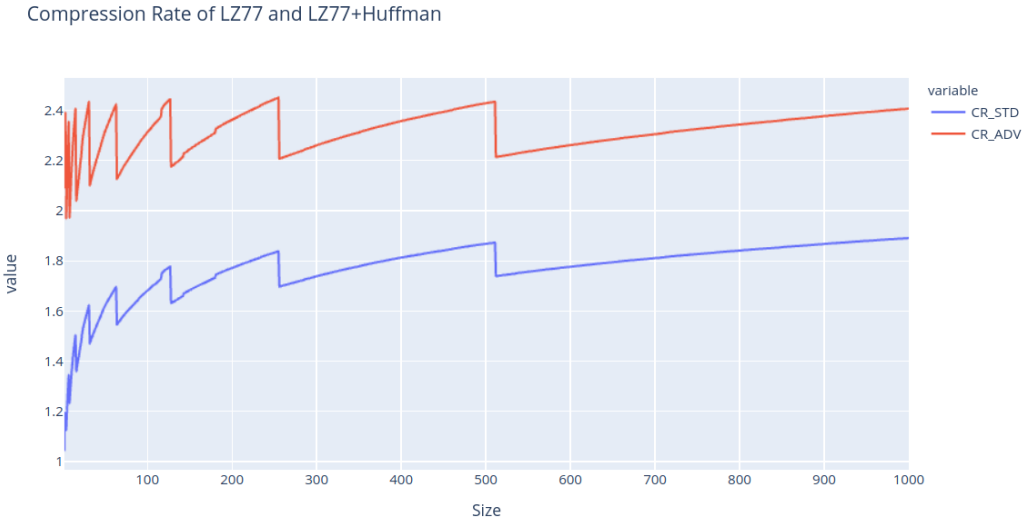
The audio file contains a quote from "Mission: Impossible 2" (2000), the sound wave is plotted in Figure 3 (top).

Size	Measure	LZ77	LZ77 + Huffman
L = 10	**Total Length**	5,740,096.00	3,587,560.00
	Compression Rate	1.335946	2.137513
L = 25	**Total Length**	4,961,394.00	3,307,596.00
	Compression Rate	1.545625	2.318438
L = 50	**Total Length**	4,731,440.00	3,312,008.00
	Compression Rate	1.620745	2.315349
L = 100	**Total Length**	4,561,040.00	3,317,120.00
	Compression Rate	1.681295	2.311781
L = 250	**Total Length**	4,185,192.00	3,138,894.00
	Compression Rate	1.832283	2.443044
L = 500	**Total Length**	4,106,050.00	3,158,500.00
	Compression Rate	1.867599	2.427879
L = 1000	**Total Length**	4,056,332.00	3,187,118.00
	Compression Rate	1.890490	2.406078

Table 2: Comparison between LZ77 and LZ77+Huffman



(a)



(b)

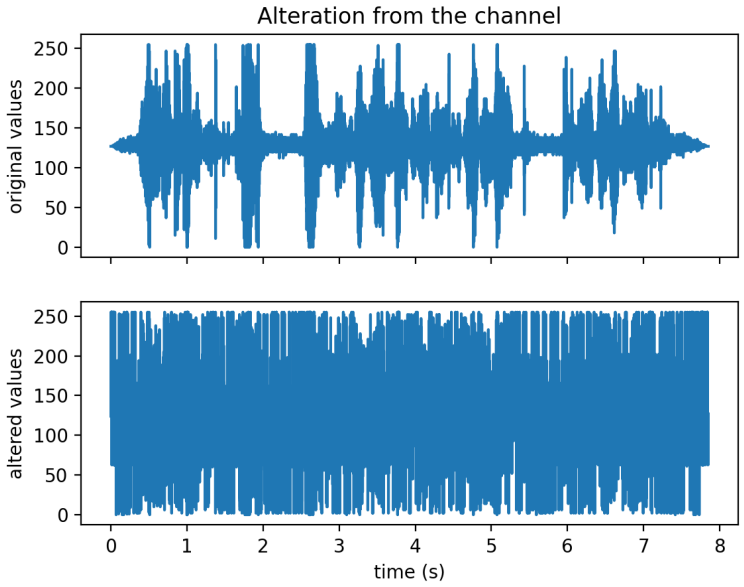


Figure 3: Alteration of the sound-wave before (top) and after (bottom) passing through a binary symmetric channel with a probability of alteration of 1%.

1.16 Question 16

The signal contains value in the range $[0, 256)$ which can be fully encoded on 8 bits ($2^8 = 256$) without wasting space (a priory, as this depends on the content of the file).

1.17 Question 17

To simulate the effect of the binary symmetric channel we create a random mask with a length equal to that of the message such that each element having a probability p of being one (all elements are drawn i.i.d.). To randomly flip the message we apply a bit-wise XOR between the mask and the message.

The sound-wave of the content after passing through the noisy channel is plotted in Figure 3 (bottom), after listening to it we can see that it is significantly deteriorated even though we can still make up what is said.

1.18 Question 18

To implement the Hamming (7, 4) code we followed the details of the theoretical lectures. The data is split by group of four bits and the three parity bits are added after each block of four for redundancy.

For simplicity we made function to encode both from a binary message or from a `uint8` message.

1.19 Question 19

After encoding the message we altered it exactly as we did before, then decoded it back to a `uint8` signal, the comparison with the original signal is shown in Figure 4. While there are still some error in the decoded wave, its shape is much closer to the original shape than without redundancy bits.

We can see that while the message is not perfectly restored, the quality of the restored signal is much better than the previous attempt with Hamming code.

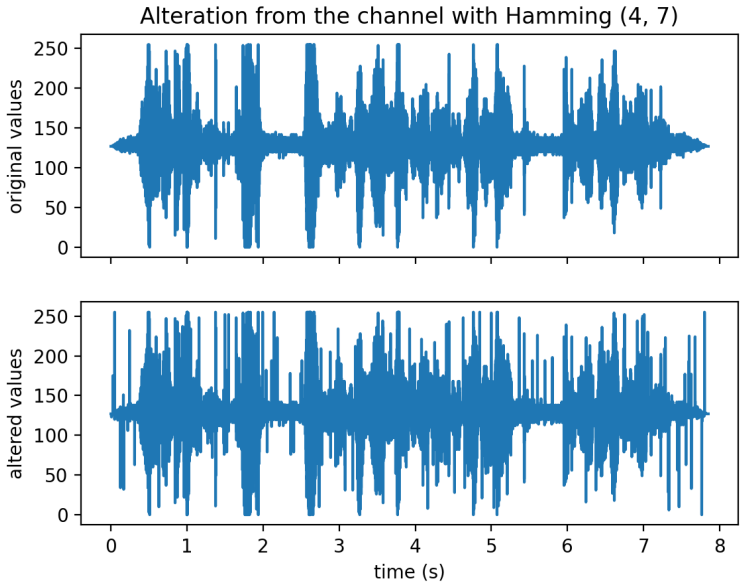


Figure 4: Alteration of the sound-wave before (top) and after (bottom) passing through a binary symmetric channel with a redundancy in the message given by a Hamming (4, 7) code.

To decode the message we try to maximize the probability $p(\hat{x}|r)$ of a message $\hat{x}$ being sent given that r was received. Assuming all code words are equally likely this is equivalent to minimizing the Hamming distance between the received bits r (with the syndrome bits) and all possibly encoded words (with their corresponding syndrome bits).

To decode the words we create two implementation, one based on lookup and one based on analysis of the syndrome bits. The first version follows the algorithm given in the theoretical lecture and flips the erroneous bits manually. For the second one, we created a table holding all 16 possible encoded values, to decoded a word we search for the closest word in the table and return this one.

We noticed that the lookup one can be a bit faster, this is probably due to the inefficiency of python control flow statement in comparison to NUMPY's vectorized instructions. In a real scenario a lookup may also be performed by dedicated hardware so this approach has some advantages.

1.20 Question 20

There are other Hamming code than the (7, 4) version, some (Hamming (3,1) which is the triple repetition) have a higher probability of detecting errors although their memory usage is much higher. Some other (255, 247) have a much more compact representation while still having a decent error detection potential.

In practice different error detection codes are used simultaneously: the ECC used on Ethernet frames aren't the same as the one in IP packets which permits a higher detection rate.

To improve efficiency we can use dedicated hardware: Hamming code have a representation where the parity bits give the position of the possible error as described here[1]. This representation can be easily implemented by printed circuits which are often faster than software implementation.

[1] `https://youtu.be/X8jsijhllIA`