LIÈGE université
School of Engineering

University of Liège
Faculty of applied science
Academic year 2024-2025

ELEN0060-2: Project 2
Source coding, data compression and channel coding

Pierre LORENZEN
S203724

Abdelilah KHALIPHI
S204896

Contents

1 Implementation **2**
1.1 Question 1 . . . 2
1.2 Question 2 . . . 2
1.3 Question 3 . . . 3
1.4 Question 4 . . . 3

2 Source coding and reversible (lossless) data compression **4**
2.1 Question 5 . . . 4
2.2 Question 6 . . . 4
2.3 Question 7 . . . 4
2.4 Question 8 . . . 5
2.5 Question 9 . . . 5
2.6 Question 10 . . . 5
2.7 Question 11 . . . 6
2.8 Question 12 . . . 6
2.9 Question 13 . . . 6
2.10 Question 14 . . . 7
2.11 Question 15 . . . 7
2.12 Question 16 . . . 8

3 Channel coding **8**
3.1 Question 17 . . . 8
3.2 Question 18 . . . 8
3.3 Question 19 . . . 9
3.4 Question 20 . . . 9
3.5 Question 21 . . . 10
3.6 Question 22 . . . 10

1 Implementation

1.1 Question 1

In the implementation for this question, we first define a node class to facilitate the creation and management of the binary tree used in the Huffman coding algorithm.

Secondly, iteratively, we sort the nodes based on their probabilities at each iteration, and we merge the two lowest probabilities. This node created by the merge of the two lowest probabilities ones is then added to the tree with a probability equal to the sum of the two lowest probabilities. The children nodes are the two nodes that were merged. The process is repeated until we have a single node left, which is the root of the tree.

Tirdly, we generate the codes for each symbol by traversing the tree. We start at the root and assign a '0' code to the left child and a '1' code to the right child. We continue this process recursively until we reach a leaf node, at which point we store the generated code for that symbol.

Finally, we reorder the symbols to be consistent with the order provided at the input of the function.

To extend the Huffman code generation to any alphabet size q, we can modify the algorithm to merge the q lowest-probabilities nodes at each step and so build a q-ary tree, where each node has up to q children.

To implement this with a number of input symbols n and output alphabet size of $q \geq 2$, we can

1. Take the n symbols with their probabilities
2. Add Dummy Symbols: If $(n-1)\ mod\ (q-1) \neq 0$, add dummy symbols of probability 0 so that: $(n'-1)\ mod\ (q-1) = 0$

   where n' is the total number of symbols (real + dummy).
3. Maintain nodes sorted by probability.
4. Merge q smallest-nodes.
5. Repeat until only one node is left.
6. Label the edges with 0,1,...,q-1.

1.2 Question 2

Initially, the dictionary is established with an empty string mapped to index 0. The algorithm then iterates through each character of the input sequence, progressively building phrases. For each iteration, it combines the current phrase with the next character from the input to form a new candidate phrase. If this new phrase is not already present in the dictionary, the algorithm performs two operations:

- It appends a tuple containing the index of the current phrase (as found in the dictionary) and the character that triggered the creation of the new phrase to the encoded_sequence.
- It then adds the new phrase to the dictionary with a unique incremental index.

The current phrase is reset after each dictionary insertion, allowing the process to restart and construct new phrases from subsequent characters. If the new phrase is already present in the dictionary, the algorithm continues extending it until it encounters a unique phrase.

After processing the entire sequence, any remaining unprocessed phrase is handled explicitly by adding a final tuple to the encoded sequence.

Applying this algorithm to the given example sequence "ababcbababaaaaaaa", results in constructing a dictionary that maps substrings to indices, and produces an encoded sequence consisting of tuples representing dictionary indices paired with subsequent characters.

This approach effectively exploits repeating patterns within data, significantly reducing the size required for storage or transmission, making it a practical and efficient method in data compression scenarios.

1.3 Question 3

As explained in the theoretical course, the Lempel-Ziv algorithms can be compared on several aspects. These aspects are the dictionary construction, the parsing of the input data, the encoding, the dictionary address size, the adaptivity of the algorithm, the efficiency, and the on-line capability. See table 1 for a comparison of the two algorithms.

Aspect	Basic Lempel-Ziv	On-line Lempel-Ziv
Dictionary construction	Grows incrementally by adding new words	Same as basic
Parsing	Greedy: find the longest prefix in dictionary	Same as basic
Encoding	Tuple (address of prefix, next symbol)	Same as basic but address is encoded dynamically
Dictionary address size	Fixed-size = 2^n (where n is number of bits)	Variable-length based on current dictionary size
Adaptivity	No adaptation based on dictionary	Fewer bits used at early stages
Efficiency	Good compression with large dictionary	More efficient in early stages due to shorter addresses
On-line capability	No	Yes, can transmit as text is read

Table 1: Comparison of Basic Lempel-Ziv and On-line Lempel-Ziv

Base on this comparison, we can conclude that the advantages of the basic Lempel-Ziv algorithm are that the compression on large dictionary is more efficient than the on-line Lempel-Ziv algorithm, and that the dictionary address size is fixed. The backwards of this version is that it is not adaptive, and it is not on-line.

The advantages of the on-line Lempel-Ziv algorithm are that it is adaptive, on small and medium dictionary size the efficiency is better due to the shorter addresses, and it is on-line. The backwards of this version is that the compression on large dictionary is less efficient than the basic Lempel-Ziv algorithm, and that the dictionary address size is variable.

1.4 Question 4

To decode a LZ77 encoded text, we need first to reset the basis. The encoded text is a sequence of triples (Offset, Length, NextChar). To decode a full sequence of triple, the same logic is applied to each triple in the sequence.

Firstly, **if the offset is 0 and the length is 0**, then the next character can be appended to the output string.

Secondly, **if the offset is not 0 and the length is not 0**, then we need to move back by the specified offset in the output string and copy the specified length of characters from the output string to the output string. After that we can add the next character to the output string.

This process is repeated until we have decoded the entire sequence of triples.

Example:
If the encoded text is (0, 0, ’a’), (0, 0, ’b’), (0, 0, ’r’), (3, 1, ’c’), (5, 1, ’d’), (7, 4, ’d’). The decoding algorithm will run like seen in the table 2:

Triple	Current output string	Afterwards output string
(0, 0, 'a')	" "	"a"
(0, 0, 'b')	"a"	"ab"
(0, 0, 'r')	"ab"	"abr"
(3, 1, 'c')	"abr"	"abrac"
(5, 1, 'd')	"abrac"	"abracad"
(7, 4, 'd')	"abracad"	"abracadabrad"

Table 2: Decoding of the encoded text

2 Source coding and reversible (lossless) data compression

2.1 Question 5

After having estimated the marginal probabilities of all the 27 symbols, determine the Huffman codes for each symbol, and encoded the English text. The total length of the encoded text is 239008 bits and the compression rate is 1.9496585888338465.

2.2 Question 6

The average length for our Huffman code is 4.10328 bits.

Let's compare this value with the empirical average length. We can see that the empirical average length is 4.10328. So,

$$empirical_{avg} = 4.10328 = expected_{avg}$$

The theoretical bounds of a Huffman code are related to how closely its average codeword length approximates the entropy of the source.

$$\frac{H(S)}{\log\ q} \leq \overline{n} < \frac{H(S)}{\log\ q} + 1$$

This means that the probabilities used to build the Huffman code match exactly the observed frequencies in the text. That was expected because the Huffman code is built based on the frequencies of the symbols in the text. This means that our algorithm works well.
By knowing that the entropy in this case is

$$H(S) = 4.06223$$

and with $q = 2$ because the size of the alphabet is 2 (the Huffman tree is a binary tree), we can calculate the bounds of the average codeword length:

$$\frac{4.06223}{\log 2} \leq \overline{n} < \frac{4.06223}{\log 2} + 1$$
$$4.06223 \leq \overline{n} < 5.06223$$

We can see that the expected average length is 4.10328 which is between the bounds 4.06223 and 5.06223. The value of the expected average length tells us that the Huffman code is near-optimal.

2.3 Question 7

The empirical average code length evolution with text length is shown in the figure 1.

We can see that the empirical average code length is decreasing with the text length. This means that the Huffman code is getting better as the text length is increasing. We can also see that the empirical average code length is growing exponentially until we reach the 10,000 characters. This demonstrates that the Huffman code is not optimal for short texts.

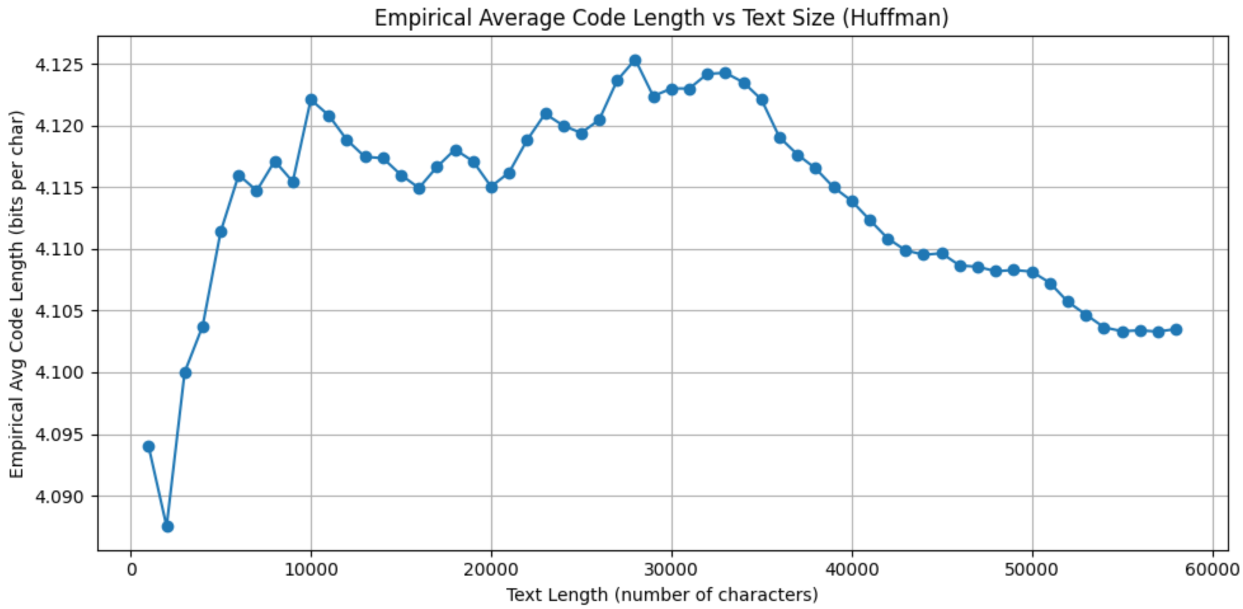

Figure 1: Empirical average code length evolution with text length

2.4 Question 8

The total encoded text length is 233396 bits and the compression rate is 1.9965380726319217 using the on-line Lempel-Ziv algorithm.

2.5 Question 9

The total encoded text length is 441364 bits and the compression rate is 1.0557816224250278 using the LZ77 algorithm with a window size of 7 characters.

2.6 Question 10

One straightforward method for combining LZ77 and Huffman coding involves a two-stage process. First, the input text is compressed using the LZ77 algorithm, which outputs a sequence of triplets in the form (offset, length, next_char). In the second stage, Huffman coding is applied exclusively to the next_char symbols extracted from these triplets. This approach is relatively simple to implement and captures the redundancy in literal symbols effectively. However, it does not compress the metadata components (offset and length), which are typically stored using a fixed number of bits. As a result, the overall compression performance may be suboptimal, especially in cases where the metadata forms a significant portion of the encoded data.

A more sophisticated and efficient technique involves applying Huffman coding separately to each component of the LZ77 triplets (the offset, length, and next_char). Each component is treated as a distinct symbol stream, and frequency-based Huffman coding is used to encode each stream independently. This method leverages the statistical properties of all parts of the compressed output, resulting in significantly improved compression ratios, particularly for highly repetitive input data where certain offsets and lengths occur more frequently.

This strategy is employed in real-world compression standards such as DEFLATE (used in formats like ZIP and PNG), where additional optimization techniques such as dynamic block-based Huffman trees are also incorporated. While this approach yields better compression, it comes at the cost of increased computational complexity and memory usage, as separate Huffman trees must be built, stored, and transmitted for each component of the LZ77 output.

2.7 Question 11

In our implementation, we choose the first proposition to combine the LZ77 and Huffman coding techniques. The LZ77 algorithm was first used to parse the input text into a sequence of triplets, each of the form (offset, length, next_char), where *offset* and *length* refer to a match in a sliding window of size 7. Given the maximum window size, both the offset and length components were stored using fixed-length 3-bit binary values, which are sufficient to represent values up to 7.

To enhance compression efficiency, we applied Huffman coding to the next_char component of each triplet. A Huffman codebook was constructed based on the empirical frequency distribution of these literal characters across the entire LZ77 output. This allowed us to exploit not only the redundancy in repeated substrings (via LZ77), but also the unequal probabilities of individual characters (via Huffman coding), resulting in a more compact representation of the data.

The total number of bits required for encoding was calculated as the sum of:

- The fixed-size metadata for each triplet (6 bits per triplet: 3 bits for offset, 3 bits for length),
- The Huffman-encoded bits corresponding to each next_char.

The resulting compression rate, measured as the ratio of the original uncompressed size (in bits) to the total compressed size, was approximately 1.122731270103242. This indicates a meaningful improvement over using LZ77 alone, which does not compress the literal characters, and confirms the benefit of combining dictionary-based and statistical coding methods.

- Total encoded text length: 415045 bits
- Compression rate: 1.122731270103242

2.8 Question 12

For the given English text, the LZ78 algorithm produced a total encoded size of **233,396 bits**, resulting in a **compression rate** of approximately **1.997**. This serves as a baseline for evaluating the more advanced hybrid techniques.

By contrast, the LZ77 algorithm alone, using a sliding window of 10,001 characters, achieved a much smaller encoded size of **101,651 bits**, corresponding to a **compression rate** of **4.584** (as shown in Table 3). Furthermore, when LZ77 was combined with Huffman coding, the performance improved even further. This hybrid method resulted in a total encoded length of **96,308 bits** and a **compression rate** of **4.839** (see Table 4).

These results highlight a substantial improvement in compression when employing a two-stage approach that first eliminates redundancy via LZ77 and then compresses the remaining symbols based on their frequency. Compared to LZ78, the hybrid LZ77-Huffman method reduced the total bit length by over 58% and more than doubled the compression rate.

In summary, while the LZ78 algorithm is advantageous for its simplicity and real-time capabilities, it is significantly outperformed by LZ77-based methods, particularly when augmented with Huffman coding, in terms of both compression efficiency and encoded size. This demonstrates the value of combining dictionary-based and statistical compression techniques to better exploit the structure and redundancy present in natural language text.

2.9 Question 13

Based on the results obtained in Question 12, the best-performing compression algorithm was the combination of LZ77 with Huffman coding applied to the next_char field. This method effectively reduces redundancy at two levels: structural repetition through LZ77, and symbol frequency through Huffman coding. As the sliding window size increases, LZ77 is able to capture more long-distance repetitions, significantly improving compression rates. In contrast, LZ78 performs reasonably well, but does not match the efficiency of LZ77

Window size	Total length	Compression rate
1	628,859	0.7410
1,001	145,541	3.2017
2,001	131,120	3.5539
3,001	123,442	3.7749
4,001	118,492	3.9326
5,001	115,379	4.0387
6,001	112,288	4.1499
7,001	109,978	4.2371
8,001	104,027	4.4795
9,001	102,806	4.5327
10,001	101,651	4.5842

Table 3: Total length and compression rates using the LZ77 with different sliding windows

with large windows, as it lacks a mechanism to look back over long ranges. This confirms that the most effective data compression approach in this context is one that combines dictionary-based matching with entropy coding, particularly when applied to large, repetitive datasets.

2.10 Question 14

The results of the encoded encrypted text using the binary Huffman algorithm and the LZ77 algorithm can be found in the table 5.

We can see that the binary Huffman algorithm is nearly twice better than the LZ77 algorithm.

2.11 Question 15

The compression rates obtained on the encrypted text using the **LZ77** and **binary Huffman** algorithms were **0.923** and **1.698**, respectively (see Table 5). In contrast, substantially higher compression rates were achieved when the same algorithms were applied to unencrypted English text. Specifically, in Question 5, the binary Huffman algorithm achieved a compression rate of **1.950**, while in Question 9, LZ77 achieved a rate of **1.056**.

This difference can be attributed to the statistical properties of encrypted versus natural language data. Strong encryption algorithms are designed to maximize the entropy of the output, making the ciphertext statistically resemble random noise. In such data, characters are distributed nearly uniformly, and structural patterns are intentionally removed. This directly undermines the assumptions that dictionary-based (LZ77) and statistical (Huffman) compression algorithms rely on the presence of repeated substrings and non-uniform symbol frequencies.

The **encryption key** plays a crucial role in shaping the compressibility of the encrypted output. Several intuitive factors are worth noting:

- **Key size:** Larger keys increase the entropy of the encryption process, further randomizing the output and eliminating predictable patterns.
- **Key complexity:** Keys containing a broader variety of characters and a less predictable structure (e.g., random keys versus fixed templates) result in ciphertexts with more uniform character distributions, making Huffman coding less effective.

Window size	Total length	Compression rate
1	577,270	0.8072
1,001	138,265	3.3702
2,001	124,451	3.7443
3,001	117,101	3.9793
4,001	112,377	4.1466
5,001	109,376	4.2604
6,001	106,419	4.3788
7,001	104,243	4.4702
8,001	98,641	4.7240
9,001	97,449	4.7818
10,001	96,308	4.8385

Table 4: Total length and compression rates using the LZ77 and Huffman with different sliding windows

Algorithm	LZ77	Binary Huffman algorithm
Compression rate	0.9231848196260775	1.6980129651020848

Table 5: Compression rate using the binary Huffman algorithm and the LZ77 algorithm

- **Encryption strength:** As the encryption method becomes more secure (e.g., adopting stronger substitution-permutation structures), even small statistical hints are suppressed, leaving virtually no redundancy for compression algorithms to exploit.

2.12 Question 16

If Alice wants to send the more compressed private message to Bob, she should compress it before the encryption is made. The encryption will suppress the redundancy in the message, making it less compressible.

3 Channel coding

3.1 Question 17

Here is the image generated (see figure 2):

3.2 Question 18

In order to encode a grayscale image using a fixed-length binary code, we must determine the minimum number of bits required to represent all possible pixel intensity values.

A grayscale image typically represent pixel intensities as integer values ranging from:

$$\text{Minimum } 0 \text{ to Maximum } 255$$

So there is 256 distinct values.

To represent 256 distinct values in binary, we must choose the smallest integer b such that:

$$2^b \geq 256$$

Figure 2: Image generated

Solving this gives:

$$b = \lceil \log_2(256) \rceil = \lceil 8 \rceil = 8 \text{ bits}$$

This matches the required range [0,255] perfectly. Any representation using fewer than 8 bits (e.g., 7 bits $\rightarrow$ 128 values) would be insufficient to encode the full grayscale spectrum, leading to a loss of information and reduced image quality.

3.3 Question 19

In our simulation, we modeled the transmission of a grayscale image through a noisy channel using a Binary Symmetric Channel (BSC) with a crossover probability $p = 0.01$. This means that each bit of the transmitted image has a 1% chance of being flipped (from 0 to 1 or vice versa) independently of the others.

The resulting image (see figure 3), after transmission through the BSC, exhibits noticeable noise artifacts:

Figure 3: Image after transmission through the BSC

The noise introduced by the channel manifests as random speckling and abrupt variations in pixel intensity across the image. As a result, edges become less defined and fine structural details are partially lost or distorted. Overall, the image exhibits a noticeable reduction in clarity and visual fidelity when compared to the original.

3.4 Question 20

The resulting image after adding some redundancy and passing through the BSC is shown in figure 4.

Figure 4: Image after adding some redundancy and passing through the BSC

The noise noticed in this image is less visible than for the previous image. This improvement is due to the Hamming code's ability to identify and correct single-bit errors, which are the most likely type of corruption under the given channel conditions. Although this approach increases the data size by a factor of $\frac{7}{4}$, the trade-off is justified by the substantial gain in visual quality and robustness.

3.5 Question 21

3.6 Question 22