Immagine che contiene testo, emblema, simbolo

Descrizione generata automaticamente

**Homework 4 – Lossless Compression**

Mobile Programming and Multimedia

Gabriel Rovesti – 2103389

Table of Contents

[2 Assignment 3](#_Toc166622315)

[3 LZW Algorithm 3](#_Toc166622316)

[4 Huffman Algorithm 5](#_Toc166622317)

[5 Conclusions 6](#_Toc166622318)

# Assignment

*Encode the string:*

*abcabcabcabcffffffffffff000000000000ffffffffffffffffffffffff*

*using:*

1. *the LZW algorithm and*
2. *choose an algorithm between Shannon-Fano and Huffman.*

*Compare the two results in terms of compression ratio.*

# LZW Algorithm

Let’s first start considering the LZW algorithm on the string given in input. Here, the table is created to give an overview representation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Output | Code | Symbol |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

The algorithm is applied step by step based on the algorithm code taken from the [slides](https://stem.elearning.unipd.it/pluginfile.php/803212/mod_resource/content/3/mmdata.pdf). Basically, we encode each character starting from the ASCII table (0-255) and see if the word exists inside the dictionary; if not, the concatenation is then added inside of the dictionary and the code for the specific word is given in output, then adding the code for the considered string. This way, the vocabulary is dynamically built, encoding the variable-length strings each time.

So, the encoded sequence is:

The original size is given by the number of bits of the whole original encoding multiplied by the number of bits given the representation, so .

We are using two bytes, given the number of bits occupied is , so we have the number of characters occupied by the encoding (21). Given the encoding is multiplied by the number of bytes occupied (), we would have for the total encoded size.

Quoting the formula of data compression ratio present [here](https://en.wikipedia.org/wiki/Data_compression_ratio#:~:text=Data%20compression%20ratio%2C%20also%20known,uncompressed%20size%20by%20compressed%20size.):

# Huffman Algorithm

In this section, the Huffman algorithm is chosen and applied, with the following table describing each symbol, occurrences and the encoding, obtained looking at the tree obtained ( for left children, for the right children), hence considering the total number of those.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Symbol* | *N. of occurrences (#)* |  | *Code* | *N. of bits* |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Immagine che contiene calligrafia, disegno, Carattere

Descrizione generata automaticamenteThe corresponding tree is represented here representing the encoding is shown here:

The algorithm is bottom-up, so we start from the lowest-occurrences nodes, in this case , forming a new node as sum. Given the tree would be unbalanced, the character is then summed subsequently, forming a sum node of . Continuing this way, we sum all occurrences of the nodes, reaching the root of .

The algorithm gives in output as encoded string:

* Original: abcabcabcabcffffffffffff000000000000ffffffffffffffffffffffff
* Encoded: 1110 1111 110 1110 1111 110 1110 1111 110 1110 1111 11 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

We then compute how many bits are occupied, considering this is computed multiplying the number of occurrences with how many bits the single code occupies:

The occupation is given from the number of bits needed by the ratio (8 bits) multiplied by the different chars found inside the encoding () multiplying by 2 to represent the column of the encoded sequence (as much as the number of symbols). Combining all of this we get . This is then summed with the result of the encoding, specifically for the encoded size.

To calculate the effect of the entropy, we use the formula , obtaining as theoretical occupation multiplied by the number of characters (60), .

The original uncompressed ratio is given by the total number of original bits multiplied by bits, so (so, number of bits occupied multiplied by 8 bits).

Quoting the formula of data compression ratio present [here](https://en.wikipedia.org/wiki/Data_compression_ratio#:~:text=Data%20compression%20ratio%2C%20also%20known,uncompressed%20size%20by%20compressed%20size.):

# Conclusions

The LZW algorithm achieved a compression ratio of , compressing the original data from bits to bits.

The Huffman algorithm performed better, achieving a compression ratio of , compressing the original data from bits to bits (plus an additional bits for the encoding table). So, in this specific case, it provided significantly better compression.

This is due to the Huffman algorithm's ability to assign shorter codes to more frequent symbols, effectively exploiting the skewed symbol distribution in the input data. This is based on construction of a code tree based on symbol frequencies, which can be computationally expensive for large input data.

LZW started becoming more efficient after more appearances of the same characters and their combinations. This, on the other hand, does not require any prior knowledge of symbol frequencies and can adapt to the input data dynamically, making it more suitable for scenarios where the input data is not known in advance or has varying symbol distributions.

As a matter of fact, recall LZW does not need to memorize the table, reducing the overhead needed in saving it but building it dynamically during the decompression phase. So, in the end, Huffman always obtains the optimum in terms of compression for each string length, while LZW needs data of at least 100 kb to obtain efficient results comparable to Huffman's.