# Data Compression

Ionescu Matei-Ștefan

Faculty of Automatic Control and Computers
University Politehnica of Bucharest

matei.ionescu7@gmail.com

**Abstract.** Compression is the process of reducing the size of data in order to save space or transmission time. This paper aims to compare two popular lossless data compression algorithms, the Lempel-Ziv-Welch algorithm, and the Huffman coding algorithm.

**Keywords:** Data compression, Huffman coding, Lempel-Ziv-Welch, LZW.

## 1 Introduction

### 1.1 Problem description

Data compression represents the process of reducing the size of a computer file by encoding the information using fewer bits than the original representation. [1] There are two types of data compression algorithms. Lossless compression algorithm ensure that no data is lost during the process and the file will expand to its original size when decomposed, while lossy compression algorithms reduce the size of the file even further by removing unnecessary or less important information. [2] “Lossless algorithms are typically used for text or executable codes, while lossy are used for images and audio where a little bit of loss in resolutions is often undetectable, or at least acceptable. Lossy is used in an abstract sense; it does not mean random loss in samples, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise.” [1]

### 1.2 Real life utility

Data compression is a very important topic in the Computer Science field. The main advantages of compression are reduced times for data transfers, saving data storage space and consuming less network bandwidth. Thus, data compression increases cost efficiency and productivity.

“Compression is built into a wide range of technologies, including storage systems, databases, OSes and software applications used by businesses and enterprise organizations. Compressing data is also common in consumer devices, such as laptops, PCs and mobile phones.” [3]

Some of the most popular file compression utility software are: WinRAR, WinZip, 7-Zip, Zip Archiver, MacRAR, MacZip and Gzip.

### 1.3 Chosen solution

There is no compression algorithm that can efficiently compress all types of data. Thus, there are many different lossless compression algorithms. Some of the more used ones are: ANS, Arithmetic coding, Burrows-Wheeler transform, Huffman coding, Lempel-Ziv compression, Prediction by partial matching (PPM), Run-length encoding (RLE).

In this paper, I am comparing the Lempel-Ziv-Welch algorithm and the Huffman coding algorithm, which are two of the most popular lossless data compression algorithms.

### 1.4 Solution evaluation criteria

The first part of the evaluation process focuses on the compression ratio that each of the algorithms achieves, by comparing the size of the file before and after being compressed.

The second part of the evaluation process focuses on the efficiency of the program while compressing and decompressing the data. The metrics used for this analysis consist of the time and memory that the process utilizes.

## 2 Solutions

### 2.1 Lempel-Ziv-Welch

Lempel–Ziv–Welch (LZW) is a “greedy" universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978 and had become the first widely used universal data compression method on computers. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. [4] It is the algorithm of the Unix file compression utility ‘compress’ and is used in the GIF image format.

#### Explaining the algorithm

The high-level model for the LZW algorithm works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes. The codes take up less space than the strings they replace, thus the data is compressed. [5]

*Encoding.* In my implementation, I am improving the basic algorithm by using a dictionary that can store 65535 different strings. Allowing the dictionary to store more codes should, theoretically, result in a better compression ratio, at the expanse of more memory utilized. In the program, the dictionary is implemented using the C++ ‘map’ and associates the string with its code. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through

65535 are created in the dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence with no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary. [4]

*Decoding.* The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the dictionary. However, the full dictionary is not needed, only the initial dictionary that contains single-character strings (and that is usually hard coded in the program, instead of sent with the encoded data). Instead, the full dictionary is rebuilt during the decoding process the following way: after decoding a value and outputting a string, the decoder concatenates it with the first character of the next decoded string (or the first character of current string, if the next one can't be decoded; since if the next value is unknown, then it must be the value added to the dictionary in this iteration, and so its first character is the same as the first character of the current string), and updates the dictionary with the new string. The decoder then proceeds to the next input (which was already read in the previous iteration) and processes it as before, and so on until it has exhausted the input stream. [4]

*Dictionary reset.* To ensure that the data is compressed at a decent ratio throughout the whole file, when the dictionary is filled, it will reset, leaving only the codes from 0 to 255. By doing this, good compression can be achieved, even if data differs greatly from one part of the file to another.

**Complexity analysis**

*Compression.* The LZW algorithm compresses the file by reading one byte at a time from the input file, adding it to the current string, checking if the string is already in the dictionary and adding the string to it if this is not the case. Also, the dictionary resets when the dictionary becomes full. So, to find the complexity of the compression algorithm we need to find the complexity of each operation.

- Reading the bytes from the input file has $O(m)$ complexity where $m$ is the number of bytes in the file.
- For checking if the string is already in the dictionary the `map::count` function is called. The function's time complexity is logarithmic ($O(\log(n))$ where $n$ is the number of elements in the map). [6]
- For adding a new element in the map structure, the insertion time is $O(\log(n))$ where $m$ is the number of elements in the map. [7]
- Depending on the input file the dictionary might reset, action which is performed in $O(r)$ where $r$ is number of entries in the dictionary (in our case 65535).

The average overall complexity of the compression is $O(m \cdot \log(n))$ where $m$ is the number of bytes in the file and $n$ is the number of elements in the map, because for each byte read from the file a search for the current string will be performed. The time complexity of the algorithm can be improved by using an `unordered_map` which

would reduce the time complexity to $O(m)$ where $m$ is the number of bytes from the input file, because of its improved search and insertion times.

*Decompression.* The decompression algorithm works using the `map::at` function (logarithmic complexity [8]) to update the current string after reading each 16-bit sequence from the input file and adding it to the dictionary when required.

Therefore, the average overall time complexity for the decompression algorithm is $O(m \cdot \log(n))$ where $m$ is the number of 16-bit sequences read from the file and $n$ the number of elements in the dictionary. Like the compression algorithm, the decompression would benefit from using an `unordered_map` which would reduce the time complexity to $O(m)$ on the average case.

### Advantages and disadvantages

The main advantages of the LZW algorithm are:

- not needing prior information about the input data stream
- compressing the input stream in one single pass
- the simplicity that allows fast execution.

The main disadvantages of the LZW algorithm are:

- not being able to considerably compress files that do not contain repetitive data
- the amount of memory needed is indeterminate as it depends on the total length of all the strings
- higher memory usage than other algorithms.

## 2.2 Huffman coding

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes". [9] The Huffman coding algorithm is used by conventional compression formats like PKZIP, GZIP and Multimedia codecs like JPEG, PNG, and MP3. [10]

### Explaining the algorithm

The Huffman coding algorithm is a greedy algorithm that assigns variable-length codes to input characters. The lengths of the assigned codes are based on the frequencies of the corresponding characters. The most frequent character gets the smallest code, and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character

is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

*Encoding.* The first step in encoding the data from the input file is finding the frequency of each character in the file (which can be viewed as a 8-bit sequence for non-text files). Next, the Huffman tree is build using the following rules:

1. "Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete." [10]

After it is created, the Huffman tree is traversed, and new codes are assigned for each character. Finally, the frequency of each character is written in the output file, and then the encoded text.

*Decoding.* The first step in decompressing the file is reading the frequency of each character and constructing the Huffman tree the same way we did for encoding. Then, the compressed input file is decoded by replacing the code resulted from the Huffman tree with its original 8-bit sequence.

**Complexity analysis**

*Compression.* There are four actions done when compressing a file using the Huffman coding algorithm:

1. Finding the frequency of each character. This is done by reading each byte from the input file and updating its number of appearances. The time complexity of this operation is $O(r)$ where $r$ is the number of bytes in the input file.
2. Building the Huffman tree. The time complexity of the Huffman algorithm is $O(n \cdot log(n))$ where $n$ is the number of unique characters. Using a heap to store the weight of each tree, each iteration requires $O(log(n))$ time to determine the cheapest weight and insert the new weight. There are $O(n)$ iterations, one for each item.
3. Filling the codebook which stores the new code of each character. This is done by traversing the Huffman binary tree. The recurrence formula is:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + const$$

Where $T(n)$ is the number of operations executed by traversing the tree and $const$ a constant number, equal to the number of constant time operations in the function. The $2 * T\left(\frac{n}{2}\right)$ represents the sum of the each $T\left(\frac{n}{2}\right)$ operation of the left node and the right

node. Using the Masters' Theorem, $T(n) = \text{a} * T\left(\frac{n}{\text{b}}\right) + f(n)$. So $a = 2, b = 2, f(n) = constant \rightarrow c = 0$. From where the complexity of this operation is $O(n)$, where $n$ is the number of nodes in the binary tree.

4. Writing the data to the output file. The time complexity of this action is $O(r)$ where $r$ is the number of bytes from the input file.

Taking all into consideration, the overall complexity of the compression is $O(r + n * \log(n))$. where $n$ is the number of unique characters and $r$ the number of bytes in the input stream.

*Decompression.* The decompression can be split into four stages:

1. Retrieving the frequency of each character from the input file. The complexity of this action is $O(n)$ where $n$ is the number of possible characters (256).
2. Building the Huffman tree. The time complexity is the same for both compression and decompression, $O(n \cdot log(n))$ where $n$ is the number of unique characters.
3. Filling the codebook which stores the new code of each character. This action has the same complexity explained in the compression paragraph, $O(n)$, where $n$ is the number of nodes in the binary tree.
4. Writing the data to the output file. The time complexity of this action is $O(r * n)$ where $r$ is the number of bytes from the input file and $n$ is the number of unique characters, because for each byte read, the codebook will be checked to see if a code is found.

The overall complexity of the decompression operation is $O(n * (r + \log(n)))$ where $r$ is the number of bytes from the input file and $n$ is the number of unique characters.

**Advantages and disadvantages**

The main advantages of the Huffman coding algorithm are:

- being able to better compress high entropy files (files containing data that does not tend to repeat) better because it stores frequent characters using less data, and rare ones using more. Therefore, it is focusing on compressing the size of characters not strings.
- Average compression ratio of 45%-50%
- Pairs well with other compression algorithms.

The main disadvantages of the Huffman coding algorithm are:

- requires two passes over the input (one for computing the frequencies and one for encoding the data), thus encoding might be slower
- needs to store the frequencies of the characters in the encoded file to be able to de-code it.

# 3 Evaluation

## 3.1 Testing set

The testing set is composed of 60 tests:

- Tests 1-10: contain text from Wikipedia articles about different countries in the English language. These should test the basic usage in some day-to-day utilization conditions of each algorithm.
- Tests 11-20: are text files, increasing in size, in Latin, generated using an Online Lorem Ipsum generator. These should be favorable tests for the algorithm because their lower entropy (tendency to have repeating data).
- Tests 21-24 and 26: are text files, increasing in size, in English, that contain lines of characters from the Breaking Bad TV Show.
- Test 25: is a random generated text file, that contains English words related to crypto currencies and the blockchain.
- Test 27: is a text file containing the story “Adventures of Huckleberry Finn” by Mark Twain.
- Test 28: is a random generated text file, in English, using the DeLorean Ipsum Text Generator.
- Tests 29-30: are random generated text file, using the Sagan Ipsum generator, that produces text about the universe, in English.
- Tests 31-40: are text files, increasing in size, that contain random bytes of data. This are files that are very hard to compress due to their high entropy. These tests were created using the OnlineFileTools website.
- Tests 41-50: are images in the bmp format, of different sizes.
- Tests 51-53: jpg images.
- Tests 54-56: mp3 files.
- Tests 57-60: pdf files.

The text files are the focus of the evaluation process. It is also expected that the bmp files will have a decent compression ratio since bmp is an uncompressed format. The jpeg, pdf, and mp3 formats are already compressed and I expect bad results in compression ratio. They are included to test if the algorithms will increase or decrease the size of already compressed files.

## 3.2 System specifications

The laptop the code was tested runs on an i5-1135G7 processor with 4 cores and 8 logical processors, has 16 GB of LPDDR4X RAM 4266 MHz and a 1TB SSD.

The code has been compiled using g++ compiler with -O3 optimizations level and ran on an Ubuntu 20.04 VirtualBox virtual machine running on 3 cores of the processor with 6144MB RAM and 200GB SSD.

### 3.3 Results

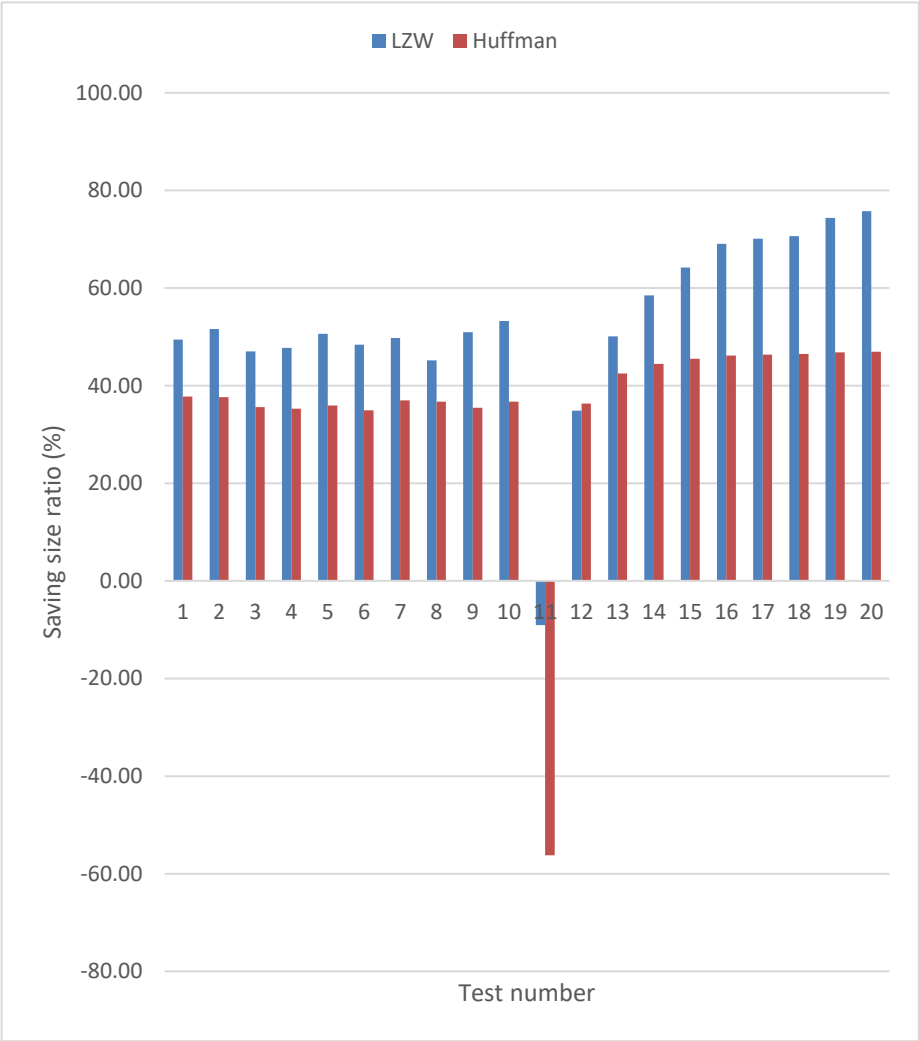
**Compression ratio**

In the first part of the evaluation, I am focusing on the space saving capabilities of each algorithm. I will be using the following formula to calculate the metric used for comparison:

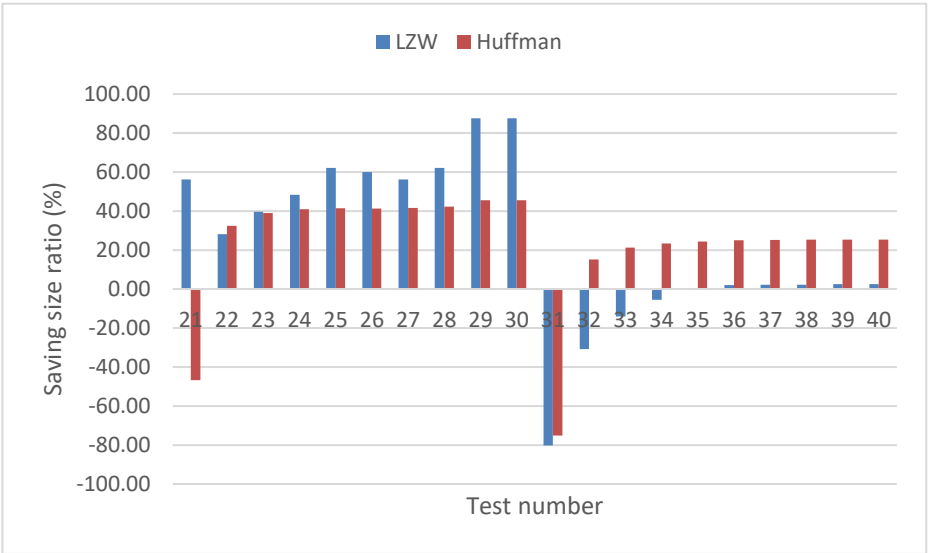
$$Space\ Saving\ Ratio\ (\%) = 1 - \frac{Compressed\ Size}{Uncompressed\ Size}$$

*Results illustration*

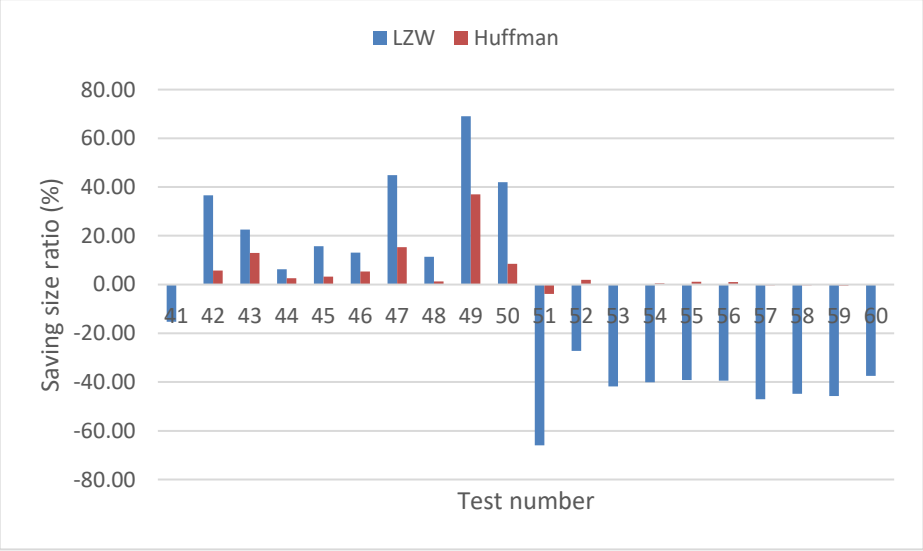
| Test Nr. | Uncompressed data size (bytes) | LZW | | Huffman | |
|---|---|---|---|---|---|
| | | Compressed size (bytes) | Space saving ratio (%) | Compressed size (bytes) | Space saving ratio (%) |
| test01 | 168,289 | 85,094 | 49.44 | 104,744 | 37.76 |
| test02 | 228,965 | 110,756 | 51.63 | 142,807 | 37.63 |
| test03 | 112,603 | 59,696 | 46.99 | 72,500 | 35.61 |
| test04 | 144,871 | 75,686 | 47.76 | 93,796 | 35.26 |
| test05 | 219,130 | 108,272 | 50.59 | 140,456 | 35.90 |
| test06 | 161,898 | 83,518 | 48.41 | 105,370 | 34.92 |
| test07 | 148,579 | 74,608 | 49.79 | 93,674 | 36.95 |
| test08 | 117,414 | 64,400 | 45.15 | 74,330 | 36.69 |
| test09 | 201,738 | 98,978 | 50.94 | 130,238 | 35.44 |
| test10 | 225,099 | 105,202 | 53.26 | 142,462 | 36.71 |
| test11 | 1,003 | 1,094 | -9.07 | 1,567 | -56.23 |
| test12 | 10,033 | 6,536 | 34.85 | 6,389 | 36.32 |
| test13 | 25,081 | 12,522 | 50.07 | 14,431 | 42.46 |
| test14 | 50,171 | 20,830 | 58.48 | 27,861 | 44.47 |
| test15 | 100,335 | 35,898 | 64.22 | 54,660 | 45.52 |
| test16 | 250,831 | 77,582 | 69.07 | 135,033 | 46.17 |
| test17 | 501,642 | 149,874 | 70.12 | 269,038 | 46.37 |
| test18 | 1,003,325 | 294,452 | 70.65 | 537,186 | 46.46 |
| test19 | 5,013,056 | 1,285,266 | 74.36 | 2,664,744 | 46.84 |
| test20 | 10,072,791 | 2,443,380 | 75.74 | 5,343,541 | 46.95 |


LZW
Huffman
100.00
80.00
60.00
40.00
20.00
0.00
-20.00
-40.00
-60.00
-80.00
Saving size ratio (%)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Test number

| Test Nr. | Uncom-pressed data size (bytes) | LZW | | Huffman | |
|---|---|---|---|---|---|
| | | Compressed size (bytes) | Space saving ratio (%) | Compressed size (bytes) | Space saving ratio (%) |
| test21 | 1,000 | 438 | 56.20 | 1,468 | -46.80 |
| test22 | 10,000 | 7,184 | 28.16 | 6,759 | 32.41 |
| test23 | 25,000 | 15,078 | 39.69 | 15,259 | 38.96 |
| test24 | 50,000 | 25,862 | 48.28 | 29,530 | 40.94 |
| test25 | 100,000 | 37,858 | 62.14 | 58,510 | 41.49 |
| test26 | 249,716 | 99,900 | 59.99 | 146,727 | 41.24 |
| test27 | 500,000 | 219,108 | 56.18 | 292,163 | 41.57 |
| test28 | 1,000,000 | 379,510 | 62.05 | 577,767 | 42.22 |
| test29 | 5,000,000 | 625,062 | 87.50 | 2,725,745 | 45.49 |
| test30 | 10,000,000 | 1,247,956 | 87.52 | 5,449,671 | 45.50 |
| test31 | 1,024 | 1,846 | -80.27 | 1,793 | -75.10 |
| test32 | 10,000 | 13,074 | -30.74 | 8,486 | 15.14 |
| test33 | 25,000 | 28,496 | -13.98 | 19,674 | 21.30 |
| test34 | 50,000 | 52,764 | -5.53 | 38,315 | 23.37 |
| test35 | 100,000 | 99,722 | 0.28 | 75,607 | 24.39 |
| test36 | 250,000 | 245,028 | 1.99 | 187,478 | 25.01 |
| test37 | 500,000 | 489,424 | 2.12 | 373,968 | 25.21 |
| test38 | 1,000,000 | 977,884 | 2.21 | 746,937 | 25.31 |
| test39 | 5,000,000 | 4,875,184 | 2.50 | 3,730,729 | 25.39 |
| test40 | 10,000,000 | 9,743,992 | 2.56 | 7,460,526 | 25.39 |



| Test Nr. | Uncom-pressed data size (bytes) | LZW | | Huffman | |
|---|---|---|---|---|---|
| | | Compressed size (bytes) | Space saving ratio (%) | Compressed size (bytes) | Space saving ratio (%) |
| test41 | 460,938 | 531,072 | -15.22 | 460,441 | 0.11 |
| test42 | 907,086 | 575,324 | 36.57 | 855,527 | 5.68 |
| test43 | 1,111,818 | 861,646 | 22.50 | 967,341 | 12.99 |
| test44 | 1,686,138 | 1,579,546 | 6.32 | 1,643,066 | 2.55 |
| test45 | 1,729,326 | 1,456,784 | 15.76 | 1,673,247 | 3.24 |
| test46 | 2,430,138 | 2,112,246 | 13.08 | 2,299,017 | 5.40 |
| test47 | 2,430,138 | 1,338,928 | 44.90 | 2,058,569 | 15.29 |
| test48 | 5,116,938 | 4,533,192 | 11.41 | 5,050,699 | 1.29 |
| test49 | 5,334,138 | 1,653,912 | 68.99 | 3,359,390 | 37.02 |
| test50 | 7,372,938 | 4,273,148 | 42.04 | 6,743,744 | 8.53 |
| test51 | 25,769 | 42,754 | -65.91 | 26,750 | -3.81 |
| test52 | 862,178 | 1,096,756 | -27.21 | 845,044 | 1.99 |
| test53 | 1,320,802 | 1,872,674 | -41.78 | 1,320,966 | -0.01 |
| test54 | 3,607,322 | 5,055,032 | -40.13 | 3,591,609 | 0.44 |
| test55 | 1,592,773 | 2,216,474 | -39.16 | 1,574,369 | 1.16 |
| test56 | 6,710,892 | 9,353,888 | -39.38 | 6,643,517 | 1.00 |
| test57 | 311,799 | 458,688 | -47.11 | 312,772 | -0.31 |
| test58 | 273,520 | 396,010 | -44.78 | 274,390 | -0.32 |
| test59 | 219,038 | 319,256 | -45.75 | 219,893 | -0.39 |
| test60 | 233,038 | 320,226 | -37.41 | 232,951 | 0.04 |



*Results interpretation*

Throughout the first 20 tests I observe that the LZW algorithm achieves a better compression than the Huffman coding. Test 11 is very small in size, has only 1003 bytes, so the size of the file increases when trying to compress it. The Huffman algorithm increases the size of this file more than the LZW because it must write the frequency of each character in the compressed file to be able to decode it.

Tests 21-30 follow the same pattern where the LZW achieves better compression. Test 21 is interesting because LZW achieved a very good compression, while the Huffman coding increased the data size. This is due to the highly repetitive text from the input file.

For tests 31-40, we can see that both algorithms are struggling to compress the unpredictable data, but the Huffman algorithm achieves far better compression. This is because Huffman coding compresses the data by modifying characters not strings.

Tests 41-50, the images in the bmp format, are overall better compressed by the LZW algorithm. Test 49 is far better compressed than the other photos, because of image content. A blue rose with a black background, which increased the compression ratio for both algorithms.

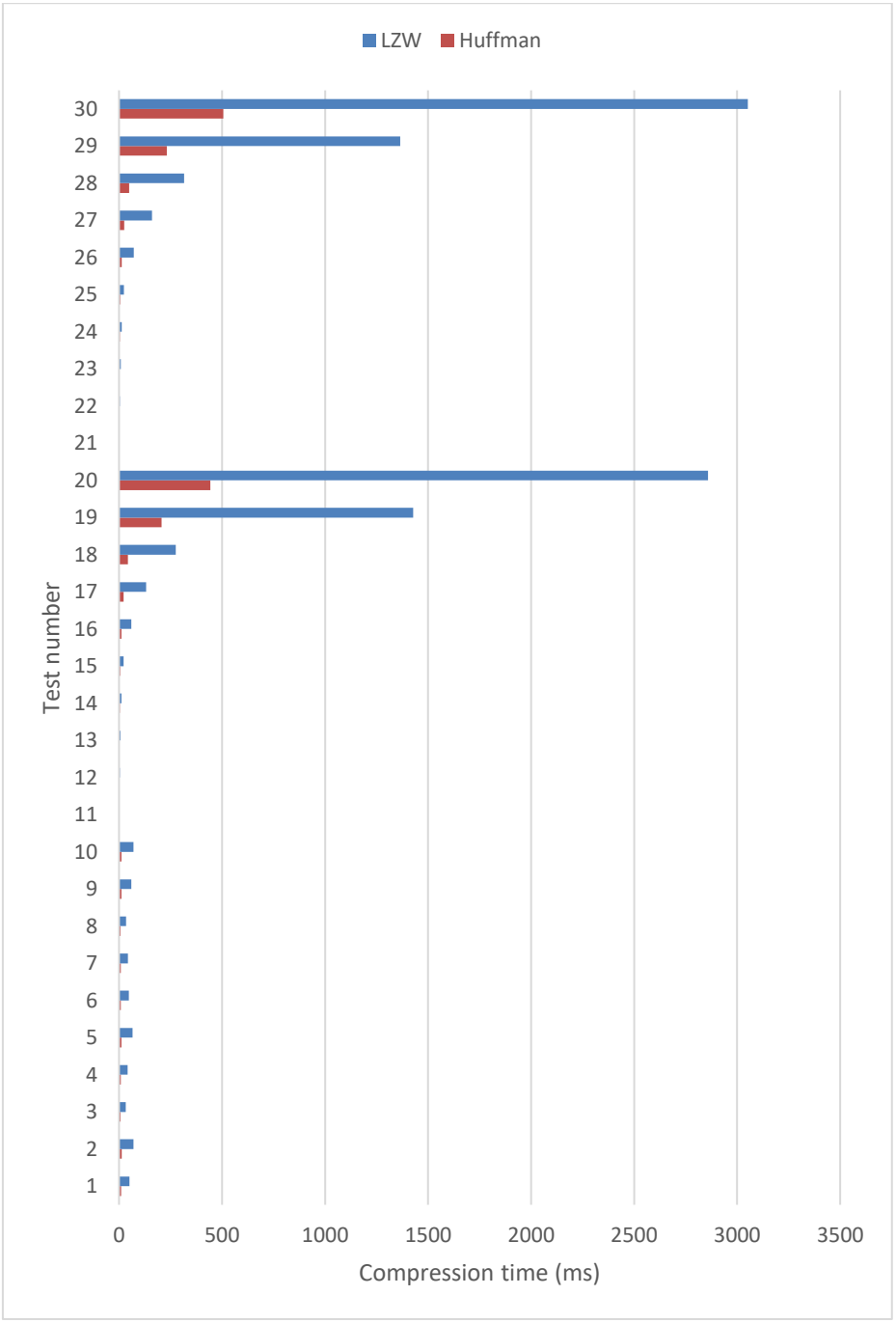
Tests 51-60 are files in formats that are already compressed, thus the compression ratio for the LZW algorithm are very bad, increasing the size of each file. The Huffman algorithm did better by not increasing the size of the files in a significant amount, and even achieving a 1-2% compression for some files.

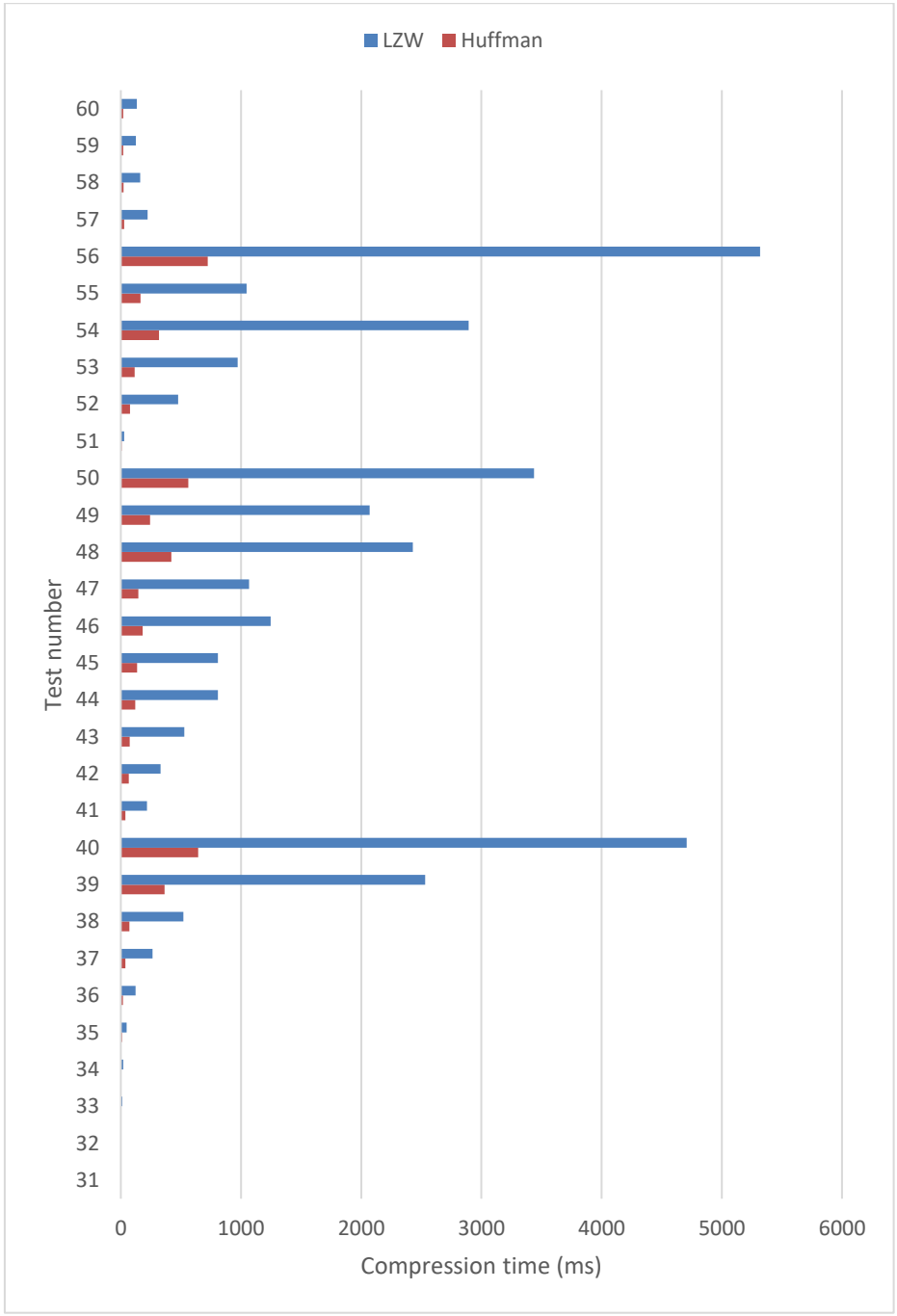
**Execution time**

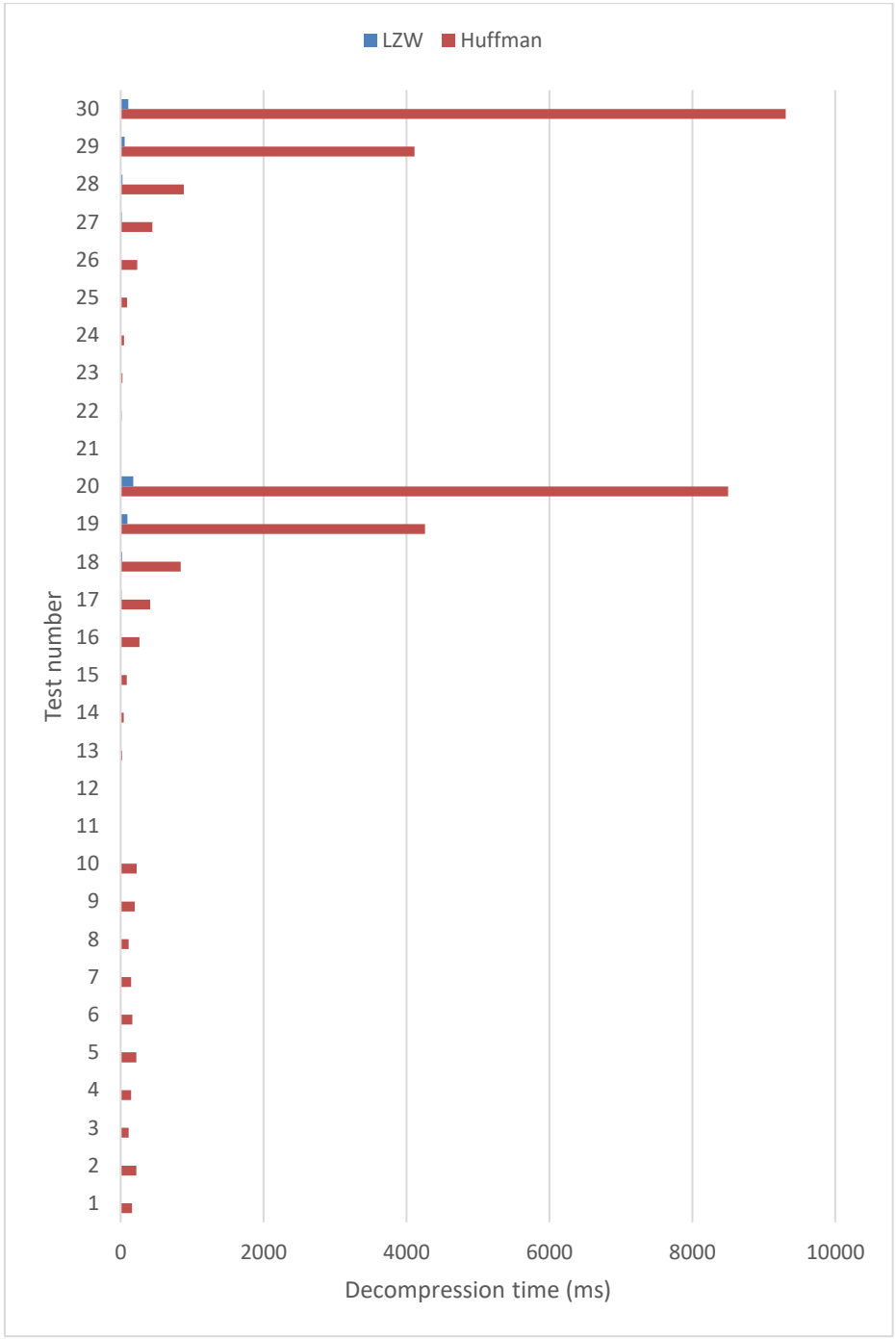
The execution time of the program was calculated using the hyperfine benchmarking tool.

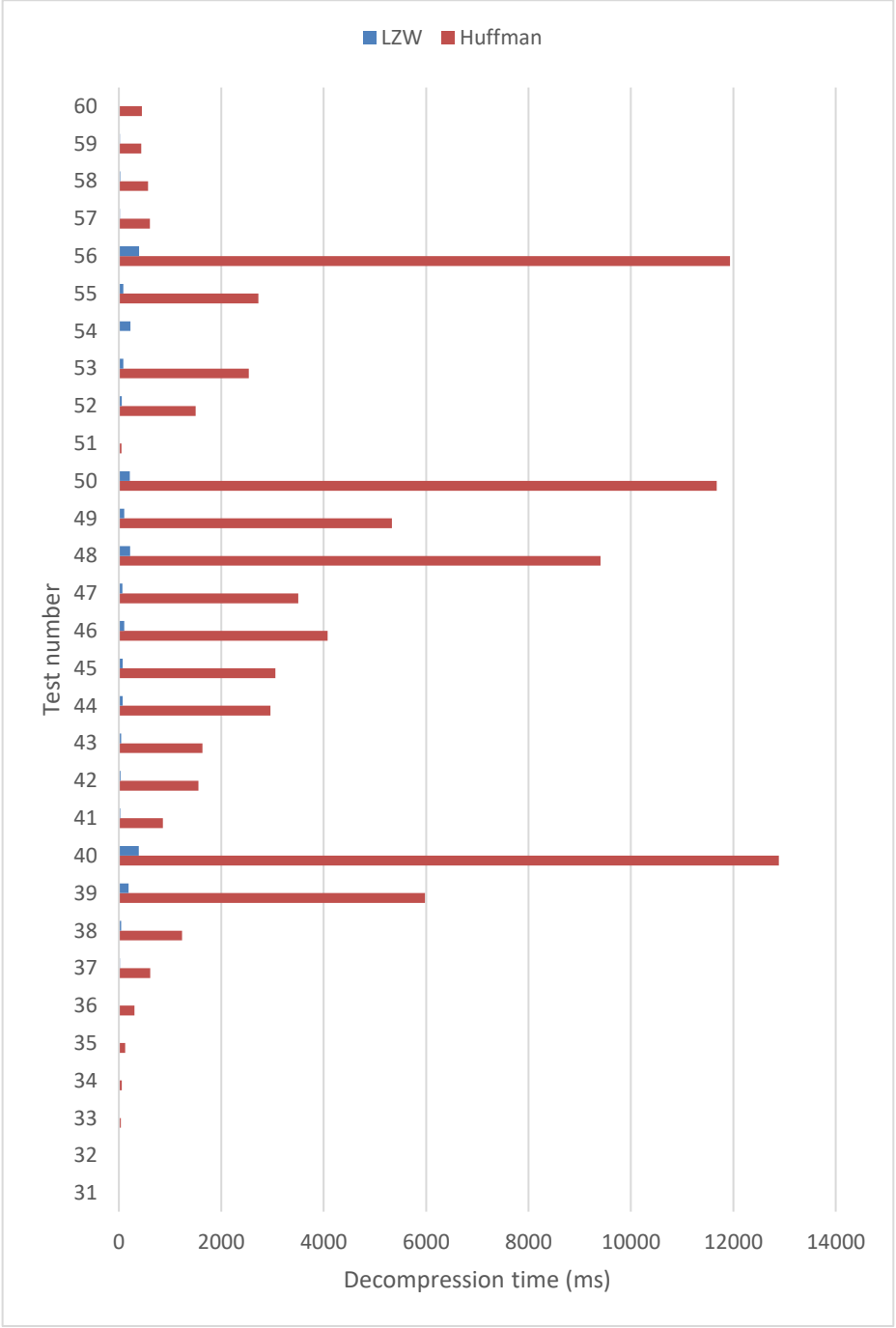
| Test Nr. | Uncompressed data size (bytes) | LZW | | Huffman | |
|---|---|---|---|---|---|
| | | Compression time (ms) | Decompression time (ms) | Compression time (ms) | Decompression time (ms) |
| test01 | 168,289 | 50.2 | 6.7 | 9.5 | 159.8 |
| test02 | 228,965 | 70.3 | 8.7 | 13.4 | 219.6 |
| test03 | 112,603 | 33.3 | 4.8 | 7 | 110.2 |
| test04 | 144,871 | 42.4 | 6.1 | 8.5 | 143.7 |
| test05 | 219,130 | 66.2 | 8.2 | 12.1 | 217 |
| test06 | 161,898 | 47.5 | 6.6 | 9.4 | 163.5 |
| test07 | 148,579 | 43.3 | 6.3 | 8.5 | 145.5 |
| test08 | 117,414 | 34.7 | 5.4 | 7.2 | 112.9 |
| test09 | 201,738 | 59.8 | 8 | 11.8 | 197.2 |
| test10 | 225,099 | 69.4 | 8.2 | 12.3 | 224.6 |
| test11 | 1,003 | 1.6 | 1.3 | 1.2 | 2.1 |
| test12 | 10,033 | 3.5 | 1.6 | 1.7 | 9.9 |
| test13 | 25,081 | 6.8 | 2.1 | 2.4 | 22.6 |
| test14 | 50,171 | 12 | 2.7 | 3.5 | 43.4 |
| test15 | 100,335 | 22.8 | 3.7 | 5.7 | 85.4 |

| | | | | | |
|---|---|---|---|---|---|
| test16 | 250,831 | 59.6 | 8 | 12.2 | 262.5 |
| test17 | 501,642 | 132.3 | 12.7 | 22.6 | 414 |
| test18 | 1,003,325 | 274.8 | 21.8 | 43.6 | 840.7 |
| test19 | 5,013,056 | 1427 | 94.7 | 206 | 4257 |
| test20 | 10,072,791 | 2859 | 174.5 | 443.1 | 8502 |
| test21 | 1,000 | 1.5 | 1.2 | 1.3 | 2 |
| test22 | 10,000 | 3.9 | 1.7 | 1.9 | 10.5 |
| test23 | 25,000 | 8 | 2.3 | 2.6 | 24.5 |
| test24 | 50,000 | 13.9 | 3 | 3.8 | 47 |
| test25 | 100,000 | 24.1 | 3.7 | 6.4 | 89.2 |
| test26 | 249,716 | 71.9 | 8.7 | 13.1 | 233.2 |
| test27 | 500,000 | 159.8 | 16 | 25.5 | 442.1 |
| test28 | 1,000,000 | 315.8 | 23.1 | 49.6 | 884.3 |
| test29 | 5,000,000 | 1365 | 55.4 | 231.5 | 4114 |
| test30 | 10,000,000 | 3052 | 104.9 | 506.2 | 9309 |
| test31 | 1,024 | 1.7 | 1.4 | 1.3 | 2.6 |
| test32 | 10,000 | 5 | 2 | 2.1 | 14.6 |
| test33 | 25,000 | 11.1 | 3 | 3.1 | 36.3 |
| test34 | 50,000 | 20.8 | 4.2 | 5 | 59.6 |
| test35 | 100,000 | 48.2 | 6.8 | 9.8 | 121.5 |
| test36 | 250,000 | 122 | 13.9 | 18.3 | 305.7 |
| test37 | 500,000 | 263.3 | 23.2 | 38.5 | 609.6 |
| test38 | 1,000,000 | 519.6 | 43.1 | 71.4 | 1232 |
| test39 | 5,000,000 | 2532 | 189.2 | 364.8 | 5976 |
| test40 | 10,000,000 | 4708 | 388.3 | 643.7 | 12888 |
| test41 | 460,938 | 218 | 24.8 | 37.5 | 858 |
| test42 | 907,086 | 330.3 | 31.9 | 66.7 | 1556 |
| test43 | 1,111,818 | 528 | 42.2 | 74.4 | 1630 |
| test44 | 1,686,138 | 808.5 | 75 | 119 | 2959 |
| test45 | 1,729,326 | 806.8 | 74.1 | 136.7 | 3059 |
| test46 | 2,430,138 | 1247 | 105 | 182.5 | 4078 |
| test47 | 2,430,138 | 1066 | 68.9 | 146.2 | 3502 |
| test48 | 5,116,938 | 2430 | 217.3 | 420.3 | 9404 |
| test49 | 5,334,138 | 2069 | 103.1 | 242.5 | 5334 |
| test50 | 7,372,938 | 3437 | 212.6 | 560.5 | 11679 |
| test51 | 25,769 | 26.7 | 3.6 | 6 | 51.9 |
| test52 | 862,178 | 477.4 | 55.4 | 77.4 | 1498 |
| test53 | 1,320,802 | 971.2 | 86.6 | 113.8 | 2535 |
| test54 | 3,607,322 | 2894 | 224.7 | 318.4 | 6.798 |
| test55 | 1,592,773 | 1046 | 89.8 | 163.3 | 2723 |
| test56 | 6,710,892 | 5318 | 391.8 | 722.5 | 11934 |
| test57 | 311,799 | 221.7 | 22.2 | 27.1 | 605.4 |
| test58 | 273,520 | 160.3 | 23.8 | 23.5 | 569.6 |
| test59 | 219,038 | 125.2 | 17.8 | 19.2 | 434.5 |
| test60 | 233,038 | 132.2 | 17.4 | 20.7 | 449.2 |


LZW
Huffman
Test number
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
500
1000
1500
2000
2500
3000
3500
Compression time (ms)


LZW
Huffman
Test number
60
59
58
57
56
55
54
53
52
51
50
49
48
47
46
45
44
43
42
41
40
39
38
37
36
35
34
33
32
31
0
1000
2000
3000
4000
5000
6000
Compression time (ms)


LZW
Huffman
Test number
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
2000
4000
6000
8000
10000
Decompression time (ms)


LZW
Huffman
Test number
60
59
58
57
56
55
54
53
52
51
50
49
48
47
46
45
44
43
42
41
40
39
38
37
36
35
34
33
32
31
0
2000
4000
6000
8000
10000
12000
14000
Decompression time (ms)

*Result interpretation*

The compression times of the Huffman coding algorithm implementation are significantly better than those of the Lempel-Ziv-Welch algorithm. I think that a more optimized version of the LZW algorithm could achieve similar or even better results.

The decompression times of the LZW algorithm are much better than the ones of Huffman, which were surprisingly slow comparing to the other one.

**Memory usage**

| Test Nr. | Uncompressed data size (bytes) | LZW | | Huffman | |
|---|---|---|---|---|---|
| | | Compression memory usage (KB) | Decompression memory usage (KB) | Compression memory usage (KB) | Decompression memory usage (KB) |
| test01 | 168,289 | 7848 | 5404 | 1524 | 1520 |
| test02 | 228,965 | 9056 | 6280 | 1524 | 1516 |
| test03 | 112,603 | 6520 | 4636 | 1520 | 1516 |
| test04 | 144,871 | 7312 | 5228 | 1516 | 1520 |
| test05 | 219,130 | 9764 | 5932 | 1516 | 1580 |
| test06 | 161,898 | 7856 | 5468 | 1516 | 1516 |
| test07 | 148,579 | 7328 | 5128 | 1528 | 1524 |
| test08 | 117,414 | 6784 | 4940 | 1520 | 1524 |
| test09 | 201,738 | 8624 | 5952 | 1584 | 1524 |
| test10 | 225,099 | 8920 | 5892 | 1528 | 1528 |
| test11 | 1,003 | 3332 | 3312 | 1528 | 1524 |
| test12 | 10,033 | 3492 | 3392 | 1584 | 1584 |
| test13 | 25,081 | 3892 | 3488 | 1528 | 1520 |
| test14 | 50,171 | 4372 | 3504 | 1524 | 1524 |
| test15 | 100,335 | 5156 | 4016 | 1584 | 1524 |
| test16 | 250,831 | 7308 | 5200 | 1528 | 1520 |
| test17 | 501,642 | 10228 | 6716 | 1520 | 1520 |
| test18 | 1,003,325 | 10224 | 6728 | 1580 | 1528 |
| test19 | 5,013,056 | 10112 | 6712 | 1528 | 1528 |
| test20 | 10,072,791 | 10240 | 6780 | 1584 | 1580 |
| test21 | 1,000 | 1524 | 3280 | 1528 | 1516 |
| test22 | 10,000 | 3628 | 3316 | 1580 | 1524 |
| test23 | 25,000 | 3752 | 3516 | 1520 | 1524 |
| test24 | 50,000 | 4676 | 3756 | 1520 | 1524 |
| test25 | 100,000 | 5080 | 4148 | 1524 | 1528 |
| test26 | 249,716 | 8644 | 5860 | 1516 | 1528 |
| test27 | 500,000 | 10140 | 6728 | 1520 | 1524 |
| test28 | 1,000,000 | 10240 | 6800 | 1520 | 1528 |
| test29 | 5,000,000 | 10672 | 7728 | 1528 | 1520 |

| | | | | | |
|---|---|---|---|---|---|
| test30 | 10,000,000 | 10704 | 7864 | 1528 | 1580 |
| test31 | 1,024 | 3380 | 3308 | 1528 | 1528 |
| test32 | 10,000 | 3888 | 3624 | 1524 | 1520 |
| test33 | 25,000 | 4600 | 3876 | 1584 | 1528 |
| test34 | 50,000 | 5956 | 4664 | 1520 | 1584 |
| test35 | 100,000 | 8624 | 5992 | 1580 | 1588 |
| test36 | 250,000 | 10092 | 6660 | 1528 | 1524 |
| test37 | 500,000 | 10244 | 6712 | 1520 | 1516 |
| test38 | 1,000,000 | 10216 | 6740 | 1516 | 1516 |
| test39 | 5,000,000 | 10224 | 6752 | 1584 | 1580 |
| test40 | 10,000,000 | 10228 | 6796 | 1524 | 1516 |
| test41 | 460,938 | 10212 | 6784 | 1584 | 1528 |
| test42 | 907,086 | 10180 | 6744 | 1524 | 1512 |
| test43 | 1,111,818 | 10440 | 6928 | 1516 | 1520 |
| test44 | 1,686,138 | 10484 | 6724 | 1520 | 1584 |
| test45 | 1,729,326 | 10104 | 6656 | 1524 | 1524 |
| test46 | 2,430,138 | 10148 | 6656 | 1524 | 1524 |
| test47 | 2,430,138 | 10368 | 7212 | 1588 | 1516 |
| test48 | 5,116,938 | 10440 | 6940 | 1580 | 1524 |
| test49 | 5,334,138 | 11020 | 7840 | 1520 | 1524 |
| test50 | 7,372,938 | 10676 | 7840 | 1584 | 1516 |
| test51 | 25,769 | 5468 | 4336 | 1584 | 1584 |
| test52 | 862,178 | 10112 | 6704 | 1584 | 1520 |
| test53 | 1,320,802 | 10240 | 6712 | 1584 | 1528 |
| test54 | 3,607,322 | 10156 | 6728 | 1584 | 1524 |
| test55 | 1,592,773 | 10240 | 6800 | 1584 | 1524 |
| test56 | 6,710,892 | 10240 | 6784 | 1520 | 1524 |
| test57 | 311,799 | 10088 | 6776 | 1584 | 1528 |
| test58 | 273,520 | 10212 | 6772 | 1524 | 1524 |
| test59 | 219,038 | 10148 | 6728 | 1584 | 1524 |
| test60 | 233,038 | 10172 | 6660 | 1580 | 1520 |

*Result interpretation*

As expected, the LZW algorithm is consuming more memory in order to both compress and decompress files, due to the code dictionary that is using.

# 4 Conclusion

Taking all into account, I think that both algorithms have their utility and necessity, but the better standalone data compression utility from the two for compressing text files is the Lempel-Ziv-Welch algorithm. It achieved better overall compression ratios on text files and the execution time for a full cycle of compressing and decompressing a file was better since its decompression time was far superior even if it was lacking a bit in

compression speed. However, the Huffman compression algorithm proves better for compressing high entropy files that have unpredictable data, such as multimedia data files and even already compressed files.

# 5 Bibliography


[1] O. A. Mahdi, M. A. Mohammed and A. J. Mohamed, ""Implementing a novel approach an convert audio compression to text coding via hybrid technique."," *International Journal of Computer Science Issues (IJCSI),* vol. 9, no. 6, pp. 53-59, 2012.

[2] J. H. Pujar and L. M. Kadlaskar, "A new lossless method of image compression and decompression using Huffman coding techniques.," *Journal of Theoretical & Applied Information Technology,* vol. 15, no. 1, pp. 18-23, 2010.

[3] P. Crocetti, "Data Compression," TechTarget, December 2022. [Online]. Available: https://www.techtarget.com/searchstorage/definition/compression. [Accessed 07 January 2023].

[4] T. Welch, "A Technique for High-Performance Data Compression," *Computer,* vol. 17, no. 6, pp. 8-19, 1984.

[5] A. R. Saikia, "LZW (Lempel–Ziv–Welch) Compression technique," 08 November 2021. [Online]. Available: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/. [Accessed 07 January 2023].

[6] "C++ map - count() Function," [Online]. Available: https://www.alphacodingskills.com/cpp/notes/cpp-map-count.php. [Accessed 07 January 2023].

[7] R. Thapliyal, "map vs unordered_map in C++," 09 November 2022. [Online]. Available: https://www.geeksforgeeks.org/map-vs-unordered_map-c/. [Accessed 07 January 2023].

[8] "map::at," [Online]. Available: https://cplusplus.com/reference/map/map/at/. [Accessed 07 January 2023].

[9] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE,* vol. 40, no. 9, pp. 1098-1101, 1952.

[10] A. Barnwal, "Huffman Coding," 21 December 2022. [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. [Accessed 07 January 2023].