# Compression algorithms

Răducanu George-Cristian, 321CA

University Politehnica of Bucharest

**Abstract.** The paper analyses several algorithms for compression, taking into account the speed, compression ratio and the resources needed during the process. Also, some algorithms are designed in such a way to read the decompression information as the process is running, while in others you have to wait for the entire operation to finish. The algorithms efficiency depend on the nature of the input and its repetitive characteristics.

**Keywords:** Compression · Decompression · Huffman Coding · Arithmetic Coding · Lempel-Ziv-Welch.

## 1 Introduction

Since the apparition of the first computers and even earlier, storage size has been a challenge. Although modern computers have more space than they used to have, storage size is still a problem encountered often due to the continuous increase in the volume of information available. Especially in the data transmission it is needed to have the data occupy as little as possible. Also, in storing a big file or folder not accessed often it is recommended to compress it.

Over the years there have been developed several algorithms to overcome the problem of compression. The ones analyzed in this paper are: Huffman coding, Arithmetic coding and the Lempel-Ziv-Welch algorithm.

**Short description of the algorithms** The classic Huffman coding uses optimal prefix code to construct minimum redundancy codes.[1] The output of the algorithm is a variable-length code (v.l.c.) for the characters in the file. This variable length code is derived from the frequency of apparition of the characters and is considered thanks to this fact to be an entropy based coding.[7]

Another entropy coding, in most cases better than the Huffman coding is the Arithmetic coding. It is less widespread in compression (the case of image compression - JPEG) due to the fact that there were numerous patents on these classes of algorithms until the middle of 2010. It takes a different approach, by storing a larger string into a single floating-point number between *0* and a *max-value* (usually 1), with the help of probabilities. For the compression of text files the idea is to give each character a probability of appearance. This can be equal for all symbols, but most often the probabilities are known before for a language from analyzing each character from each word of this language to obtain a real-world probability.[5] In this way the probabilities (range of probabilities/cumulative probabilities) are composed at the apparition of each character with the corresponding interval. The decompression is also very efficient, the only thing you know are the probabilities that have been used. Because of this fact, the algorithm calibrated on a certain language does not achieve the same performance for another language or alphabet structure.

A third algorithm widely used nowadays is the Lempel-Ziv-Welch compression algorithm. It is used in the Linux kernel and is found in many areas, such as image and sound compression. The idea behind this algorithm is to try to extend the "alphabet" by encoding 8-bit characters as usually 12-bit characters. In the extra encoding we store the sequences that repeat. Although the new "characters" are stored on 12 bits (or any other value) the compression is efficient, especially for redundant text.[5]

## 2 Testing Criteria

### 2.1 Tests

To have a correct evaluation of the algorithms, the tests will consist of real-world files but, also some corner cases to test out the limits of the algorithms, and also some random generated text. From the point of view of the theory of information transmission, tests with different entropies will be considered.

The tests will be as exhaustive as possible, containing paragraphs and multiple chapters from English literature books for real world simulations of the algorithms. Also, there will be some large files to test the limit the algorithms. For testing corner cases there will be files containing a small set of characters but with lots of apparitions in both predictable and non-predictable orders. For example, some of the tests will contain random text, while others will consist of practical applications, compressing sequences of DNA and RNA that contain a small alphabet(A, C, T/U, G). There will be binary text inputs as well.

After the compression and decompression is done there will be a check between the original file and the one obtained to be sure and to demonstrate that the compression is lossless. The time needed for both compression and decompression will be obtained and at the end of the tests, a comparison for each test will be done for both time and compression ratio.

### 2.2 Programs used for testing

The programs used for testing are from GitHub and are licensed. All the code is written in C/C++ and is as efficient as possible, while maintaining the original ideas of the algorithms. C/C++ is a good language for implementing these algorithms thanks to it's efficiency and speed.

All the programs are tested with the help of the *valgrind* tool so that they don't have any memory leaks. When using the *gcc* compiler there are no optimization flags, just the flags for checking warnings. No optimizations flags are used because our purpose is to analyze the algorithms and not the optimization power of the compiler.

## 3 Description of the algorithms

### 3.1 Huffman coding

Huffman coding uses prefix codes. It works by creating a binary tree of nodes, where a node corresponds to a symbol in the data. The frequency of each symbol is used in determining the probability of appearance of each character. The tree is constructed in such a way that the nodes with higher frequency are closer to the root, allowing a more efficient traversal. The more efficient traversal can only be done if the most frequently occurring symbols use fewer bits in encoding. The first step is to construct a frequency table. Then make a tree by repeatedly merging the two characters with lowest frequency that have not been used. A left traversal through the tree is marked as 0, and a right traversal is marked as 1. Thus we obtain the new codes associated with each symbol from the alphabet.

The theoretical time complexity of the Huffman algorithm is $\mathcal{O}(n \log n)$ where $n$ represents the number of symbols from the input file. This time complexity is due to the construction of the binary tree. The space complexity of the algorithm is $\mathcal{O}(n)$ due to the frequency vector and of the tree.[4]

For the inverse part, the decompression, we need to construct the tree using the prefix code, which was stored alongside the compressed data. After building the tree, we must traverse the tree following the path represented by the bits in the compressed file until a leaf is reached. This last process is repeated until the entire file is decompressed.

### 3.2 Arithmetic coding

The base idea of the arithmetic coding algorithm is to represent a sequence of characters as a fraction. The resulted fraction is then usually encoded using a fixed number of bits. The algorithm is very efficient, with high compression ratio for files with redundancy. It is also more simple to implement than other algorithms. The reverse operation, of decoding follows the reverse path, and determines the characters (from last to first) by analyzing the fraction, and then updating it for the next step after a symbol is found. In this mode, for a complete decompression, it is necessary to wait for the entire data package and complete the entire process.

Complexity depends on the implementation, generally both the encoding and decoding algorithms have $\mathcal{O}(n)$ complexity, where $n$ is the number of characters/symbols in the input file.

The arithmetic coding is considered to be a moderate compression algorithm, and is quite popular due to it's simplicity. It is often used in hybrid compression algorithms.

### 3.3 Lempel-Ziv-Welch (L.Z.W.)

This algorithm was developed in 1970, and then improved in 1980 [5], reaching it's today structure. The LZW algorithms identifies repeating sequence of patterns and replaces them with a reference to the first appearance. The decompression replaces all the references with the original phrases and thus the original file is obtained. It is a fast algorithm and achieves very good compression rates, but it has a a higher space complexity due to the fact that it requires a dictionary to store the phrases.

LZW is a greedy type algorithm, in the sense that it looks for the first possible expansion of the dictionary, without necessarily being the most suitable from the point of view of the compression ratio.

Due to its great compression ratio, and it's time complexity of $\mathcal{O}(n)$, this algorithm is used in lots of practical context, alongside hybrid algorithms.

### 3.4 Hybrid algorithms

Modern compression algorithms are hybrid ones, that use a combination of the algorithms described above. In this way, we can take advantage of different properties of the algorithms. Such a modern algorithm is DEFLATE which is widely used today in zip files. Other specific file formats use different hybrids such as JPEG2000 for images (wavelet transform + arithmetic coding), or MP3 (psychoacoustic modelling + Huffman coding) or video compression (entropy coding + block-based transform).

## 4 Testing

### 4.1 Machine used for testing

For the tests I have used my personal laptop with a i7-1165G7, 16GB of RAM, running Ubuntu 22.10. The tests where run using the gcc compiler, version 12.2.0, without any optimization flags. All the tests where run with the laptop plugged in, on the Performance mode from Ubuntu.

### 4.2 Programs

The programs used for the testing are from github. The links to the repositories are in the bibliography and also in the README. All the flags for optimizations where removed allowing only the default optimizations. The decompressed files are compared with the original ones using diff.

## 5 Tests

There are 20 tests each with its own characteristics. The first test is a small one, to ensure that the decompression also works. Then about half of the remaining tests analyze the behaviour of the algorithms on real world files. The real world tests are literature, plays of W. Shakespeare, chapters of books (Tarzan), and even poetry to analyze the behaviour. Another category of real world tests are those consisting of code. Using GPT3, I generated code in C++ and ASM, x86 and also a small fairy tale. In this way several kind of texts are taken into account.

The other half of the tests are generated randomly by hand (blind-folded), generated randomly using *bash* scripts. Also there is a test with a large matrix generated in Octave. For the 15th test, I have decided to try to compress part of the 15th human chromosome to test a real world, scientific application. Using *convert* and *hexdump* from linux a 4K image was put into a text file. The few last test consist of ASCII art and a lot of words from the English language.

A more detailed description of the content of the tests can be found in Table1.

**Table 1.** Description of test files input

| Test no. | Test description |
|---|---|
| 01 | basic dummy in file containing 175 characters |
| 02 | text file containing Romeo and Juliet play with 153292 characters |
| 03 | another text file with theater play 140098 |
| 04 | text file containing newlines and also a theater play with 140894 characters |
| 05 | text file containing another 2 chapters from Tarzan, 17913 symbols |
| 06 | file containing poetry, 12231 characters |
| 07 | text file containing C++ code, 1351 characters |
| 08 | Assembly code generated using GPT-3 , 2373 symbols |
| 09 | file containing fairy tale generated by the AI, GPT-3, 6426 characters |
| 10 | test generated by typing blind-folded, 12725 symbols |
| 11 | file containing random generated test using /dev/urand, 10071 characters |
| 12 | text obtained with random and hexdump, limiting the alhpabet size, 251697 characters |
| 13 | text file containing 0 and 1 generated randomly, 651532 symbols |
| 14 | file containing random matrix and eigenvalues, 18472 symbols |
| 15 | text file containing parts of the 15th human chromosome, 120772 symbols |
| 16 | file obtained by doing hexdump un a 4K image, 415634 characters |
| 17 | text file with lots of repetitive characters and small insertions, 6550 symbols |
| 18 | file containing ASCII art , 5691 symbols |
| 19 | more ASCII art of different type, 7174 characters |
| 20 | test containing letters from the English dictionary, 22288 symbols |

## 6 Results and analysis

### 6.1 Compression ratio

After running all the programs several times, the results obtained are presented in Table2 and Table5. Table2 presents the size of both the original and compressed files is in bytes [B].

**Table 2.** Sizes of original and compressed files with Huffman, Arithmetic and LZW coding

| Test no. | Original file size [B] | Huffman coding [B] | Arithmetic coding [B] | L.Z.W. [B] |
|---|---|---|---|---|
| 01 | 157 | 230 | 124 | 128 |
| 02 | 153293 | 88940 | 87707 | 78300 |
| 03 | 140892 | 79740 | 78041 | 67616 |
| 04 | 19921 | 11388 | 11157 | 10218 |
| 05 | 17912 | 10204 | 10001 | 9215 |
| 06 | 12230 | 7338 | 7068 | 6595 |
| 07 | 1350 | 1160 | 963 | 887 |
| 08 | 2372 | 1718 | 1513 | 1212 |
| 09 | 6424 | 3804 | 3667 | 3061 |
| 10 | 12725 | 6035 | 5919 | 2801 |
| 11 | 10069 | 7981 | 7762 | 10622 |
| 12 | 251697 | 128075 | 126214 | 143085 |
| 13 | 631527 | 204196 | 199329 | 178025 |
| 14 | 18748 | 8970 | 8897 | 8114 |
| 15 | 120771 | 47872 | 46342 | 40650 |
| 16 | 415631 | 208537 | 204378 | 212219 |
| 17 | 6535 | 1208 | 866 | 535 |
| 18 | 5687 | 2494 | 2236 | 2078 |
| 19 | 7172 | 2442 | 2295 | 1873 |
| 20 | 22287 | 12669 | 11996 | 11439 |

A short analysis of the table above suggests that the L.Z.W. algorithm has the higher compression ratio.

The tests are ordered by the nature of the input and not by the size. It can be observed that the L.Z.W. algorithm obtains higher compression ratios in the most cases.

It is known that the compression ratio is defined as the ratio between the size of the input file and the size of the compressed file. It is expressed as *n:1* or *n/1*. When $n$ is greater than *1* the output file is smaller than the input file.

In Table3 the compression ratios are calculated using the data from Table2.

**Table 3.** Compression ratios for Huffman, Arithmetic and L.Z.W. coding

| Test no. | Huffman ratio | Arithmetic coding ratio | L.Z.W. ratio |
|---|---|---|---|
| 01 | 0.68:1 | 1.26:1 | 1.22:1 |
| 02 | 1.72:1 | 1.74:1 | 1.95:1 |
| 03 | 1.76:1 | 1.80:1 | 2.08:1 |
| 04 | 1.74:1 | 1.78:1 | 1.95:1 |
| 05 | 1.76:1 | 1.79:1 | 1.94:1 |
| 06 | 1.66:1 | 1.73:1 | 1.85:1 |
| 07 | 1.16:1 | 1.40:1 | 1.52:1 |
| 08 | 1.38:1 | 1.56:1 | 1.96:1 |
| 09 | 1.68:1 | 1.75:1 | 2.09:1 |
| 10 | 2.10:1 | 2.14:1 | 4.50:1 |
| 11 | 1.26:1 | 1.29:1 | 1.01:1 |
| 12 | 1.96:1 | 1.99:1 | 1.75:1 |
| 13 | 3.09:1 | 3.16:1 | 3.58:1 |
| 14 | 2.09:1 | 2.10:1 | 2.31:1 |
| 15 | 2.52:1 | 2.61:1 | 2.98:1 |
| 16 | 1.99:1 | 2.03:1 | 1.95:1 |
| 17 | 5.41:1 | 7.54:1 | 12.21:1 |
| 18 | 2.28:1 | 2.54:1 | 2.73:1 |
| 19 | 2.93:1 | 3.12:1 | 3.82:1 |
| 20 | 1.75:1 | 1.86:1 | 1.94:1 |

For statistics reasons it is necessary to have a mean of ratio compression for each algorithm. In this case, the mean is calculated for all 20 tests. The final results are presented in Table4.

**Table 4**. Averages of ratio compression for all the algorithms

| Huffman ratio | Arithmetic coding ratio | L.Z.W ratio |
|---|---|---|
| 2.046:1 | 2.256:1 | 2.765:1 |

It can be concluded that the Arithmetic coding is with roughly 10% better than Huffman, and the L.Z.W. algorithm is 20% better than the Arithmetic coding.

Although, such a small amount of tests cannot be used to make a definitive answer, due to the extensive nature of the tests. It can be concluded that ranking from lowest to the best in terms of compression ratio is: Huffman, Arithmetic and LZW.

A hybrid approach combining two of the following, such as the DEFLATE algorithm may potentially give better results. For example, using gzip on the tests gave a 10% increase ratio than L.Z.W.

### 6.2 A time efficiency analysis

In Table 5 only the encoding time is illustrated, the decoding time being roughly small and similar for all algorithms. All the information from the table is in milliseconds [ms].

**Table 5.** Execution time for Huffman, Arithmetic and LZW encoding

| Test no. | Huffman time[ms] | Arithmetic coding time [ms] | L.Z.W. time [ms] |
|---|---|---|---|
| 01 | 1 | 1 | 1 |
| 02 | 39074 | 12 | 400 |
| 03 | 26622 | 15 | 346 |
| 04 | 502 | 3 | 54 |
| 05 | 351 | 3 | 45 |
| 06 | 249 | 2 | 30 |
| 07 | 8 | 1 | 1 |
| 08 | 8 | 1 | 2 |
| 09 | 25 | 2 | 10 |
| 10 | 86 | 2 | 9 |
| 11 | 269 | 3 | 26 |
| 12 | 78325 | 24 | 766 |
| 13 | 138809 | 41 | 1390 |
| 14 | 93 | 3 | 39 |
| 15 | 5610 | 9 | 257 |
| 16 | 252389 | 38 | 1118 |
| 17 | 2 | 1 | 2 |
| 18 | 16 | 2 | 4 |
| 19 | 8 | 2 | 4 |
| 20 | 1009 | 3 | 53 |

Although it clearly depends of the nature of implementation, we can at least conclude that both theoretically and practically the Huffman algorithm has higher time complexity. The arithmetic is faster than the LZW algorithm, but it depends of the nature of the CPU (type, architecture, generation, etc) as the Arithmetic coding is higly dependant on the multiplication and divide operations on floating point numbers.

### 6.3 Conclusions

Table 3 shows that the compression ratios are approximately the same for a compression algorithm, regardless of the content (images, literary and random texts, C++ code, etc.) and the file size.

For test 17, which contains many "A" characters with small insertions of other symbols, a high compression ratio can be observed regardless of the chosen algorithm. For this test it is observed that the maximum compression ratio is achieved with the LZW algorithm.

In theory, it is mentioned that the Huffman coding algorithm [7] is optimal for a distribution of the probability of occurrence of dyadic symbols (negative powers of 2). If the probability distribution structure departs from this condition and there is a large number of symbols, the algorithm decreases in coding (compression) efficiency.

From the proposed tests, it can be seen that the theoretical statement regarding the time complexity of the algorithms is verified practically on the proposed tests. Thus, the ratio between the longest execution time and the shortest leads to the following ranking in descending order of this ratio: Huffman coding, LZW coding, Arithmetic coding.

In conclusion, according to the analyzed tests and the results obtained from tables 3 and 4, the compression ratios can be graphically represented in Figure 1.

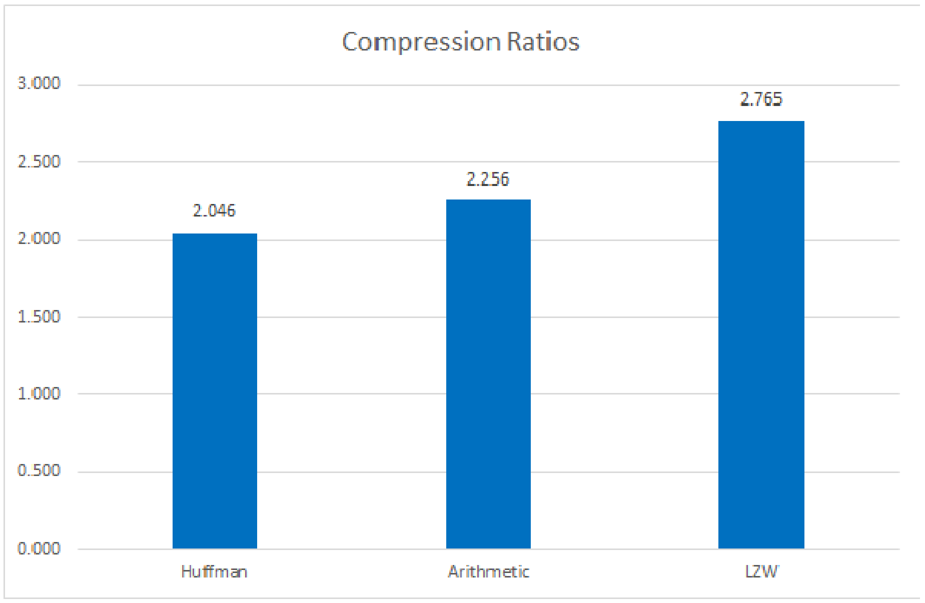

Figure 1. Averages for all tests of ratio compression for the algorithms

## References

1. Thomas M. Cormen et. all; Introduction to Algorithms, fourth edition
2. Robert Sedgewick, Kevin Wayne; Algorithms 4th edition
3. Antti Laaksonen; Competitive Programmer's Handbook 2018
4. Radu Iacob, Algorithms Analysis - seminary notes 2022-2023, Faculty of Automatics and Computer Science, "POLITEHNICA" University of Bucharest
5. https://en.wikipedia.org/wiki/Arithmetic_coding
6. https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
7. https://en.wikipedia.org/wiki/Huffman_coding
8. https://www.youtube.com/watch?v=umTbivyJoiI&t=50s
9. https://www.youtube.com/watch?v=goOa3DGezUA
10. https://github.com/PrototypePHX/huffman
11. https://github.com/dmitrykravchenko2018/arithmetic_coding
12. https://github.com/eloj/lzw-eddy