Data compression

Dragos Andrei Radut

University Politehnica of Bucharest

Abstract. Data compression is the technique that makes it possible to represent information in a compact form, increasing efficiency in data storage and transmission. This paper briefly presents the base idea of encoding and decoding this information in order to reduce redundancy and represent it in a condensed way. Besides an overview of the chosen algorithms (Huffman Coding and Lempel-Ziv-Welch), this paper provides a comparative analysis of the two, following parameters like compression ratio, space savings and resources used (time/memory).

Keywords: - Data Compression · Huffman coding · Lempel-Ziv-Welch · Lossless Techniques · Algorithm analysis

1 Introduction

1.1 The Problem Statement

Since the amount of data in use is facing an continuous increase day by day, data compression became a vital approach in storing and transmitting information. The idea of size reducing is based on representing data in a less redundant way, while being able to recover initial content. This recovery splits data compression techniques into two main categories: Lossy compression (involves some loss of data) and Lossless compression (initial data is recovered completely).

This paper approaches Lossless compression of text only, containing algorithm analysis of two widely used techniques: Huffman Coding and Lempel-Ziv-Welch.

1.2 Practical Applications

Data compression algorithms are used for both text and multimedia content, making it possible to share any kind of information efficiently over the internet. This efficiency is not visible only in space savings, but with smaller sizes come higher transmission speed, reduced access time and reduced use of bandwidth. Lossless technique is especially used in text compression, where character modification of the original file will result in change of meaning.

1.3 Solutions overview

As previously stated, this study covers two algorithms used in lossless data compression:

- **Huffman Coding**: entropy based; encoding is realised by following a binary tree constructed by sorting input characters by probability of appearance; symbols are stored in leafs and the code is generated by traversing from root to each leaf and adding '0' for choosing the left child or '1' for the right one.
- **Lempel-Ziv-Welch (LZ77)**: dictionary based; encoding is realised by finding sequences and creating a dictionary with each unique one; redundant patters are replaced with pointers to already existing ones form the dictionary

1.4 Evaluation Criteria

Compression's main goal is size reducing. Text compression is always Lossless, therefore correctness of test results is guaranteed. Decompressed files must be identical to original files given as input in order to be considered as a relevant efficiency test. This factor is verified using "diff".

The same set of tests is applied to both algorithms to provide a comparative analysis of the two, targeting different situations. The results are interpreted using the following parameters:

- **Execution Time** : for both compression and decompression
- **Compression/Decompression Speed**
$$\frac{CompressedFileSize}{ExecutionTime} \tag{1}$$
- **Compression Ratio**
$$\frac{CompressedFileSize}{OriginalFileSize} \tag{2}$$
- **Saving Percentage**
$$\frac{OriginalFileSize - CompressedFileSize}{OriginalFileSize} \tag{3}$$

Using these parameters as evaluation criteria, each algorithm's efficiency is determined. Although the best method between the two can be determined, techniques of implementation are different, both having advantages and disadvantages on given situations.

2 The solutions

2.1 Huffman Coding

- **Algorithm description** Huffman coding creates a binary tree ordered by probability where leafs are input data symbols. Traversing this tree form root to each leaf, a code will be generated by adding '0' for left child and '1' for right child. Symbols are added to the tree in reversed order, meaning that characters with higher probability will be closer to root. Decoding is realised by traversing the tree again, following the codification from root to leaf and replacing the symbol found.

- **Storing compressed data** Compression is made reducing character representation by replacing a 1 byte character with a maximum of 5 bits representation. In order to decode, the three must be passed. This can be realised by in many ways, such as agreeing on an encoding tree in advance. The implementation chosen creates a header containing each symbol with it's corresponding code.

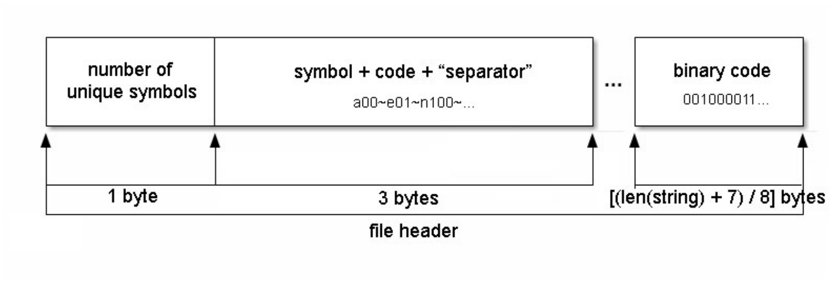

Fig.1 Huffman coding compressed file

- **Algorithm complexity** Compression and decompression speed relies on the size of file given as input. The main difference in efficiency can be briefly illustrated in the following scheme:

Table.1 Relative Speed

Algorithm	Compression Speed	Decompression Speed
Entropy-based	medium	medium

For each test, Huffman coding can be performed in O(nlog(n)), n representing the number of unique characters found in file passed to compression. Different input may result in better time complexity (for already sorted text), while completely random strings will visibly affect both time and space used. Huffman's technique is based on entropy, therefore conveniently ordered strings are compressed more efficiently.

- **Main advantages and disadvantages** The primary advantage of this technique is efficiently replacing 8 bit symbols with shorter binary codes, the most recurring ones being represented with the least number of bits. Size reduction between original and compressed file is provided in Fig. 1.
In lossless compression, the primary disadvantage is slowness, Huffman coding being considerably slower than other used techniques. Another concern can be the lack of detection of data corruption when decompressing, as symbol codes have different length.

2.2 Lempel-Ziv-Welch (LZ77)

- **Algorithm description** LZ77 method removes recurring sequences by replacing them with pointers to their previous appearance. This technique searches each symbol in the already traversed "window". When going back and finding the symbol, the offset is defined as the difference of indexes between starting position and where it was previously encountered. From there, the algorithm searches for the maximum sequence that can be replaced and stores it's length. For first encounters, both offset and length are set to 0.

– **Storing compressed data** Compression is made reducing character representation by replacing a len(string) bytes sequence with pointer to previous encounter. In order to decompress, the code is traversed and sequences are restored by following back offset and length parameters. The implementation chosen stores one character on 3 bytes, corresponding to offset, length, and symbol. This technique actually expands individual symbols, size reduction being realised when finding longer sequences that are as well stored on 3 bytes. The main idea of this method is presented in Fig 2.

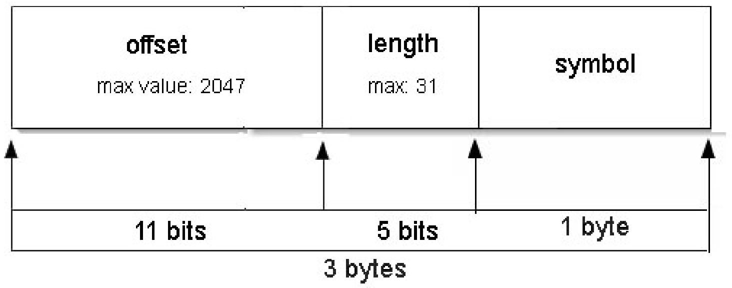

Fig.2 LZ77 coding compressed file

– **Algorithm complexity** Compression and decompression speed are different for LZ77 method:

Table.2 Relative Speed

Algorithm	Compression Speed	Decompression Speed
Dictionary-based	slow	fast

Since LZ compression methods require time to search the dictionary, compression is usually much more expensive than decompression. For each test, the algorithm can perform in O(n) for a text of n characters. Searching the dictionary for the longest string match makes the implementation really time consuming.

– **Main advantages and disadvantages** The primary advantage of this technique, besides a really efficient way of replacing redundant sequences with 3 bytes codes, is the extremely fast decompression.
On the other hand, the main disadvantage is the limitation in sequence

length, as it shall be represented on 5 bits (in the chosen implementation). This is based on the premise that a standard text won't have a recurring string longer than 31 symbols, leaving more space for offset (2047). This limitation can be changed by adding 1 byte to the whole structure, but for standard texts that are given to compression chosen values should be enough to result in a good compression ratio.

3 Evaluation

3.1 Test generation

In order to conduct a comparative analysis of the implementations, a set of 9 tests has been generated. These tests will be performed on Ubuntu distribution with AMD Ryzen 7 4700u.

These test are generated with ascii symbols only.
Each test differ in size an situation, offering the possibility to showcase program's behaviour by following time of execution, compression ratio and all evaluation criteria stated before.

First three test were generated using only one symbol in order to analyse behaviour when encountering only redundant patters ('*').

Test number 4 is a simple English phrase generated with OpenAi. This test showcases behaviour when encountering a small amount of information in a standard recurring situation.

Tests from 5 to 8 consist of random generated text using www.lipsum.com. These tests show algorithm behaviour when encountering large amount of text (form 1 to 500 KB).

Test number 9 is a small file with repeating sequences.

3.2 Results, Explanations and Interpretation

Table 1. Huffman coding results

Test Nr	Original file size	Huffman		
		Compression Time	Compression Size	Decompression Time
1*	100 bytes	0,02 s	20 bytes	0,02 s
2*	10 KB	0,03 s	1,3 KB	0,03 s
3*	50 KB	0,05 s	6,3 KB	0,06 s
4	100 bytes	0,02 s	206 bytes	0,02 s
5	1 KB	0,02 s	802 bytes	0,02 s
6	10 KB	0,04 s	6,3 KB	0,1 s
7	100 KB	0,05 s	55,6 KB	0,56 s
8	500 KB	0,15 s	276 KB	2,74 s
9	300 bytes	0,03 s	347 bytes	0,03 s

Table 2. LZ77 coding results

Test Nr	Original file size	LZ77		
		Compression Time	Compression Size	Decompression Time
1*	100 bytes	0,022 s	12 bytes	0,018 s
2*	10 KB	9,2 s	650 bytes	0,02 s
3*	50 KB	1m 8 s	3,2 KB	0,03 s
4	100 bytes	0,03 s	178 bytes	0,02 s
5	1 KB	0,05 s	853 bytes	0,02 s
6	10 KB	0,38 s	4,3 KB	0,023 s
7	100 KB	8,3 s	41,4 KB	0,07 s
8	500 KB	2 m 41 s	200 KB	0,5 s
9	300 bytes	0,03 s	274 bytes	0,02 s

From first three test, we observe that redundant information can be removed, resulting in a high saving percentage for both algorithms.

However, test number 4 shows that encoding data doesn't always result in space saving. For a small amount of information (100 Bytes), both methods require more space to store the encoded data, almost doubling the initial size.

The same can be stated following results from test 9, where only LZ77 coding generates less amount of data, as input has many repeating sequences.

Size increase comes with higher saving percentage, as seen in results of tests 5-8. LZ77 algorithm performed better on large input in terms of compression ratio, but is way less time efficient, going as high as 3 minutes execution time.

Testing larger amount of data became irrelevant for the comparison of the two techniques, as LZ77 will always get a higher saving percentage, but in amount of time exceeds execution limits. Huffman coding implementation however, did manage to compress files up to 15MB, resulting in some cases in 1 or 2 symbols lost.

Efficiency in terms of compression ratio, saving percentage and speed are showcased in tables 3 and 4:

Table 3.

Test Nr	Huffman		
	Compression Speed	Compression Ratio	Saving Percentage
1*	4,882 KB/s	20%	80%
2*	333,33 KB/s	13%	87%
3*	1000 KB/s	12,6%	87,3%
4	4,882 KB/s	206%	-106%
5	50 KB/s	78,3%	21,6%
6	250 KB/s	63%	36%
7	2000 KB/s	55,6%	44,3%
8	3333,33 KB/s	55,2%	44,7%
9	9,765 KB/s	115,66%	-15,66%

Table 4.

Test Nr	LZ77		
	Compression Speed	Compression Ratio	Saving Percentage
1*	4,438 KB/s	12%	88%
2*	1,086 KB/s	6,3%	93,6%
3*	0,735 KB/s	6,4%	93,6%
4	3,255 KB/s	178%	-78%
5	20 KB/s	83,3%	16,6%
6	26,315 KB/s	43%	57%
7	12,048 KB/s	41,4%	58,6%
8	3,105 KB/s	40%	60%
9	9,765 KB/s	91,3%	8,6%

4 Conclusion

Data compression techniques play a crucial role in handling large amounts of data, encoding making it possible to represent information in compact form. This kind of encoding is applied in various forms, resulting in reduced size for any kind of given input like audio, text or video. This paper provides a comparative analysis between two of the most widely used Lossless techniques for text compression, Huffman Coding and Lempel-Ziv-Welch (LZ77), testing a personal approach of implementation.

In practice, these methods represent ground for in use compressor implementations.

References

1. David A. Huffman: “A Method for the Construction of Minimum Redundance Codes.” (1952)
2. Ziv, J. and A. Lempel: “Compression of Individual Sequences Via Variable-Rate Coding.” (1978)
3. Matthew Simpson, Dr. Rajeev Barua and Surupa Biswas: Analysis of Compression Algorithms for Program Data (2003)
4. David Salomon: “Data Compression: The Complete Reference” (2004)
5. Neha Sharma, Usha Batra: Performance analysis of compression algorithms for information security: A Review (2020)
6. Dr. Nagamani, Sushruth Nagaraj, Sagar J, Shashank S, Tushar Kulkarni: “Comparative Analysis of Lossless Data Compression Techniques” (2022)