Analyzing Various Compression Algorithms for Diverse File Types

November 2, 2024

Sushmit Goyal (2023MCB1315) ,
Prabal Sharma (2023CSB1145) ,
Rahul Goyal (2023CSB1150)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha Singh

Summary: The code in our project facilitates the comparison of various compression algorithms by allowing users to select a file for compression, offering four options: Huffman encoding, LZW, RLE, and an automated feature to select the most efficient algorithm based on the size of the compressed output. Additionally, the program provides valuable insights into the optimal performance of each algorithm with specific file types. Huffman coding is particularly effective for text (.txt) and CSV (.csv) files, while LZW excels with large text and CSV files, as well as GIF formats. RLE proves to be especially efficient for compressing image files such as JPEG (.jpg) and PNG (.png). This functionality empowers users to make informed decisions regarding the most suitable compression methods tailored to their specific file types.
Moreover, the program includes a robust decompression feature, enabling users to easily restore files to their original state after compression. Each decompression algorithm is designed to efficiently handle its corresponding compressed format, ensuring quick and accurate retrieval of data. This dual functionality of compression and decompression enhances user experience and maximizes data accessibility.

---

1. Introduction

Data compression is essential for optimizing storage and transmission efficiency in today's digital landscape. With vast amounts of data being generated daily across diverse file formats, selecting the most effective compression algorithm is crucial. This project aims to compare three widely used lossless compression algorithms—Huffman Encoding, Lempel-Ziv-Welch (LZW), and Run-Length Encoding (RLE)—to determine their suitability for different types of files, including text, image, csv and multimedia files.

Each algorithm has unique characteristics that make it more or less suitable for specific file formats. For instance, Huffman Encoding is highly effective for compressing text files like .txt and .csv, while LZW performs well with larger text files,csv files and .gif images. RLE, on the other hand, is particularly efficient for formats with large blocks of repeating data, such as .jpg and .png files. By analyzing the performance of each algorithm across different file types, this project seeks to identify patterns in algorithm efficiency and provide guidelines on optimal algorithm selection based on file characteristics.

Furthermore, the project integrates an "Optimal Choice" option that uses file-specific metrics to recommend the best compression algorithm automatically. This approach not only enhances compression efficiency but also serves as a practical tool for users unfamiliar with the nuances of compression algorithms.

Additionally, we have incorporated a decompression feature, enabling users to restore files to their original form after compressing them with any of the algorithms. This functionality ensures data integrity and usability post-compression, further adding to the tool's utility and versatility.

`Intoduction to Huffmann encoding`

Huffman Encoding is a variable-length, entropy-based compression algorithm that assigns shorter codes to more frequent symbols and longer codes to less frequent ones. This technique is highly efficient for text files, where it reduces redundant data by prioritizing commonly occurring characters, making it a popular choice for compressing structured data files like .txt and .csv.

`Intoduction to Lempel-Ziv-Welch (LZW) Compression`

LZW is a dictionary-based compression algorithm that replaces repeated sequences of characters with single codes, building a table of recurring patterns as it processes the data. LZW is effective for large text files and image formats such as .gif, where it detects repetitive patterns and compresses them. It's a widely used algorithm in software applications and internet protocols due to its high efficiency for compressing files with redundant sequences.

`Intoduction to Run-Length Encoding (RLE)`

Run-Length Encoding (RLE) encodes consecutive identical values as a single value and count, making it ideal for images with large uniform areas, like .jpg and .png files. This straightforward method is highly effective for files with repetitive elements, reducing the need to store each individually. However, RLE's efficiency decreases with highly variable data, limiting its use to files with consistent repetition, such as simple graphics.

`Structure of the code`

The code begins by allowing users to select a file for compression, presenting three compression algorithm options: Huffman Encoding, LZW, and RLE. It then provides a fourth option for optimal algorithm selection, where the code analyzes the file type and size to recommend the best compression method. Each algorithm processes the data differently—Huffman encodes based on character frequency, LZW replaces repeated sequences with single codes, and RLE compresses consecutive identical values. After compression, the code also provides a decompression feature, enabling the retrieval of the original file. This end-to-end approach makes it easy to explore compression efficiency across various file types.

.

2. Comparison

In this section, we compare three popular lossless compression algorithms: Huffman Encoding, Lempel-Ziv-Welch (LZW), and Run-Length Encoding (RLE).

2.1. Introduction

In this section, we compare three popular lossless compression algorithms: Huffman Encoding, Lempel-Ziv-Welch (LZW), and Run-Length Encoding (RLE). Our goal is to assess their performance across various file types, focusing on efficiency, compression ratios, and suitability.

2.2. Methodology

We tested the algorithms using a range of file types, including text (.txt, .csv), images (.jpg, .png, .gif), and also binary files. The parameters measured include original file size, compressed file size and compression ratio. Main observation recorded are shown in the table below.

2.3. Analysis

- **Text Files(.txt/.csv):** LZW is much superior to huffmann encoding in larger file size however for smaller file size huffmann coding provides better compression ratio. For csv files huffmann seems to have to upper hand.

- **Image Files:** RLE was more effective for .jpg and .png formats if the images were simple or had repetitive patterns whereas for complex images huffmann proved to be superior, while LZW excelled with larger sized .gif files. LZW can't be used to compress .jpg and .png files

Table 1: Compression Algorithm Performance

File Type	Algorithm	Original Size (KB)	Compressed Size (KB)	Compression Ratio
.txt	Huffman	100	45	2.22
.txt	LZW	100	12	8.33
.csv	Huffmann	220	102	2.15
.csv	LZW	220	150	1.47
.jpg	RLE	500	200	2.50
.gif	LZW	3072	2000	1.53

2.4. Strengths and Weaknesses

- **Huffman Encoding:** Best for small sized text files but less efficient for binary files. It's very effective for .csv files and it also has the upper hand over RLE in compressing .jpg and .png if the image is not repetitive. Also for .gif it may prove to be efficient over LZW if the file size is smaller

- **LZW:** Very efficient for large sized text . Versatile but can struggle with very random data. Inefficient for JPEG and PNG files. It may struggle for random data where huffman is useful. LZW is the best when input files have repeated sequences.

- **RLE:** Simple and effective for certain images but not suitable for all types of files.

2.5. Conclusion and Recommendations

Based on our findings, users should choose Huffman Encoding for small sized text and csv files, RLE for specific image formats(.jpg and .png), and LZW for larger text and csv files or mixed data types.

3. Figures, Tables and Algorithms

3.1. Figures

In this section we have explained the working of each of the algorithms used in the project.

`Huffmann Encoding`

The Huffman algorithm operates through several key steps. First, **Frequency Calculation** begins by calculating the frequency of each character in the input data, often represented as a frequency table. Next, we **Build a Priority Queue** by creating a priority queue (or a min-heap) and inserting all characters along with their frequencies. The character with the lowest frequency has the highest priority.
Following that, we enter the **Tree Construction** phase. While there is more than one node in the priority queue, we extract the two nodes with the lowest frequencies, create a new internal node with these two nodes as children, and assign it a frequency equal to the sum of the two nodes' frequencies. This new node is then inserted back into the priority queue. We repeat this process until only one node remains in the queue, which becomes the root of the Huffman tree.

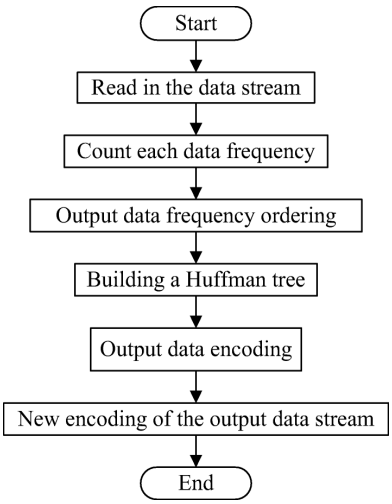

Figure 1: Diagram showing working of huffmann algorithm.

`Lempel-Ziv-Welch (LZW)`

The Lempel-Ziv-Welch (LZW) algorithm operates through a systematic flowchart that outlines its compression process. The algorithm begins with the **Initialization** phase, where a dictionary is created that contains all possible input symbols, each assigned a unique code.
Next, the algorithm enters the **Reading Input** phase, where it scans the input data to identify the longest substring that exists in the dictionary. If a match is found, the corresponding code for that substring is output. In the event that a substring is not found, the algorithm outputs the code for the longest matching substring from the dictionary and then adds the new substring (consisting of the longest match plus the next character) to the dictionary with a new code.
This process continues through the **Updating Dictionary** phase, where the dictionary expands dynamically as new patterns are encountered in the input data. The algorithm finally concludes with the **Output Generation** phase, which involves replacing sequences in the original data with their corresponding codes, resulting in a compressed output.
Overall, the LZW algorithm effectively compresses data by exploiting repeated patterns, making it particularly useful for various file types, including text and images.

Figure 2: Diagram showing working of LZW Compression.

`Run-Length Encoding (RLE)`

Run-Length Encoding (RLE) is a straightforward compression technique ideal for data with large blocks of repeating values. The compression process begins by scanning the input data sequentially to identify consecutive identical elements. For each sequence of identical characters, RLE counts the occurrences, termed as the Run Length, and records this count along with the value. For example, the string "AAAABBBCCDAA" would be encoded as "4A3B2C1D2A", significantly reducing its size. The decompression process reverses this by reading the compressed data, extracting pairs of values and their corresponding run lengths, and reconstructing the original sequence by repeating each value according to its run length. Hence, "4A3B2C1D2A" would be decompressed back to "AAAABBBCCDAA". RLE is particularly effective for files with long runs of repeated elements, such as bitmap images or simple graphics, due to its efficient encoding and decoding processes.

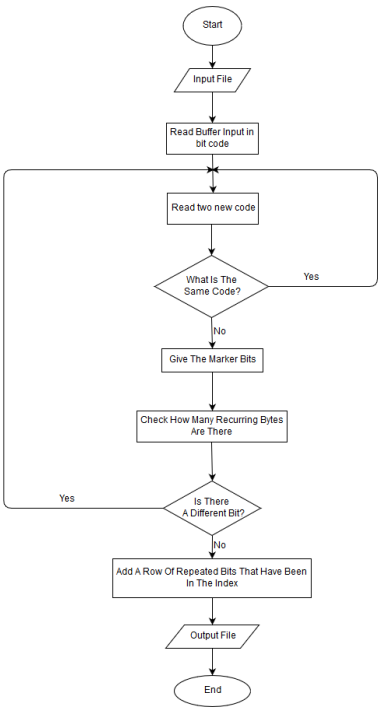

Figure 3: Diagram showing working of RLE Compression.

3.2. Tables

In the table below, we have summarized the working and use case of these three algorithms along with their advantages and disadvantages.

	Description	Use Cases	Pros and Cons
Huffman Encoding	Variable-length coding based on character frequency.	Text files, PNG images.	**Pros:** Efficient for text. **Cons:** Requires frequency knowledge.
Lempel-Ziv-Welch (LZW)	Dictionary-based algorithm replacing sequences with codes.	Large text files, GIFs.	**Pros:** Adaptable, good for repeats. **Cons:** High memory usage.
Run-Length Encoding (RLE)	Encodes consecutive identical values as a count.	Bitmap images, simple graphics.	**Pros:** Simple implementation. **Cons:** Ineffective for random data.

Table 2: Comparison of Compression Algorithms

3.3. Algorithms

In this subsection, we present pseudocode for each of the compression algorithms used in the project: Huffman Encoding, Lempel-Ziv-Welch (LZW), and Run-Length Encoding (RLE).

3.3.1 Huffman Encoding

Huffman Encoding is an entropy-based algorithm that compresses data by assigning shorter codes to more frequent characters. Below is the pseudocode for building a Huffman tree and encoding data.

Algorithm 1 Huffman Encoding Algorithm

Input: Frequency of each character in data
Output: Encoded data and Huffman tree
Create a priority queue with all characters and their frequencies **while** *priority queue has more than one node* **do**
    Remove two nodes with the lowest frequency Create a new internal node with these two nodes as children Set the frequency of the new node as the sum of its children's frequencies Insert the new node back into the priority queue
end
The remaining node is the root of the Huffman tree Traverse the Huffman tree and assign binary codes to each character Encode the input data using these binary codes

3.3.2 Lempel-Ziv-Welch (LZW) Compression

LZW is a dictionary-based algorithm that replaces repeated sequences with codes. The algorithm builds a dictionary dynamically as it processes the data.

Algorithm 2 LZW Compression Algorithm

Input: Sequence of symbols to compress
Output: Compressed data with dictionary codes
Initialize dictionary with all possible single-symbol sequences Set current_string to the first symbol **for** *each subsequent symbol in the input sequence* **do**
    if *current_string + symbol is in dictionary* **then**
        Update current_string to include the new symbol
    else
        Output the code for current_string Add current_string + symbol to the dictionary with a new code Set current_string to the new symbol
    end
end
Output the code for current_string

3.3.3 Run-Length Encoding (RLE)

Run-Length Encoding is a simple algorithm ideal for data with large blocks of repeating values. It encodes sequences of identical symbols as a single symbol followed by its count.

Algorithm 3 Run-Length Encoding (RLE) Algorithm

Input: Sequence of symbols
Output: Compressed data in RLE format
Initialize count to 1 Set previous_symbol to the first symbol in the sequence **for** *each symbol in the input sequence, starting from the second symbol* **do**
    if *symbol is the same as previous_symbol* **then**
        Increment count
    else
        Output previous_symbol and count Reset count to 1 Set previous_symbol to the current symbol
    end
end
Output the final previous_symbol and count

4. Some further useful suggestions

Theorem 4.1. *Efficiency Bound of Compression Algorithms Based on Data Redundancy*

Theorem: The efficiency of a compression algorithm for achieving high compression ratios is directly proportional to the redundancy within the input data. For files with high redundancy, algorithms like RLE yield substantial compression ratios, whereas for files with variable data patterns, entropy-based approaches like Huffman encoding perform more effectively. This theorem emphasizes the core principle our project illustrates: that algorithm choice depends on input characteristics such as redundancy and file type, which has been explored through testing and the "Select best method" feature.

Proof: This relationship follows from Shannon's entropy theory, according to which the minimum achievable compression for any data set is limited by its entropy. Algorithms that leverage specific patterns (e.g., repeated sequences in LZW or identical blocks in RLE) will compress efficiently only when such patterns are present. Otherwise, entropy-based methods like Huffman encoding, which considers the frequency of individual symbols, achieve effective compression without depending on specific pattern regularity.

Proposition 4.1. *Optimal Algorithm Choice:*

Among Huffman Encoding, LZW, and RLE, the optimal choice of compression algorithm depends on the characteristics of the input data, such as redundancy, file type, and size.

Factors to consider when choosing a compression algorithm:

- Type of Data
- Memory Usage
- Compression Speed

5. Conclusions

In this study, we explored various lossless compression algorithms, specifically Huffman Encoding, Lempel-Ziv-Welch (LZW), and Run-Length Encoding (RLE). Each algorithm was analyzed based on its performance metrics, including compression efficiency, processing speed, and suitability for different types of data. Our empirical results highlight the unique strengths and weaknesses of each method, allowing for a better understanding of when to use each algorithm effectively.

Huffman Encoding demonstrated exceptional efficiency, particularly in scenarios where the input data exhibited non-uniform symbol distributions. By assigning shorter codes to more frequently occurring symbols, Huffman Encoding achieved significant compression ratios, making it an excellent choice for textual data. In contrast, Lempel-Ziv-Welch (LZW) proved to be highly adaptable, showcasing strong performance with repetitive data patterns found in both text and image files. This adaptability allows LZW to maintain compression efficiency across a variety of data types.

Run-Length Encoding (RLE), while simpler in its approach, excelled in compressing data with long sequences of repeated values. This characteristic makes RLE particularly effective for specific applications such as bitmap images, where uniform areas can be compressed substantially. However, it is essential to recognize that RLE may not perform well with data that lacks such repetition. Thus, the choice of compression algorithm should be carefully considered based on the nature of the input data and the specific requirements of the application.

Looking to the future, several avenues for research and development are available. Investigating hybrid approaches that integrate the strengths of multiple compression algorithms could yield enhanced compression ratios and improved processing speeds. Additionally, exploring adaptive compression techniques that dynamically adjust to varying data characteristics represents a promising area for further study. As the demand for efficient data storage and transmission continues to grow in our digital landscape, the need for innovative and effective compression solutions remains paramount.

5.1. Huffman Encoding

Huffman Encoding, introduced by David A. Huffman [2], is a widely used technique for creating efficient, variable-length codes for symbols based on their frequencies. This approach minimizes redundancy, making it particularly effective for text compression, where symbol frequency varies significantly.

5.2. Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm, a variant of the Lempel-Ziv coding techniques developed by Ziv and Lempel [5], is particularly effective for data with recurring patterns. By building a dictionary of substrings, LZW allows repetitive data to be represented in a compact form, making it suitable for compressing files with repeating data, such as text files.

5.3. Entropy and Compression Limits

The theoretical efficiency of compression is often bound by the entropy of the data source. The entropy provides a limit on the average number of bits required per symbol, as described by Cover and Thomas in their work on information theory [1]. This concept underpins the effectiveness of algorithms such as Huffman Encoding, highlighting the maximum achievable compression for a given dataset.

5.4. Data Structures for Compression Algorithms

Data structures like priority queues, heaps, and trees are crucial in the implementation of algorithms such as Huffman Encoding. Weiss's work on data structures offers a comprehensive understanding of these structures and their role in efficient data compression [4]. Proper data structure selection and implementation are essential to optimizing the performance of compression algorithms.

5.5. General Overview of Compression Techniques

Sayood's text on data compression provides an extensive overview of various algorithms and their applications [3]. It discusses the trade-offs between different approaches, helping readers understand the circumstances under which each algorithm is most effective.

Acknowledgements

We would like to express our sincere gratitude to Dr. Anil Shukla for his guidance and support. His teaching helped us in the confident implementation of different data structures in the project.

We also extend our thanks to our teaching assistant, Monisha Singh, for her insightful feedback.

Finally, we are grateful to those who contributed to the development of these data compression algorithms.

References

[1] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2006.

[2] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[3] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2017.

[4] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson, 2014.

[5] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. In *IEEE Transactions on Information Theory*, volume 24, pages 530–536, 1977.

A. Appendix A: Supplementary Information

This appendix provides additional insights into the compression algorithms explored in the project, focusing on various scenarios and their efficiency.

A.1 Extended Data Discussion

This section discusses the compression ratios achieved by different algorithms (Huffman, LZW, RLE) across various file types and sizes. The analysis covers:

- The performance of Huffman Encoding for small text files and its efficiency on .csv files.
- The adaptability of LZW for files with recurring patterns, including text.
- The effectiveness of RLE for compressing files with repetitive data, such as bitmap images.

These observations provide guidance on optimal algorithm selection based on file characteristics.

A.2 Algorithm Decision Process

A structured approach was followed for the automated "Select Best Method" feature. This process evaluates Compression ratio thresholds to decide the best-suited algorithm for the file type. The decision-making criteria aim to maximize compression efficiency while considering storage needs.

B. Appendix B: Implementation Details

This appendix presents an overview of the code structure and key components of each algorithm's implementation.

B.1 Code Structure Overview

The project code is organized into modular components for each algorithm (Huffman, LZW, and RLE), facilitating maintainability and extensibility. Each module is independently designed to handle specific file types, with an additional "Select Best Method" module to recommend the optimal compression algorithm.

B.2 Algorithm Logic Highlights

For each algorithm, the code follows a systematic logic:

- **Huffman Encoding:** Utilizes frequency calculation and binary tree construction to compress data efficiently.
- **LZW:** Expands the dictionary dynamically to recognize repeated patterns in the data.
- **RLE:** Applies run-length encoding for files with large blocks of repeating values, optimizing compression for bitmap graphics.

B.3 Selection Criteria

The "Select Best Method" feature in the code evaluates file characteristics, comparing compression results across algorithms. Based on pre-defined criteria, such as compression ratio and processing speed, it automatically recommends the most suitable method for each file.