Project Report

Youssef Ayman Hassan - Adham Elrouby - Hassan Ashraf -
Mohamed Elsayed - Mohamed Hazem Ali

December 7, 2024

§1 Abstract

This project presents a file compression system using Huffman coding to reduce file sizes while preserving data accuracy. The system performs the effective implementation of binary trees and priority queues by allocating shorter binary codes to the higher frequency characters. It can handle text files of large size, so it is scalable, flexible, and reliable. This project emphasizes how advanced data structures are used in real-world challenges involving file storage and transmission. Future development might entail the addition of efficient performance with very large datasets and additional data format support.

§2 Introduction

File compression plays a pivotal role in modern computing, enabling efficient storage and transmission of data. This project implements a file compression and decompression system based on Huffman coding, a lossless compression algorithm that assigns shorter binary codes to characters with higher frequencies. The system is built around the construction of Huffman trees using priority queues, ensuring efficient encoding and decoding of text files while preserving data integrity.

The primary goals of this project include achieving efficient compression, ensuring scalability for large datasets, and maintaining the reliability and flexibility needed for various use cases. By leveraging advanced data structures, this project highlights a practical application of computer science principles to address challenges in file storage and transmission.

This report details the implementation and evaluation of the proposed system, demonstrating its effectiveness in reducing file sizes while ensuring lossless recovery of the original data.

§3 Problem Definition

Handling large data volumes is a challenge in computing. While popular compression tools perform well in many cases, they often struggle with performance on large files, adaptability across different use cases, or fine-grained customization.
This project aims to address these limitations with the following goals:

- **Efficient Compression:** Minimize file sizes while preserving the exact original data.

- **Scalability:** Ensure the system can handle large datasets efficiently.
- **Flexibility:** Make the tool adaptable for various data formats and use cases.
- **Reliability:** Guarantee lossless decompression.

By focusing on these objectives, the project delivers a strong compression system that combines theoretical efficiency with practical utility.

§4 Methodology

§4.1 Encoding Methodology

1. Calculate Character Frequencies.
2. Build the Huffman Tree using the min-heap.
3. Assign Binary Codes for each character according to its frequency.
4. Encode the Input by replacing each character with its representative binary code.
5. Write Encoded Data by dividing the binary string into 8-bit chunks for storage and padding any leftover bits with zeros.

§4.2 Decoding Methodology

1. Read the Encoded Data from the compressed file.
2. Decode the Binary Data using the Huffman tree.
3. Remove Padding.

§4.3 Compression Methodology

1. Verify if the input is a .txt file.
2. Build the Huffman tree using the character frequencies from the input file.
3. Construct the header with character and Huffman code information.
4. Convert the file content into a binary string using the Huffman codes.

§4.4 Decompression Methodology

1. Extract the header and Huffman codes from the compressed file.
2. Use the header to reconstruct the Huffman tree.
3. Convert the binary string back into the original text using the Huffman tree.
4. Ensure proper data extraction by removing any padding.

§5 Specifications of Algorithms to be used

§5.1 Huffman Tree

First, to construct the Huffman tree, we will follow the following algorithm:

1. **Initialize the priority queue:** Every character which has a frequency not equal to zero becomes a leaf. The priority queue sorts those nodes according to their frequency; the lowest frequency is on top of it, using the custom comparator.
2. **Merge nodes:**
   - Two nodes are taken away one after another with a low frequency
   - Create a new inner node and its frequency being the sum of its previous two nodes.
   - Throw it back in the queue again.
3. **Set Root:** When only one node is left in the queue, that becomes the root of the Huffman Tree.
4. **Generate Codes:** Now, use the following recursive generate_codes function in order to generate the binary codes for each character using the root specified in the previous step.

Then, to generate the binary mask of each character, we will use the following algorithm. It takes the current node, the root will be inserted in the first call, and a string representing the binary code built so far. Then,

1. **Check if the node is a leaf:**
   - If the node is NULL, then it stops the base case.
   - If it is a leaf node, the character is not the special value 127 used for internal nodes, it saves the binary code for this character in the huffman_codes array.
2. **Recursive calls:**
   - Appending '0' to the code while going left.
   - Appending '1' when going right.

Using the previous algorithms of creating a Huffman tree, it becomes possible to encode and decode the characters in any text using the following encoding and decoding algorithms.

§5.2 Encoding algorithm

1. Create Temporary Storage
2. Iterate Over Each Character in originalText. For each character in originalText:
   - Its ASCII value is retrieved.
   - In case this character is within the valid range of ASCII (0 - 127), its corresponding Huffman code is appended to the string.
3. Convert Binary Data in Chunks of 8 Bits
   - The function processes the accumulated binary string s:

- Slices off the first 8 bits (byteStr).
- Converts this binary string into its decimal equivalent
- Converts the decimal into character and appends to it.
- Removes the processed 8 bits from the string.

4. Handle Remaining Bits
   - After processing all full bytes, whatever remaining bits (less than 8) are padded with zeros to make them a full byte.
   - This padded byte is converted to a character and appended to in.
5. Store Padding Information
   - The number of padding bits count is stored.
   - The process also ensures that during decoding, the exact number of valid bits in the last byte can be identified, together with the removal of padding.

§5.3 Decoding algorithm

1. Initialization:
   - decodedText: An empty string that shall store the final decoded text.
   - i: A pointer for the current position in the encoded text (codedText).
2. Iterative Decoding: The function goes through the encoded text character by character, and in each iteration:
   - Bits are appended one after the other to the current string from encoded text.
   - At each bit addition, the function calls linearSearchKey to verify whether the so far accumulated current string matches with any Huffman code.
3. Matching Code:
   - In case of a match -linearSearchKey returns a valid index, the character corresponding to the index is appended to decodedText.
   - The while-loop exits as the current Huffman code is completely decoded.
4. Update Position: Move position i to j + 1, so the next portion of the encoded text will be processed for decoding by the function.

§5.4 Compression

First, the algorithm checks if the input file is .txt. If it is not, the program outputs an error message and stops the algorithm. Otherwise, we read the input file and generate a frequency array that is used as an argument to generate the Huffman tree and map. Afterward, we use the Huffman Map to generate the header structure that will be used in decompression later on. In the header structure, the first byte is the number of characters used in the Huffman map (let it be n). The following segments consist of three parts:

1. Character Byte (the letter)
2. Huffman Code Size Byte (S)
3. Huffman Code encoded in ceil(S/8.0) bytes

Afterward, the content of the input file is converted into a large sequence of a binary string according to the Huffman codes for each character in the map. Afterward, The file content is encoded as described in the encoding algorithm.

§5.5 Decompression

Similarly, the algorithm first checks if the input file is .hassan. If it is not, the program gives an error message and stops the algorithm. Otherwise, the Huffman Map is first regenerated by reversing the process described in the compression algorithm and moving along the specified header structure. After that, the file content bytes are all converted to a binary string that is then processed to regenerate the original text. During the process of the binary string, we check if the current sequence of the binary string corresponds to an existing binary string in the Huffman Map (Note: this is true since in Huffman Tree, no two characters' codes have the same prefix). After retrieving the original text, a .txt file is created with the original content.

§6 Experimental Results

We made five different text files with varying sizes to test our compression algorithm. For each, we calculated the compression ratio (CR), which is defined as the following:

$$CR = \frac{\text{Size of the Compressed File}}{\text{Size of the Original File}} \times 100\%$$

Table 1 illustrates the sizes of the test cases and the corresponding compressed files and compression ratios.

Table 1: Test Results

Test case	Original Size (KB)	Compressed Size (KB)	Compression Ratio
T1	0.895	0.638	71.3%
T2	76.7	42.7	55.7%
T3	230	127	55.2%
T4	446	245	54.9%
T5	1613	895	55.4%

Figure 1 shows a graphical relationship between the input text file size and the output compression ratio.

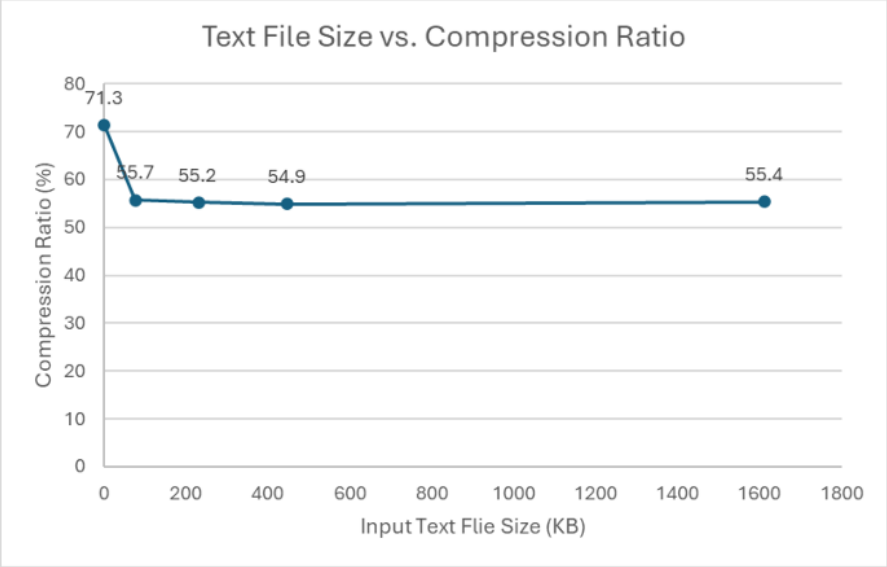

Figure 1: The relationship between text file size and the compression ratio

§7 Analysis and Critique

For the Huffman tree data structure, we represent the character held by internal nodes by the DEL character (ASCII code: 127). This is because it is not a printable character, so it will never need a Huffman code string.
For analyzing the compression/decompression algorithm, we find that the Huffman Map is an array of 128 strings, where each index represents the ASCII code for the 128 ASCII characters. Therefore, if a file contains a character that is outside those 128 characters or, for instance, a different language, then the compressed file will not take these into account, and these characters will not be retrieved after decompression. It would be difficult to account for characters outside of the 128 ASCII characters because some of these characters require a combination of two or more ASCII characters, which would require a significant change in the data structures used, although the algorithm will be the same.
If we measured the average compression ratio using the five test cases, we find that:

$$CR_{avg} = \frac{71.3 + 55.7 + 55.2 + 54.9 + 55.4}{5} = 58.5\%$$

However, we notice that the compression ratio at small file sizes is relatively large while starting from a threshold of 76.7 KB file size, the compression ratio (CR) stabilizes. This makes sense because the header used to regenerate the Huffman map during decompression takes a relatively fixed size regardless of the file input size. This fixed size becomes very small compared to the input file text when the input text file size increases, therefore, when we measure the average compression ratio after a threshold of 76.7 KB, then:

$$CR_{avg} = \frac{55.7 + 55.2 + 54.9 + 55.4}{4} = 55.3\%$$

This average compression ratio remains approximately the same for larger input file texts.

§8 Conclusion

The implementation of Huffman coding in this project demonstrates its effectiveness in file compression. By using binary trees and priority queues, the system achieves significant size reductions and handles large text files efficiently. The tool's adaptability and reliability make it suitable for diverse applications. Future work could explore optimizing algorithms for extreme-scale datasets or expanding support for more data types, further increasing its practical applicability in modern computing.

§9 Acknowledgements

We would like to express our sincere gratitude to Dr. Ashraf for his continuous support, invaluable guidance, and insightful feedback throughout the course of this project. His expertise and encouragement were instrumental in shaping our approach and ensuring our success.
We also extend our thanks to Dr. Dina for her constructive feedback, which helped us refine our work and overcome challenges.
A special thank you to Eng. Mohamed Ibrahim, our teaching assistant, for his unwavering support and guidance. His assistance in addressing technical issues and providing helpful advice greatly contributed to the completion of this project.
Lastly, we are grateful for the opportunity to work on this project and for the collaborative effort of all team members.

§10 References

[1] M. A. Weiss, Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2013. ISBN 978-0132847377.
[2] GeeksforGeeks, "Huffman Coding — Greedy Algo-3," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.
[3] J. Zelenski, K. Schwarz, and M. Stepp, "Huffman Encoding and Data Compression," Stanford University, [Online]. Available: Link

§11 Appendix

The code for the project can be found in this GitHub repository.