CSCE 2211 Term Project Spring 2025

Kareem Abdellatif – Department of Computer Science and Engineering, The American University in Cairo, Cairo, Egypt – kareemelramly@aucegypt.edu

Shahd Elfeky – Department of Computer Science and Engineering, The American University in Cairo, Cairo, Egypt – shahdelfeky@aucegypt.edu

Mariam Abd Alrahim – Department of Computer Science and Engineering, The American University in Cairo, Cairo, Egypt – mariamibrahim05@aucegypt.edu

Abstract

This project implements and evaluates a lossless text compression system using the Huffman coding algorithm. The core objective is to reduce file sizes by addressing the inefficiency of fixed-length character encodings. A greedy algorithm constructs an optimal prefix tree based on character frequency, assigning shorter binary codes to more frequent symbols. The system features a custom-built backend, including manually implemented data structures, and a Qt-based graphical interface for usability. Performance is mainly evaluated on multiple text files using compression ratio. The results confirm Huffman coding's effectiveness, with an average of 69% in compression ratio. This demonstrates significant size reduction while guaranteeing perfect data reconstruction, validating its utility for efficient data storage and transmission.

Keywords: Huffman Coding, Lossless Compression, Compression Ratio, Data Structures

1.1 Introduction

In the digital age, efficient data storage and transmission remain critical challenges. Text compression directly addresses this by reducing file sizes, saving storage space, and accelerating data transfer. Among various techniques, lossless compression is essential for textual data, as it allows for the exact reconstruction of the original information. This project explores Huffman coding, a foundational and elegant algorithm in this domain. Developed by David A. Huffman in 1952, it employs a greedy strategy to create an optimal prefix code, ensuring no ambiguity during decoding. By implementing this algorithm from first principles, the project delves into core computer science concepts, including priority queues and binary tree traversal. The work serves as a practical application of theoretical data structures and algorithms to solve a real-world problem.

1.2 Problem Definition

The core problem is that standard text encoding schemes like ASCII use a fixed number of bits per character, leading to significant redundancy. For instance, in typical English text, the letter 'e' appears far more frequently than 'q', although both consume identical storage space. This inefficiency becomes pronounced in large documents. The problem, therefore, is to design and implement a compression system that minimizes this redundancy by creating a variable-length encoding where common characters are represented by shorter bit sequences. The Huffman algorithm solves this by constructing a binary tree from the bottom up based on symbol frequencies. The primary technical challenge lies in correctly implementing the tree construction, generating a prefix-free code map, and serializing and deserializing the compressed data to enable flawless decompression, all while evaluating the system's performance based on compression ratio and saving ratio.

1.3 Methodology

The problem addressed in this project is text compression. The methodology followed a structured approach consisting of algorithm selection, manual implementation, interface design, and evaluation.

1.3.1 Algorithm Selection

First, it was important to identify an efficient algorithm for the task. We evaluated four potential options:

- Huffman Tree
- Arithmetic Coding
- Run-Length Encoding
- Bit Reduction

The Huffman Tree was chosen because it balanced efficiency and performance, in addition to fulfilling the project requirement of providing a non-linear data structure.

1.3.2 Implementation

To comply with project constraints, all data structures required for the Huffman algorithm were implemented manually. This included:

- A custom Vector class.
- A Minimum Heap priority queue.

Following the backend implementation, the Qt framework was used to provide a Graphical User Interface (GUI) for the program, ensuring user accessibility.

1.3.3 Testing and Evaluation

Finally, the program was tested on seven different files. The performance was measured using the following evaluation metrics:

- **Compression Ratio:** This metric measures the ratio of the compressed file size to the original file size.

$$CompressionRatio = \frac{Size_{after}}{Size_{before}} \tag{1}$$

- **Saving Ratio:** This represents the percentage reduction in file size, calculated as the difference relative to the original size.

$$SavingRatio = \frac{Size_{before} - Size_{after}}{Size_{before}} \tag{2}$$

- **Bits Per Character (BPC):** This indicates the average number of bits required to represent a single character in the compressed file.

$$BPC = \frac{TotalBits_{newfile}}{NumberofCharacters_{oldfile}} \tag{3}$$

To conclude the analysis, the average value for each evaluation metric was calculated across the test cases.

1.4 Specification of Algorithms to be Used

The compression and decompression logic is built upon the Greedy Huffman Coding algorithm. The core concepts and greedy strategy used in this implementation are based on the approach described in "Huffman Coding — Greedy Algo-3" by GeeksforGeeks [2].

The system uses two primary algorithmic procedures: one for constructing the optimal prefix tree during compression, and another for reconstructing the tree from the header during decompression.

1.4.1 Compression Algorithm: Tree Construction

To generate the optimal prefix codes, A Min-Heap (Priority Queue) is used to build the Huffman Tree from the bottom up.

1. **Frequency Calculation:** Read the input file byte-by-byte and calculate the frequency of every ASCII character (0-255).
2. **Heap Initialization:** Create a `Leaf Node` for every character with a non-zero frequency and insert it into the custom `Min-Heap`.
3. **Tree Building (Greedy Strategy):**
   - While the Heap size is greater than 1:
   - Extract the two nodes with the smallest frequencies (Min_1 and Min_2).
   - Create a new internal node with frequency equal to $Min_1.freq + Min_2.freq$.
   - Set Min_1 as the left child and Min_2 as the right child.
   - Insert the new internal node back into the Heap.
4. **Code Generation:** Perform a recursive traversal (DFS) of the resulting tree:
   - Assign '0' for every left traversal and '1' for every right traversal.
   - When a leaf is reached, map the accumulated binary string to that character in the `CodeMap`.

1.4.2 Decompression Algorithm: Tree Reconstruction

Unlike standard implementations that might serialize the tree structure directly, our approach stores the *Character-to-Code* mapping in the file header. The decompression algorithm reconstructs the tree by inserting these codes into a prefix tree structure.

1. **Header Parsing:** Read the file header to populate the `CodeMap` with the original character-code pairs.
2. **Tree Reconstruction:**
    - Initialize a Root Node.
    - For each character in the `CodeMap`:
    - Traverse from the Root using the binary code string (0 = Left, 1 = Right).
    - If a child node does not exist on the path, create a new internal node.
    - At the end of the code string, mark the node as a Leaf and store the character data.
3. **Decoding Stream:** Read the compressed binary stream bit-by-bit, traversing the reconstructed tree from the Root. When a Leaf Node is reached, output the character and reset traversal to the Root.

1.5 Data Specifications

The system is designed to handle specific input formats and uses custom internal structures to process and store data efficiently.

1.5.1 Input Data

The application accepts the following input file formats:

- **Text Files (.txt):** Standard ASCII text files containing alphanumeric characters and symbols.
- **Document Files (.docx):** Word document files, processed as binary streams.

1.5.2 Output Data

The output is a binary file containing two distinct sections:

1. **Header:** A serialized representation of the CodeMap, storing pairs of characters and their corresponding Huffman codes to enable reconstruction during decompression, and the maximum number of bits per character.
2. **Payload:** The compressed bit stream, padded with zeros to ensure byte alignment.

1.5.3 Test Files

Seven files are compressed and then compressed to evaluate the performance of the program.

File	Original Size (Bytes) (%)
File 1	106,264,576
File 2	16,384
File 3	2,560
File 4	191,488
File 5	10,240
File 6	2,167,808
File 7	448

Table 1: Information of the test files.

1.6 Experimental Results

The Huffman coding implementation was tested on seven text files ranging from 448 bytes to 106 MB. Performance metrics include compression ratio, saving percentage, bits per character (BPC), and encoding/decoding times. Table 2 summarizes the results across all test cases.

File	Original Size	Comp. Size	BPC	Saving (%)	Ratio (%)	Enc (ms)	Dec (ms)
File 1	106,264,576	57,622,528	4.34	45.78	54.23	14,100	8,000
File 2	16,384	9,216	4.50	43.75	56.25	20	29
File 3	2,560	2,243	7.01	12.40	87.62	140	16
File 4	191,488	100,352	4.19	47.59	52.41	149	85
File 5	10,240	7,168	5.60	30.00	70.00	637	13
File 6	2,167,808	1,146,880	4.23	47.10	52.91	589	353
File 7	448	504	9.00	-12.50	112.50	5.7	11.5
Average	-	-	**5.55**	**30.59**	**69.41**	**2,234**	**1,215**

Table 2: Huffman Tree Performance Metrics

The key findings from the experiments show an average saving percentage of 30.59%, which demonstrates a substantial size reduction for most files. The average bits per character (BPC) is 5.55, compared to 8 bits in standard ASCII. The best performance was reached by Files 1, 2, 4, and 6, achieving savings between 43% and 48%. The encoding time averaged 2,234 milliseconds and increased with file size, while the decoding time was faster, averaging 1,215 milliseconds due to efficient prefix tree traversal. Larger files like File 1 and File 6 consistently showed low compression ratios from 55 to 52%, validating the algorithm's scalability. Smaller files showed more variability in results because the overhead from the header dominated the compressed data size.

1.7 Analysis and Critique

The results confirm the effectiveness of Huffman coding for English text that has non-uniform character distributions, achieving near-optimal prefix codes through a greedy tree construction approach. The average saving of 30.59% outperforms fixed-length encoding while ensuring perfect, lossless reconstruction. However, File 7 shows a negative saving of -12.50%, highlighting overhead issues where the codebook header that stores character-code mappings is larger than the actual payload savings for very small files under 500 bytes.

There are clear strengths in the implementation. The custom min-heap and vector classes handled large-scale tree construction efficiently, even for files as large as 106 MB. The Qt GUI adds practical usability advantages compared to command-line tools. Additionally, the use of prefix-free codes guarantees unambiguous, streaming decompression.

Nonetheless, some limitations and critiques must be considered.The absence of preprocessing techniques, such as bit reduction or pre-run-length encoding, misses an opportunity to improve compression ratios further as observed in other studies. Also, the necessity to store the full frequency table and code map in memory restricts the system's ability to handle extreme-scale files effectively. The anomaly observed with File 7 suggests that header serialization needs to be optimized with awareness of input size, to better handle small files.

1.8 Conclusions

The project implements the File Zipper application using the Huffman tree, a lossless algorithm that is suitable for compressing and decompressing text files. The program showed high performance in compressing text, as it provided an average of 30.9% in saving ratio and an average of 69% in compression ratio. It also provided an average of 5.5 bits per character. And for the speed, the program provided an average of 19.31 and 5.21 per KB for encoding and decoding, respectively.

To improve the program, the STL library is recommended to be used, as it is more optimized. In addition, the algorithm can be enhanced by adding bit-reduction preprocessing before the compression, which showed a higher compression ratio[1]. Finally, the program can be hosted on a website instead of QT for a better user experience.

References

[1] A. S. Sidhu and E. M. Garg, "Research Paper on Text Data Compression Algorithm using Hybrid Approach," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 12, pp. 1–10, Dec. 2014.

[2] GeeksforGeeks. "Huffman Coding — Greedy Algo-3." *GeeksforGeeks*, [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. [Accessed: Dec. 06, 2025].

Appendix

https://github.com/kareemelramly/File-Zipper-Project (The source code of the project)