A

Project Report

On

Text File Compression-Decompression

Submitted in fulfillment of the requirement for the degree of

Bachelor of Technology

In

Computer Science and Engineering

By

Manish Singh Rautela	**2261349**
Govind Singh Kathayat	**2261236**
Pulkit Singh Bora	**2261448**
Sagar Singh Khanka	**2261503**

Under the Guidance of

Mr. Anubhav Bewerwal

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS

SATTAL ROAD, P.O. BHOWALI,

DISTRICT- NAINITAL-263132

2024-2025

STUDENT'S DECLARATION

We, **Manish Singh Rautela and Group** hereby declare the work, which is being presented in the project, entitled '**Text File Compression-Decompression**' in fulfillment of the requirement for the award of the degree **Bachelor of Technology (B.Tech.)** in the session **2024-2025**, is an authentic record of my work carried out under the supervision of **Mr. Anubhav Bewerwal.**
The matter embodied in this project has not been submitted by me for the award of any other degree.

Date:

Manish Singh Rautela

Govind Singh Kathayat

Pulkit Singh Bora

Sagar Singh Khanka

CERTIFICATE

The project report entitled "Text File Compression-Decompression" being submitted by Manish Singh Rautela(2261349) S/o Dalip Singh Rautela ,Govind Singh Kathayat(2261236) S/o Bhupal Singh Kathayat , Pulkit Singh Bora(2261448) S/o Bhupendra Singh , Sagar Singh Khanka(2261503) S/o Surendra Khanka , of B.Tech.(CSE) to Graphic Era Hill University Bhimtal Campus for the award of bonafide work carried out by them. They have worked under my guidance and supervision and fulfilled the requirement for the submission of a report.

Mr. Anubhav Bewerwal
(Project Guide)

Dr. Ankur Singh Bisht
(Head, CSE)

ACKNOWLEDGEMENT

We take immense pleasure in thanking the Honorable Director **'Prof. (Col.) Anil Nair (Retd.)',** GEHU Bhimtal Campus to permit me and carry out this project work with his excellent and optimistic supervision. This has all been possible due to his novel inspiration, able guidance, and useful suggestions that helped me to develop as a creative researcher and complete the research work, in time.

Words are inadequate in offering my thanks to GOD for providing me with everything that we need. We again want to extend thanks to our president **'Prof. (Dr.) Kamal Ghanshala'** for providing us with all infrastructure and facilities to work in need without which this work could not be possible.

Many thanks to **'Dr. Ankur Singh Bisht'** (Head, Department of Computer Science and Engineering, GEHU Bhimtal Campus), our project guide **'Mr. Anubhav Bewerwal '** (Assistant Professor, Department of Computer Science and Engineering, GEHU Bhimtal Campus) and other faculties for their insightful comments, constructive suggestions, valuable advice, and time in reviewing this report.

Finally, yet importantly, We would like to express my heartiest thanks to our beloved parents, for their moral support, affection, and blessings. We would also like to pay our sincere thanks to all my friends and well-wishers for their help and wishes for the successful completion of this project.

Manish Singh Rautela	**2261349**
Govind Singh Kathayat	**2261236**
Pulkit Singh Bora	**2261448**
Sagar Singh Khanka	**2261503**

Abstract

In the modern digital landscape, the efficient storage and transmission of data are critical challenges. One of the most effective solutions to address this issue is data compression, particularly lossless compression techniques which ensure that no data is lost during the compression and decompression processes. This project presents the design and development of a web-based application that performs text file compression and decompression using the Huffman Coding algorithm — a well-established and optimal method for entropy-based lossless compression.

The system is implemented entirely using client-side technologies: HTML for structure, CSS for styling, and JavaScript for logic and algorithm execution. At its core, the project uses custom implementations of fundamental data structures including MinHeap and Binary Trees to construct a Huffman Tree. This tree is used to generate unique binary codes for each character based on their frequency in the input file. More frequent characters are assigned shorter codes, while less frequent characters receive longer codes, thus minimizing the overall size of the file.

The user interface allows users to upload a .txt file, which is then parsed and analyzed. Upon clicking the "Compress" button, the application calculates character frequencies, builds the Huffman Tree, generates the encoded output, and displays compression statistics such as original size, compressed size, and compression ratio. Users can download the compressed file in text format. Similarly, the "Decompress" functionality allows users to restore the original file from its compressed version using the stored Huffman Tree structure.

One of the key advantages of this project is that all processing is done locally in the browser. There is no need for backend support or software installation, making the tool platform-independent, secure, and lightweight. It offers an intuitive user experience while providing real-time performance feedback, making it suitable for both educational and practical purposes.

The project also emphasizes the application of theoretical computer science principles in real-world scenarios. It effectively bridges the gap between academic learning (e.g., data structures, algorithms, software engineering principles) and practical implementation in modern web development environments.

Although the current version is limited to plain text files and lacks advanced features like bit-level encoding, encryption, or multi-format support, it sets a strong foundation for future enhancements. Potential upgrades include support for various file types, persistent Huffman Tree storage, encrypted compression, and improved UI features.

In conclusion, this project showcases how classical algorithms such as Huffman Coding can be successfully integrated into accessible, browser-based applications. It serves as a practical demonstration of how efficient data handling, user-centric design, and algorithmic optimization can work together to solve everyday problems related to data compression and storage.

TABLE OF CONTENTS

INTRODUCTION

1.1 Prologue

In the modern digital world, the efficiency of data storage and transmission plays a pivotal role in computing systems. As the volume of text-based data continues to grow, optimizing storage and ensuring fast transmission without loss of data is increasingly important. Compression is a key solution to this challenge. This project presents a **web-based application** that performs **lossless text file compression and decompression** using the **Huffman Coding algorithm** — an optimal technique well-known for its simplicity and efficiency in minimizing file size without sacrificing data integrity.

1.2 Background and Motivations

The concept of **data compression** has been a core part of software engineering and operating systems for decades. Efficient file handling mechanisms reduce storage overhead, network load, and processing time — critical for both users and systems. Huffman Coding, a **lossless compression algorithm**, assigns variable-length codes to characters based on their frequencies. This ensures more frequent characters use fewer bits, thereby reducing overall file size.

Motivated by the need for an intuitive, browser-based tool for compressing .txt files, this project was developed to help users compress and decompress files efficiently using **client-side JavaScript**, removing the need for server-side dependencies or installations. The integration of **heap data structures**, **binary trees**, and **encoding-decoding logic** offers a hands-on application of theoretical concepts from software engineering and operating systems.

1.3 Problem Statement

Conventional file storage and sharing can be inefficient, especially when dealing with large volumes of text data. Although there are existing compression utilities, many:

- Require installation or platform dependency,
- Are not tailored for .txt files,
- Do not visualize or explain the algorithmic process.

Thus, there is a need for a lightweight, **platform-independent** solution that allows users to:

- Compress .txt files using **lossless Huffman encoding**, and
- Decompress them back to original form without data loss.

This project addresses the following core problem:

How can we design and implement a web-based, user-friendly tool that performs efficient text file compression and decompression using Huffman coding?

1.4 Objectives and Research Methodology

Objectives

- To implement Huffman Coding algorithm in a web-based application.
- To support compression and decompression of .txt files with optimal compression ratios.
- To ensure the solution is entirely client-side, with no need for external software or servers.
- To create an educational interface that informs users about the underlying compression technique.

Research Methodology

1. **Literature Review**: Study and analyze the Huffman Coding algorithm and its application in data compression.
2. **Design and Development**: Implement the algorithm in JavaScript using appropriate data structures such as **MinHeap** and **binary trees**.
3. **Testing and Evaluation**: Evaluate the application across different file sizes to assess compression ratio and accuracy.
4. **User Interface Design**: Build an intuitive interface using HTML, CSS, and Bootstrap to guide users through upload, compression, and download steps.

1.5 Project Organization

This report is organized to present a comprehensive overview of the **Text File Compression and Decompression** project using Huffman Coding in a structured and logical manner. It begins with an **introduction** to data compression, motivations for the project, and the objectives and methodology followed. The **literature survey** chapter provides the relevance and advantages of Huffman Coding for text files.

The **system analysis** section outlines the functional and non-functional requirements of the application and includes a feasibility study. The **system design** chapter details the architecture, including the use of min-heaps, binary trees, and client-side JavaScript for implementing the Huffman algorithm. It also discusses UI design and flow.

The **implementation** chapter explains the core logic behind the encoding and decoding processes, the use of data structures like MinHeap, and the integration with a web interface using HTML, CSS, and JavaScript. The **testing and results** chapter evaluates the application's correctness, performance, and compression ratio across different input file sizes.

Finally, the **conclusion** summarizes the outcomes, discusses challenges faced during development, and proposes future enhancements, such as support for other file types, performance optimizations, or integration with cloud storage service

PHASES OF SOFTWARE DEVELOPMENT CYCLE

2.1Hardware Requirement

Specification	**Windows**	**macOS (OS X)**	**Linux**
Operating System	Microsoft Windows 7/8/10/11 (32/64 bit)	macOS 10.12 (Sierra) or higher	Ubuntu (GNOME/KDE) or similar distro
RAM	Minimum 4 GB, Recommended 8 GB	Minimum 4 GB, Recommended 8 GB	Minimum 4 GB, Recommended 8 GB
Storage	Minimum 1 GB free space	Minimum 1 GB free space	Minimum 1 GB free space
Development Tools	Visual Studio Code, Node.js, Browser	Visual Studio Code, Node.js, Browser	Visual Studio Code, Node.js, Browser
Notes	JavaScript supported browser required	Use Safari/Chrome with JS enable	Use Firefox/Chrome with JS enabled

2.2 Software Requirement

Sno.	**Name**	**Specifications**
1	Operating System	Windows/Linux/macOS
2	Programming Languages	JavaScript,HTML,CSS
3	Development Environment	Visual Studio Code
4	Runtime Environment	Node.js (for testing Huffman logic, optional)
5	Web Technologies	HTML5, CSS3, JavaScript ES6+
6	Browser Compatiblity	Chrome,Safari,Edge
7	Version Control	Git and GitHub
8	Libraries/Modules	Custom Heap and Huffman Implementation
9	Compression Algorithm	Huffman Coding
10	Testing Tools	Console based javascript testing

CODING OF FUNCTIONS

This chapter provides a comprehensive explanation of the functional code modules implemented in the browser-based application developed for lossless text file compression and decompression. The project follows a modular design, adhering to software engineering principles such as separation of concerns, reusability, and maintainability. All source code is written in JavaScript, structured logically into separate files for heap data structure implementation, Huffman coding logic, and user interaction management. These modules work together to offer users a fast and intuitive file compression tool that runs entirely on the client side.
The following are the major source files and their responsibilities:

- heap_implementation.js – Implements the MinHeap data structure required for building the Huffman Tree.
- codec_implementation.js – Contains the Huffman coding algorithm: tree construction, code generation, encoding, and decoding.
- script.js – Manages file input/output operations, interacts with the DOM, and controls the compression/decompression workflows.

Each module is described in detail below.

3.1 heap_implementation.js – MinHeap Module

The heap_implementation.js file defines a custom MinHeap class used to construct the Huffman Tree. A MinHeap is a binary heap where the parent node is always smaller than its child nodes. In Huffman coding, this structure ensures that the two least frequent symbols can be quickly accessed for combining.
Key Features:

- Dynamic insertion and deletion of nodes.
- Maintains the heap invariant with time-efficient operations.
- Written using plain JavaScript arrays.

Key Functions and Their Roles:

1. **constructor():**
   Initializes the heap array to an empty list. Internally, the heap is represented as a dynamic array.
2. **insert(node):**
   Adds a new node to the heap and ensures the min-heap property is maintained by bubbling up the new node to its correct position.
3. **extractMin():**
   Removes and returns the node with the smallest frequency (the root of the heap). This is the most critical function for the Huffman Tree as the two nodes with the lowest frequency are repeatedly extracted to form a new parent node.
4. **heapify(index):**
   Recursively restores the heap order starting from a given index. This is used internally after insertion or deletion to rebalance the tree.
5. **size():**
   Returns the current number of nodes in the heap.

Usage:
This module supports the core Huffman algorithm by maintaining the priority queue of nodes during tree construction. Every time two nodes are combined, a new node is inserted, and the heap is updated efficiently.

3.2 codec_implementation.js – Huffman Codec Module

This file includes the main implementation of the Huffman algorithm. It defines classes and functions for constructing the Huffman Tree, generating binary codes for each character, encoding the input, and decoding the compressed output back to original text.

1. **classHuffmanNode:**
   A constructor for creating tree nodes. Each node contains:
   - character (symbol)
   - frequency
   - left and right children
2. **buildHuffmanTree(frequencyMap):**
   Accepts a frequency map of all characters in the input text and builds a Huffman Tree. It pushes all character nodes into a MinHeap and repeatedly extracts the two least frequent nodes, merges them into a parent node, and reinserts until only one node (the root) remains.
3. **generateCodes(root,currentCode,codeMap):**
   Recursively traverses the Huffman Tree to assign binary codes to each character based on their path from the root (left = 0, right = 1). Stores the codes in codeMap.
4. **encodeText(text,codeMap):**
   Translates the original text into a binary string using the character codes from codeMap. This binary string is the compressed form of the file content.
5. **decodeText(encodedText,root):**
   Traverses the Huffman Tree based on bits in the encodedText to reconstruct the original string. This ensures the original text is fully recovered with no data loss.
6. **getFrequencyMap(text):**
   Helper function that computes how many times each character appears in the input. This frequency data is used to build the Huffman Tree.

3.3 script.js – File and UI Handler Module

This file manages user interactions, file input/output operations, and connects the HTML interface to the Huffman algorithm. It captures the file uploaded by the user, reads its contents, and triggers encoding or decoding actions.
Key Functionalities:

1. **handleFileUpload(event):**
   Handles the file input event. It reads the uploaded .txt file using FileReader and stores its content in a variable for processing.
2. **compressText():**
   - Analyzes the text to compute frequency.
   - Builds the Huffman Tree and code map.
   - Encodes the text and displays the compressed output.
   - Calculates statistics such as compression ratio and reduced file size.

3. **decompressText():**
    - Uses the stored Huffman Tree and encoded binary string.
    - Reconstructs the original text using decodeText.
    - Displays or enables download of the original content.
4. **displayCompressionStats(originalSize,compressedSize):**
   Calculates and shows percentage reduction in file size, helping users understand the efficiency of compression.
5. **Event_Listeners:**
   DOM event handlers for “Compress” and “Decompress” buttons are defined and linked to respective functions.

Usage:
script.js ties everything together. It acts as the controller in an MVC-like architecture, routing user actions to algorithmic modules and managing results.

3.4 Supporting Files and Frontend Logic

- **index.html**:
  A simple and responsive HTML file that includes:
    - File upload buttons
    - Action buttons (Compress / Decompress)
    - Output containers for displaying statistics and messages
- **styles.css:**
  Defines visual styling of all UI components using modern responsive design practices. Includes button effects, layout control, spacing, and font formatting.
- info.html:
  Provides a separate information/help page detailing how the system works and what Huffman compression does.

3.5 Integration Flow of Modules

The following steps summarize how all code modules integrate:
1. User uploads a .txt file via index.html.
2. script.js reads the content and calls codec_implementation.js functions.
3. heap_implementation.js manages tree construction during encoding.
4. Encoded content and statistics are shown in browser; output can be downloaded.
5. During decompression, the stored tree and encoded string are used to restore the original text.

3.6 Summary

Each code module in this project is built with a clear objective. The MinHeap implementation simulates a priority queue essential for Huffman logic. The codec implementation encapsulates the core algorithm for efficient and lossless text compression. The script file connects the interface to the algorithm and offers an intuitive user experience.

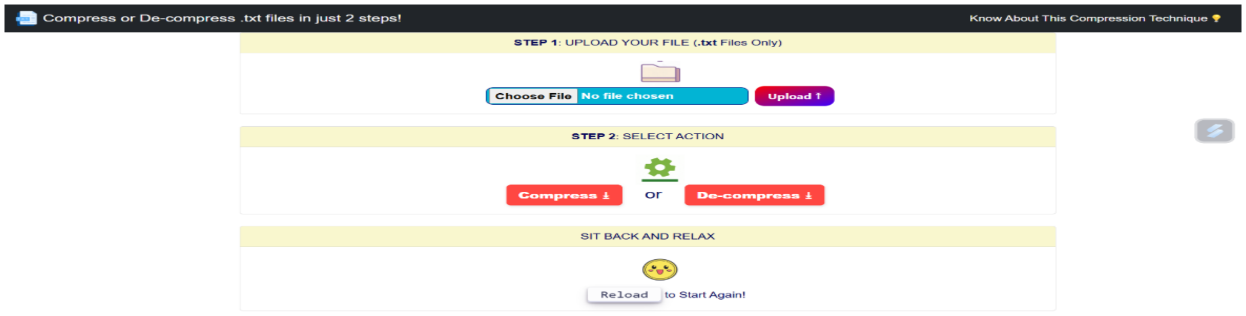
SNAPSHOT

This chapter provides a visual walkthrough of the Text File Compression and Decompression application. Each snapshot illustrates a key part of the system's functionality — from uploading a text file to compressing, decompressing, and downloading results. These screenshots demonstrate the user-friendly and interactive design of the system and verify the correct implementation of the Huffman algorithm through a browser-based interface.

4.1 Home Interface

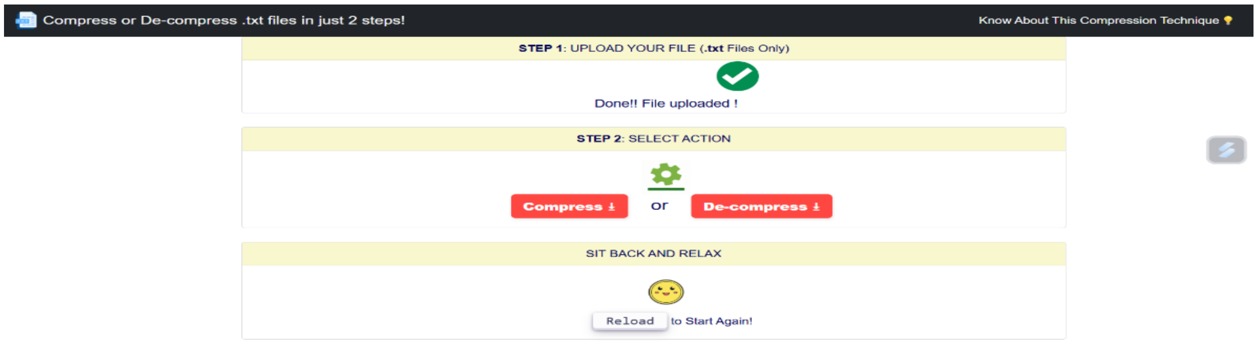
This is the default view of the application upon launching index.html in any modern web browser. It presents the user with:

- A file input field to upload a .txt file.
- Buttons to perform compression and decompression.
- Instructional labels and areas to display output.


Compress or De-compress .txt files in just 2 steps!
Know About This Compression Technique
STEP 1: UPLOAD YOUR FILE (.txt Files Only)
Choose File
No file chosen
Upload
STEP 2: SELECT ACTION
Compress
or
De-compress
SIT BACK AND RELAX
Reload
to Start Again!

4.2 Uploading a File

The user selects a .txt file for compression. Once the file is uploaded, it is read and processed by the FileReader API. A preview of the file size or content is optionally shown.


Compress or De-compress .txt files in just 2 steps!
Know About This Compression Technique
STEP 1: UPLOAD YOUR FILE (.txt Files Only)
Done!! File uploaded !
STEP 2: SELECT ACTION
Compress
or
De-compress
SIT BACK AND RELAX
Reload
to Start Again!

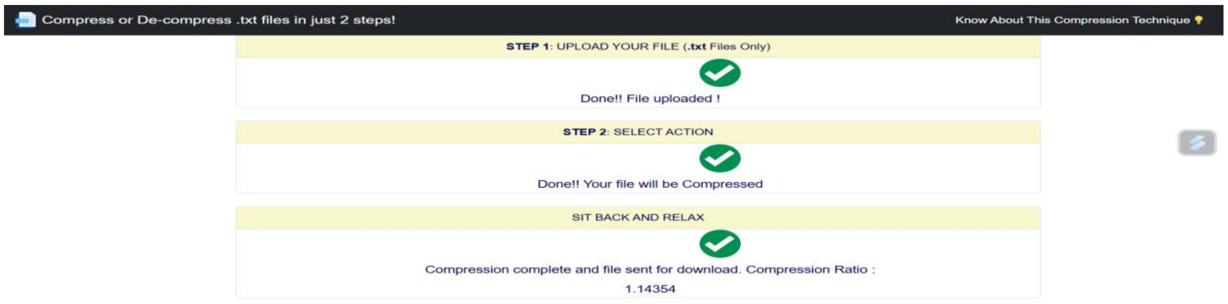
4.3 Compression Execution

After clicking the Compress button:

- The script calculates character frequencies.
- Builds the Huffman Tree.
- Generates binary codes and compresses the original text.

The result displays:

- Original and compressed sizes.
- Compression ratio.
- Option to download the compressed file.



Compress or De-compress .txt files in just 2 steps!

Know About This Compression Technique

STEP 1: UPLOAD YOUR FILE (**.txt** Files Only)

Done!! File uploaded !

STEP 2: SELECT ACTION

Done!! Your file will be Compressed

SIT BACK AND RELAX

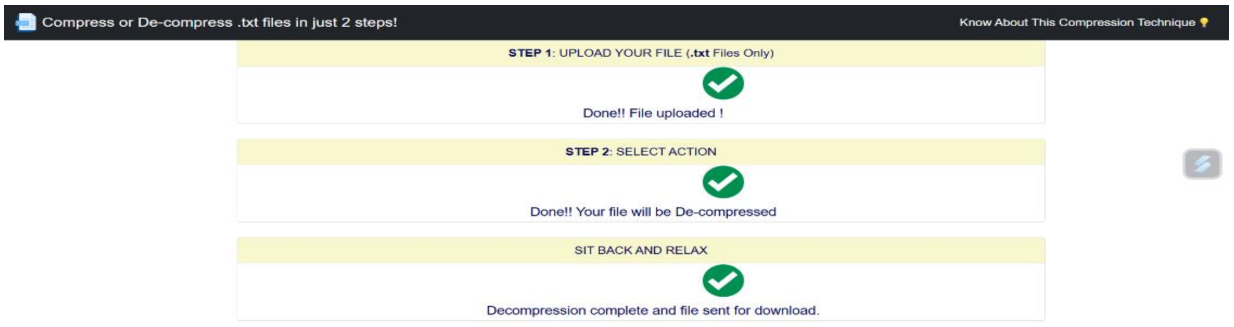
Compression complete and file sent for download. Compression Ratio : 1.14354

4.5 Decompression Process

By clicking the Decompress button:

- The stored Huffman Tree and binary string are used to decode the original message.
- The full text is displayed or made available for download.

This step ensures the accuracy and reversibility of the Huffman compression method.



Compress or De-compress .txt files in just 2 steps!

Know About This Compression Technique

STEP 1: UPLOAD YOUR FILE (**.txt** Files Only)

Done!! File uploaded !

STEP 2: SELECT ACTION

Done!! Your file will be De-compressed

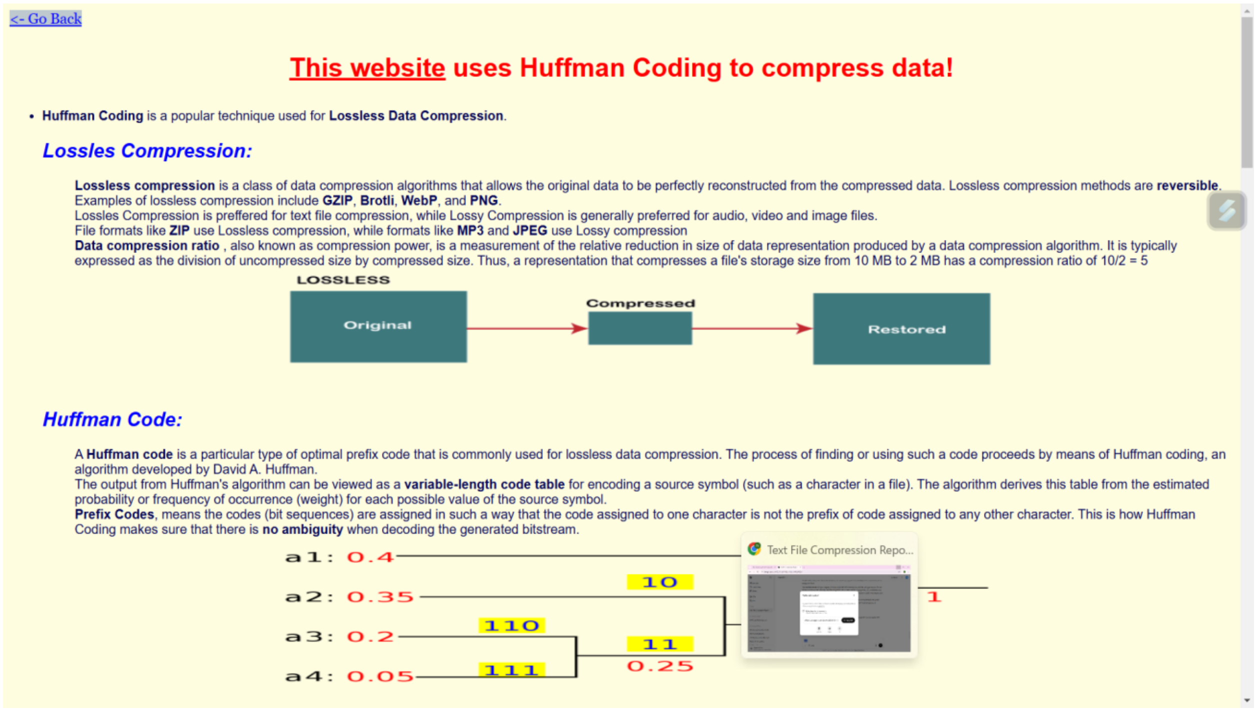
SIT BACK AND RELAX

Decompression complete and file sent for download.

govind kathayat (govindkathayat12@outlook.com) is signed in

4.6 Info Page

An additional info.html page is included for user reference. It provides a short explanation of Huffman coding, usage instructions, and project background.



<- Go Back

This website uses Huffman Coding to compress data!

- **Huffman Coding** is a popular technique used for **Lossless Data Compression**.

Lossles Compression:

Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data. Lossless compression methods are **reversible**.
Examples of lossless compression include **GZIP**, **Brotli**, **WebP**, and **PNG**.
Lossles Compression is preffered for text file compression, while Lossy Compression is generally preferred for audio, video and image files.
File formats like **ZIP** use Lossless compression, while formats like **MP3** and **JPEG** use Lossy compression
Data compression ratio , also known as compression power, is a measurement of the relative reduction in size of data representation produced by a data compression algorithm. It is typically expressed as the division of uncompressed size by compressed size. Thus, a representation that compresses a file's storage size from 10 MB to 2 MB has a compression ratio of 10/2 = 5

Huffman Code:

A **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman.
The output from Huffman's algorithm can be viewed as a **variable-length code table** for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol.
Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is **no ambiguity** when decoding the generated bitstream.

Summary:

This chapter confirms that the application performs as expected with intuitive flow and accurate results. From file selection to compression and decompression, each action is clearly mapped on the interface and supported with real-time feedback and downloadable results.

LIMITATIONS

While the Text File Compression and Decompression project using Huffman Coding demonstrates effective functionality in a browser-based environment, certain limitations were identified during its design, development, and testing phases. These limitations arise primarily due to the constraints of client-side programming, algorithmic decisions, and file handling capabilities of web browsers. This chapter outlines both technical and practical limitations associated with the current version of the project.

5.1 File Type Support

- The application is designed specifically for plain text (.txt) files.
- It does not support other formats such as PDF, DOCX, or HTML.
- Binary files and special character encodings (like UTF-16) may not compress properly.

5.2 Compression Ratio Efficiency

- Huffman coding works best when there is a high frequency of repeating characters.
- For short or random-character files, the compression ratio might be negligible or even slightly increase the size due to header information (e.g., code map storage).
- Compared to advanced compression methods (e.g., LZW, DEFLATE, BWT), Huffman may not always yield the highest compression ratio.

5.3 No Persistent State

- The application is completely client-side with no backend.
- If the page is refreshed, the encoded data, Huffman tree, and any output stored in memory are lost.
- There is no local storage or session caching, so the user must restart the entire process after a reload.

5.4 No Encryption or Data Security

- Compressed files are saved without any encryption.
- Since Huffman compression is not a security mechanism, any third-party who understands the algorithm can easily decompress the file.
- This limits the use of the tool for confidential or sensitive documents.

5.5 Limited Error Handling

- No support for corrupted or partially encoded files.
- If a malformed or incompatible file is provided for decompression, the application may crash or yield incorrect results without a helpful error message.
- Error messages are minimal and do not guide the user in all possible failure scenarios (e.g., empty file, invalid format, zero-length input).

5.6 Huffman Tree Non-Persistence

- The Huffman Tree used for decoding must exactly match the one used during encoding.
- In this implementation, both encoder and decoder reside on the same session. If the tree is not stored or transmitted alongside the compressed data, decompression outside this app is not possible.

5.7 Performance on Large Files

- Although the app can handle moderate-sized files (~1 MB) comfortably, performance degrades with very large files due to memory constraints in browsers.
- The in-browser memory usage can increase rapidly during tree construction and encoding/decoding, potentially causing the browser tab to freeze or crash.

5.8 No Progress Feedback or Cancel Option

- Compression and decompression operations are performed synchronously.
- For large files, the UI becomes unresponsive while the operation completes.
- There is no progress bar, estimated time, or cancel button for users to manage long operations.

5.9 No Multi-language or Unicode Text Support

- The app is primarily tested with ASCII-based English text.
- Unicode characters (e.g., emojis, accented characters, non-Latin scripts) may not be encoded or displayed correctly.
- This limits its applicability for global/multilingual document compression.

5.10 Browser Dependency

- Although designed for compatibility with modern browsers (Chrome, Firefox, Edge), behavior may vary slightly depending on JavaScript engine implementations.
- Older browsers or those with JavaScript restrictions may fail to run the application.

Summary

While the application meets its primary goals of demonstrating client-side Huffman compression and decompression, it faces several limitations related to file type support, algorithm performance, scalability, and usability. These challenges are typical in educational and prototype-level projects and provide strong motivation for future enhancement in upcoming versions.

ENHANCEMENTS

This chapter discusses various enhancements that can be introduced in future versions of the Text File Compression and Decompression application. The project, while functional and effective as a proof-of-concept, is open to numerous improvements in terms of user experience, performance, scalability, and compatibility. These enhancements aim to overcome the current limitations (discussed in Chapter 5) and make the system more robust, versatile, and production-ready.

6.1 Support for Multiple File Formats

- Extend file input compatibility to include other text-based formats like .csv, .log, and even rich-text formats (.docx, .pdf after text extraction).
- Implement a pre-processing layer that extracts plain text from such formats before compression.

6.2 Persistent Huffman Tree Storage

- Introduce serialization of the Huffman Tree into a compact format (e.g., JSON) and store it as a header in the compressed file.
- This enables the decompression module to reconstruct the tree independently, making it possible to decompress files across different sessions or systems.

6.3 Downloadable Custom Decoder

- Provide an option to export a decoder script along with the compressed file.
- This allows users to send compressed files to others along with the decoder, even if they do not use the original web tool.

6.4 Progress Bar and Asynchronous Operations

- Introduce a visual progress bar to track compression/decompression for large files.
- Refactor JavaScript logic to use asynchronous functions (async/await or Promises) to prevent UI freezing.
- Include cancel or reset buttons for long operations.

6.5 Unicode and Multilingual Text Support

- Expand encoding logic to fully support Unicode characters, enabling compression of text files in any language, including emojis and special characters.
- Ensure UTF-8 or UTF-16 encoding is properly handled during file reading and writing.

6.6 Compression Format Optimization

- Implement better space management in the compressed file by introducing bit-level storage instead of string-based binary representation.
- Reduce the size of the code map and optimize Huffman Tree serialization to increase compression efficiency.

6.7 Cloud Integration & File Sharing

- Enable integration with cloud services (Google Drive, Dropbox, OneDrive) for importing/exporting files.

6.8 Cross-Session Decompression

- Store encoded data and Huffman tree in localStorage or IndexedDB to preserve data across sessions.
- Add support for drag-and-drop decompression of files even if the original session is lost.

6.9 Security & Encryption

- Add an optional password-protection or encryption layer during compression using symmetric encryption (e.g., AES).
- This would enhance data security for users compressing confidential or sensitive documents.

6.10 Improved User Interface & Accessibility

- Redesign UI with animations, tooltips, and themes (dark/light modes) to improve user engagement.
- Add keyboard navigation and screen reader compatibility for accessibility.
- Provide user logs or history tracking of compressed/decompressed files during the session.

6.11 Mobile and Offline Support

- Optimize the web application for mobile browsers and touchscreen devices.
- Add support for offline usage using service workers or local JavaScript bundles.

Summary

These proposed enhancements can greatly extend the functionality, performance, and usability of the Huffman Coding-based compression system. By addressing the technical limitations and evolving user expectations, future versions can transform this application from an educational tool into a practical, real-world utility for secure and efficient file management.

CONCLUSION

The “Text File Compression and Decompression” project successfully demonstrates the practical application of Huffman Coding using web-based technologies. By integrating core data structures like MinHeap and binary trees, the system efficiently performs lossless compression and decompression of plain text files, all within the user's browser.

The project achieves its primary objectives:

- Implementation of a classic compression algorithm using JavaScript.
- A clean, user-friendly interface built with HTML and CSS.
- File input/output and compression results handled entirely on the client-side, requiring no server or installation.

Throughout development, the project reinforced core software engineering principles such as modular coding, separation of concerns, and efficient memory use. It also provided hands-on experience with real-time file processing and browser-based application logic.

While the system currently supports only .txt files and lacks advanced features like encryption and multi-format support, it lays a solid foundation for future expansion. The tool is especially useful for educational purposes, showcasing how foundational algorithms can solve real-world problems interactively and efficiently.

In conclusion, this project bridges theoretical concepts with practical implementation, delivering a functional, lightweight, and accessible compression utility for text-based data.

REFERENCES

1. Salomon, D. (2007). Data Compression: The Complete Reference (4th ed.). Springer. https://link.springer.com/book/10.1007/978-1-84628-602-5

2. Sayood, K. (2017). Introduction to Data Compression (5th ed.). Morgan Kaufmann. https://www.sciencedirect.com/book/9780128094747/introduction-to-data-compression

3. Ziv, J., & Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. IEEE Transactions IEEE Transactions on Information Theory, 23(3), 337–343. https://ieeexplore.ieee.org/document/1055714

4. Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE ,40(9) 1098-1101 https://ieeexplore.ieee.org/document/4051119

5. Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic Coding for Data Compression. Communications of the ACM, 30(6), 520–540. https://dl.acm.org/doi/10.1145/214762.214771

6. Nelson, M., & Gailly, J.-L. (1996). The Data Compression Book (2nd ed.). M&T Books. https://archive.org/details/data-compression-book-nelson-gailly

7. GNU (2024). Gzip Manual. https://www.gnu.org/software/gzip/manual/gzip.html 18