A SYNOPSIS ON

Text File Compression-Decompression

Submitted in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Manish Singh Rautela	**University Roll No.- 2261349**
Govind Singh Kathayat	**University Roll No.- 2261236**
Pulkit Singh Bora	**University Roll No.-2261448**
Sagar Singh Khanka	**University Roll No.-2261503**

Under the Guidance of

Mr. Anubhav Bewerwal

Assistant Professor

Project Team ID: 104

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

Graphic Era Hill University
BHIMTAL CAMPUS

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the Synopsis entitled **" Text File Compression-Decompression"** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the **Graphic Era Hill University, Bhimtal campus** and shall be carried out by the undersigned under the supervision of **Mr. Anubhav Bewerwal, Assistant Professor**, Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Manish Singh Rautela	University Roll No.- 2261349
Govind Singh Kathayat	University Roll No.- 2261236
Pulkit Singh Bora	University Roll No.-2261448
Sagar Singh Khanka	University Roll No.-2261503

The above mentioned students shall be working under the supervision of the undersigned on the **"Text File Compression-Decompression"**

Signature
Supervisor

Signature
Head of the Department

Internal Evaluation (By DPRC Committee)

Status of the Synopsis: Accepted / Rejected

Any Comments:

Name of the Committee Members: **Signature with Date**

1.

2.

Table of Contents

Chapter -1

Introduction and Problem Statement

Introduction

With the exponential growth of digital data, efficient data storage and transmission have become critical concerns in computer science and information technology. Text files, one of the most commonly used data formats, often occupy significant storage space. While text data itself may not be large compared to multimedia files such as images and videos, its widespread use in various domains—including web applications, document storage, log files, and messaging services—makes efficient storage and transmission crucial.

Text file compression is an essential technique that allows users to reduce the size of text-based files while maintaining data integrity. Compression algorithms achieve this by identifying redundant patterns within the text and encoding the information more efficiently. This process not only minimizes storage requirements but also enhances the efficiency of data transmission across networks.

The primary motivation behind text file compression is to optimize resource utilization. Whether it is cloud storage, embedded systems, or large-scale data centers, reducing file sizes can lead to significant cost savings in storage and transmission. Additionally, compressed files reduce the amount of bandwidth required for communication, making web applications and file-sharing platforms more efficient.

Importance of Text Compression

1. **Storage Optimization** – Storing vast amounts of data requires efficient utilization of disk space. Compression reduces the overall storage footprint, allowing for better resource management.
2. **Faster Transmission** – In network communication, reducing file size enhances the speed of transmission, enabling quicker uploads, downloads, and data synchronization.
3. **Cost Efficiency** – Lower storage and bandwidth requirements translate to reduced operational costs, benefiting both individual users and enterprises.
4. **Better Performance** – Applications that handle large text files, such as databases, logging systems, and web servers, benefit from faster read and write operations when data is compressed.
5. **Environmental Impact** – Reducing storage and transmission costs indirectly leads to lower energy consumption, contributing to environmental sustainability.

This project focuses on implementing efficient text compression techniques to address these challenges. By leveraging algorithms such as Huffman coding and Run-Length Encoding (RLE), the project aims to develop an effective solution for compressing text files while ensuring lossless data recovery.

Problem Statement

While text compression is a well-established field, there are several persistent challenges that motivate the need for efficient and optimized solutions. Some of the major issues in text data storage and transmission include:

1. Exponential Growth of Text Data

As digital communication and data storage become increasingly prevalent, the amount of text data generated daily has reached unprecedented levels. From social media posts and instant messages to extensive databases and system logs, the demand for efficient storage solutions has never been higher. Without proper compression techniques, managing and storing this ever-growing volume of text data becomes an overwhelming challenge.

- **Example:** Websites and cloud-based services generate logs that can grow to several gigabytes daily. Efficient compression allows these logs to be stored and retrieved quickly without consuming excessive disk space.

2. Storage Limitations in Devices and Cloud Environments

Despite advances in storage technology, devices and cloud platforms have finite storage capacities. Users who frequently store large text files may experience performance degradation and increased costs due to storage limitations.

- **Example:** A cloud-based file storage service that handles user-generated content, such as documents and logs, benefits significantly from compression as it reduces the overall storage footprint and minimizes costs.

3. Bandwidth Constraints and Network Efficiency

Transmitting large uncompressed text files over the internet consumes significant bandwidth, leading to slow data transfers and increased network congestion. This issue is particularly relevant for mobile networks, where bandwidth limitations and data caps impose restrictions on file sharing and synchronization.

- **Example:** Messaging applications that send text-based files over limited mobile data networks can optimize efficiency by compressing messages and reducing the amount of transmitted data.

4. Need for Lossless Compression

In contrast to lossy compression techniques (used in audio, video, and image compression), text file compression must be **lossless**, meaning that the original data must be perfectly reconstructed after decompression. Even a minor alteration to the text can lead to significant issues, particularly in critical applications such as financial records, legal documents, and programming code.

- **Example:** If a compressed text file contains code from a software project, any loss of data would lead to errors in execution. Lossless compression ensures that the original code remains intact after decompression.

5. Computational Complexity and Processing Time

Compression algorithms require computational resources to encode and decode text files. Some traditional compression techniques are computationally expensive and unsuitable for real-time applications. Therefore, there is a need to balance compression efficiency with processing speed.

- **Example:** A web server that compresses large text files before transmission must ensure that the compression process does not introduce delays that impact user experience.

6. Security and Compatibility Issues

While compression reduces file size, it should also ensure compatibility with various systems and platforms. Some compression formats may not be widely supported, limiting their usability. Additionally, compressed files should be resistant to corruption and maintain integrity during transmission.

- **Example:** A compressed text file shared via email should be compatible with different operating systems and file extraction tools to ensure that all recipients can access the data.

Scope of the Project

This project aims to develop an efficient and optimized text file compression tool that addresses the challenges mentioned above. The focus is on:

1. **Developing an effective lossless compression technique** that significantly reduces text file size without data loss.
2. **Implementing lightweight and efficient algorithms** such as Huffman coding and Run-Length Encoding to balance compression ratio and processing speed.
3. **Ensuring compatibility** with various platforms and software applications to maximize usability.
4. **Optimizing storage and transmission** of compressed files to improve performance in cloud-based environments, mobile applications, and low-bandwidth networks.

By implementing these features, the project aims to provide a practical and high-performance solution for text file compression, making storage and transmission more efficient in modern digital environments.

Chapter -2

Background/ Literature Survey

Historical Development of Text Compression

The field of text compression has evolved significantly over the decades, with various algorithms being developed to improve efficiency. Below is a timeline of major advancements in lossless text compression:

1. Shannon-Fano Coding (1948)

Developed by Claude Shannon and Robert Fano, this technique was an early attempt at variable-length encoding based on symbol probabilities. However, it was later replaced by more efficient methods such as Huffman coding.

2. Huffman Coding (1952)

Introduced by David Huffman, this algorithm became one of the most widely used lossless compression techniques. Huffman coding assigns shorter binary codes to more frequent characters and longer codes to less frequent ones, reducing overall file size.

3. Run-Length Encoding (RLE) (1960s)

RLE is a simple and effective compression method that replaces repeated sequences of characters with a single instance followed by a repetition count. This technique works well for text with many consecutive identical characters, such as tabular data and monochrome bitmaps.

4. Lempel-Ziv-Welch (LZW) (1977-1984)

Developed by Jacob Ziv and Abraham Lempel, LZW is a dictionary-based compression method widely used in GIF images and UNIX-based file compression tools. It dynamically builds a dictionary of patterns encountered in the text, replacing repeated sequences with shorter codes.

5. Burrows-Wheeler Transform (BWT) (1994)

BWT is not a compression method itself but a preprocessing technique that rearranges text to make it more compressible using other algorithms such as Move-to-Front encoding. It is widely used in tools like bzip2.

Existing Text Compression Techniques

Several compression algorithms have been developed over the years, each with its strengths and weaknesses. This section reviews commonly used text compression techniques and their applications.

1. Huffman Coding

Huffman coding is a variable-length prefix coding technique that constructs an optimal binary tree based on character frequency. The steps involved are:

- Count the frequency of each character in the text.
- Build a binary tree where characters with lower frequency have deeper nodes.
- Assign shorter binary codes to more frequent characters and longer codes to less frequent ones.

Advantages:

1. Optimal for symbol-by-symbol encoding.
2. Widely used in data compression applications like ZIP files.

Disadvantages:

1. Requires two passes over the text (one for frequency analysis, one for encoding).
2. Does not adapt dynamically to changes in data.

2. Run-Length Encoding (RLE)

RLE replaces consecutive repeated characters with a single character followed by the repetition count.

Example:

Original: AAAABBBCCDDDD
Encoded: 4A3B2C4D

Advantages:

1. Simple and fast to implement.
2. Effective for texts with repeated characters.

Disadvantages:

1. Inefficient for texts with no repeated characters.
2. Results in larger file sizes if repetitions are minimal.

3. Lempel-Ziv-Welch (LZW) Encoding

LZW builds a dictionary of repeated substrings dynamically and replaces them with shorter codes. This technique is widely used in GIF image compression and UNIX commands like compress.

Advantages:

1. Works well for long text files.
2. Does not require prior knowledge of character frequencies.

Disadvantages:

1. Dictionary growth may require large memory storage.
2. Performance depends on text characteristics.

4. Arithmetic Coding

Unlike Huffman coding, which assigns discrete codes to symbols, arithmetic coding represents an entire message as a fraction between 0 and 1. It provides higher compression efficiency by eliminating gaps between codewords.

Advantages:

1. Provides better compression than Huffman coding in some cases.
2. Does not require integer bit allocations for each symbol.

Disadvantages:

1. Computationally complex.
2. Susceptible to floating-point precision issues in hardware.

Comparison of Compression Algorithms

Algorithm	Type	Best Use Case	Compression Efficiency	Speed	Complexity
Huffman Coding	Lossless	General text	High	Moderate	Moderate
Run-Length Encoding	Lossless	Repetitive text	Low to Moderate	Fast	Low
LZW	Lossless	Large text files	High	Moderate	High
Arithmetic Coding	Lossless	High-compression scenarios	Very High	Slow	Very High

Relevance of Existing Work to the Project

The text file compression project aims to implement an effective lossless compression algorithm that optimizes storage and transmission. Based on the literature review, the following key insights guide the project's development:

1. **Huffman Coding is a well-established and efficient method** that provides a good trade-off between compression efficiency and processing time. It will be a central component of the project.
2. **Run-Length Encoding can be integrated** for handling highly repetitive data, improving compression efficiency in structured text files.

Chapter -3

Objectives

Introduction to Project Objectives

The primary goal of this project is to develop a **text file compression system** that efficiently reduces the size of textual data while ensuring lossless reconstruction. Given the increasing amount of digital text being generated daily—ranging from documents and web pages to logs and archives—compression plays a crucial role in optimizing storage, reducing transmission time, and improving overall efficiency in data handling.

Compression algorithms must be designed to maximize **compression ratio**, **minimize computational overhead**, and ensure **fast encoding and decoding**. The project will focus on implementing **Huffman Coding** and **Run-Length Encoding (RLE)**—two well-known lossless compression techniques—to achieve optimal performance.

This section details the **objectives of the project**, covering key performance indicators, functional and non-functional goals, and expected benefits.

Key Objectives of the Project

The project encompasses several key objectives that focus on different aspects of text compression, from algorithm implementation to efficiency evaluation. The following objectives will guide the development process:

Objective 1: Implement a Lossless Text Compression Algorithm

The primary objective is to design and implement a **lossless compression algorithm** that effectively reduces text file sizes while ensuring that the original content can be perfectly reconstructed. This means that:

- No data should be lost during compression.
- The decompressed text should be identical to the original.
- The algorithm should maintain high fidelity, ensuring that no alterations occur in the textual data.

This project will use **Huffman Coding** as the core compression algorithm, with possible integration of **Run-Length Encoding (RLE)** for better handling of repetitive text sequences.

Objective 2: Optimize Compression Efficiency

Another important goal is to achieve **high compression efficiency** by minimizing the file size after compression. Compression efficiency is determined by the **compression ratio**, which is defined as:

Compression Ratio=Original File SizeCompressed File Size*Compression Ratio=Compressed File SizeOriginal File Size*

To optimize compression efficiency, the project will:

- Identify and eliminate redundant characters or sequences.
- Use **Huffman Coding** to assign shorter binary codes to more frequently occurring characters.
- Experiment with **Run-Length Encoding (RLE)** to check if additional optimization is possible for specific types of text data.
- Compare the implemented algorithm with other existing compression methods to analyze its efficiency.

A higher compression ratio means that the algorithm is more effective in reducing file size, which is crucial for storage and transmission purposes.

Objective 3: Ensure Fast Compression and Decompression Speed

While achieving high compression efficiency is crucial, the algorithm must also be **computationally efficient** to ensure that both compression and decompression are performed quickly. To meet this objective, the project will:

- Optimize data structures (e.g., binary trees and heaps) for faster execution.
- Minimize processing overhead to ensure that large text files can be compressed within a reasonable time frame.
- Test the algorithm's execution speed with different file sizes to evaluate scalability.

The goal is to strike a balance between **compression speed** and **compression ratio**, ensuring that neither performance nor efficiency is compromised.

Objective 4: Develop a User-Friendly Implementation

A crucial part of this project is to create a **simple and user-friendly system** that allows users to easily compress and decompress text files. To achieve this, the project will:

- Provide a **command-line interface (CLI)** or **graphical user interface (GUI)** for users to interact with the compression tool.
- Allow users to select a text file, compress it, and store the compressed version efficiently.
- Implement an intuitive interface where users can quickly understand the functions and workflow of the system.

By ensuring ease of use, the project can cater to a broader audience, including software developers, data analysts, and general users who need efficient file compression.

Objective 5: Compare and Benchmark the Compression Algorithm

To determine the effectiveness of the implemented algorithm, it is essential to compare it against existing compression techniques. This objective will involve:

- Testing the compression performance on different types of text files (e.g., English literature, program source code, structured data logs).
- Comparing results with widely used compression tools such as **Gzip, ZIP, and LZW-based compression methods**.
- Evaluating **compression ratio**, **execution time**, and **memory usage** across different datasets.

By benchmarking the performance, we can validate the efficiency of the chosen approach and suggest improvements if needed.

Objective 6: Reduce Storage and Bandwidth Usage

One of the major benefits of compression is **reducing storage space requirements and bandwidth consumption** when transmitting files over networks. This objective focuses on:

- Compressing large text files to reduce their storage footprint.
- Ensuring that compressed files can be easily transmitted via email, cloud storage, or other file-sharing mechanisms.
- Minimizing bandwidth costs for data transmission, which is especially useful for applications dealing with large volumes of textual data (e.g., log files, chat archives, web content).

By achieving this objective, the project aims to contribute to **efficient data handling and cost savings** for storage and network usage.

Objective 7: Maintain Scalability and Extensibility

A well-designed compression system should be able to handle:

- Small text files (e.g., a few KBs) as well as **large documents spanning multiple MBs or GBs**.
- Different types of text content, including structured (CSV, JSON, XML) and unstructured (plain text, source code).
- Future extensions, allowing the incorporation of **new compression techniques** or hybrid models combining multiple algorithms.

This objective ensures that the system remains **scalable and adaptable** to different use cases beyond the initial scope of the project.

Objective 8: Ensure Secure and Error-Free Compression

Data integrity is a crucial factor in lossless compression. A good compression system must ensure that the compressed file is not **corrupted or altered** during storage or transmission. To achieve this, the project will:

- Implement **error detection mechanisms** to ensure that data remains intact during compression and decompression.
- Verify that the decompressed file is an exact match to the original input.
- Explore the feasibility of adding **checksum verification** to detect accidental modifications.

This objective ensures that the system provides **reliable and error-free** data compression.

Objective 9: Provide Comprehensive Documentation and Testing

To ensure usability and maintainability, the project will include:

- Detailed documentation explaining **the algorithms used, system architecture, and implementation details**.
- Well-structured **code comments** to aid future modifications and enhancements.
- A **test suite** to evaluate the compression algorithm with various sample inputs and edge cases.

By providing proper documentation and testing, the project will ensure **reliability, ease of understanding, and future extensibility**.

Expected Outcomes and Benefits

By achieving the above objectives, the project aims to provide the following benefits:

1. **Efficient storage and reduced file size**, making it easier to store and manage large volumes of text data.
2. **Faster transmission of text files** across networks, leading to improved bandwidth efficiency.
3. **Seamless compression and decompression processes**, ensuring no data loss.
4. **A user-friendly interface** that makes compression accessible to non-technical users.
5. **Benchmark results comparing the implemented algorithm with existing solutions**, validating its effectiveness.
6. **Scalability and adaptability**, allowing future improvements or integration with other compression techniques.

Chapter -4

Hardware and Software Requirements

Hardware Requirements ;

Sl. No	Name of the Hardware	Specification
1	Processor	Intel Core i5 or higher / AMD Ryzen 5 or higher
2	RAM	8GB or more
3	Storage	256GB SSD or higher
4	Operating System	Windows, Linux, or macOS

Software Requirements ;

Sl. No	Name of the Software	Specification
1	Programming Language	Python / JavaScript / C++ (as per project implementation)
2	Development Environment	VS Code / PyCharm / Eclipse / CodeBlock
3	Libraries & Dependencies	Node.js (if JavaScript-based), NumPy, Huffman Encoding Libraries
4	Compression Tools for Benchmarking	WinRAR, 7-Zip, Gzip for comparison

Chapter -5

Possible Approach/ Algorithms

Overview of Text Compression Approaches

Text compression can be broadly classified into **lossless** and **lossy compression**. Since the goal of this project is to ensure that the original text file can be **perfectly reconstructed** after decompression, we focus on **lossless compression algorithms**.

Several lossless compression techniques exist, including:

1. **Huffman Coding** – Uses variable-length binary codes based on character frequencies.
2. **Run-Length Encoding (RLE)** – Replaces consecutive repeating characters with a single character and count.
3. **Lempel-Ziv-Welch (LZW) Compression** – Uses dictionary-based encoding for pattern recognition.
4. **Arithmetic Coding** – Assigns a fractional value to entire sequences of symbols.

Why Huffman Coding?

Huffman coding is a **greedy algorithm** that efficiently compresses text by assigning shorter binary codes to frequently occurring characters and longer codes to less frequent ones. This ensures that **no code is a prefix of another**, allowing for efficient and error-free decoding.

Advantages of Huffman Coding:

1. **Optimality** – It provides the smallest possible average code length based on character frequency.
2. **Lossless Compression** – Ensures that the original data is perfectly reconstructable.
3. **Widely Used** – Applied in ZIP, PNG image compression, and various data compression applications.

Limitations:

1. **Requires two passes** – One to build the frequency table and another to encode the data.
2. **Inefficient for very small text files** – The overhead of storing the Huffman tree may outweigh the compression benefits.

Huffman Coding Algorithm

The Huffman coding algorithm consists of the following steps:

Step 1: Calculate Frequency

- Count the occurrence of each character in the text file.

Step 2: Build a Min-Heap (Priority Queue)

- Create a **min-heap** where each node represents a character and its frequency.
- The node with the smallest frequency has the **highest priority** in extraction.

Step 3: Construct the Huffman Tree

- Extract the two nodes with the smallest frequency from the heap.
- Merge them into a new node with a frequency equal to their sum.
- Repeat until only **one node (the root of the Huffman tree)** remains.

Step 4: Generate Huffman Codes

- Assign a **binary code (0 or 1)** to each character by traversing the Huffman tree.
- Left branch is assigned **0**, right branch **1**.

Step 5: Encode the Text

- Replace each character in the text with its corresponding Huffman code.

Step 6: Store the Encoded Data and Huffman Tree

- Store the compressed binary sequence.
- Save the Huffman tree structure to allow decompression.

Complexity Analysis :

Operation	Time Complexity
Constructing frequency table	**O(n)**
Building Huffman tree	**O(n log n)**
Generating Huffman codes	**O(n)**
Encoding text	**O(n)**
Overall complexity	**O(n log n)**

Where **n** is the number of unique characters in the text.

Expected Compression Efficiency:

Huffman Coding performs best when:

1. The text contains **many repeating characters**, allowing shorter codes for frequent symbols.
2. Large files are used, where storage savings are more significant.
3. The characters follow a skewed frequency distribution (e.g., English text with frequent vowels like 'e', 'a', 'o').

   However, it may not be as effective for **highly random** text, where characters have almost equal frequencies.

Conclusion:

Huffman Coding is a **powerful, efficient, and widely used lossless compression algorithm** that provides significant file size reduction. It ensures **optimal prefix encoding**, maintains **perfect reconstruction of data**, and is implemented in many real-world applications like **ZIP compression, JPEG image encoding, and data transmission protocols**.

By applying Huffman Coding, this project aims to **reduce text file size while maintaining fast encoding and decoding speeds**.

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