N˚d'ordre :.....
N˚de série :.....

الجمهورية الجزائرية الديموقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
جامعة الشهيد حمة لخضر الوادي
University of Echahid Hama Lakhdar El-Oued
كلية العلوم الدقيقة
Faculty of Exact Sciences
قسم الإعلام الآلي
Department of Computer Science

Final thesis

ACADEMIC MASTER

Field : Mathematics and Computer Science
Branch : Automated notification
Specialization : Artificial Intelligence and Data Science

Theme

Development of a data compression method by merging data blocks

Presented by :

- **Bouhafs Rokia**
- **Bouzidi Abir**

Discussed in 26 May 2025 by the jury :

Dr : ZAIZ Faouzi	MCA	Supervisor	Univ.El Oued
Dr : lAOUID ABDELKADER	M	President	Univ.El Oued
Dr :SASY AMINA	M	Examiner	Univ.El Oued

Academic year : 2024/2025

Dedication

First of all, we praise Allah, the Most Gracious and the Most Merciful, for His endless blessings and love. From our hearts, we dedicate this thesis to:

Our loving mother, who has always been our support Our father, for his patience and constant encouragement, Our dear sister, who never hesitated to offer us advice and support.
Everyone dear to our hearts, And all our friends, who have always stood by our side throughout this journey.

Acknowledgement

I would like to express my sincere gratitude to my esteemed supervisor ***Zaiez Faouzi*** *for his continuous support and encouragement throughout this journey. His expertise and patience have been truly invaluable.*

I also extend my heartfelt thanks to all the professors in the Computer Science Department at the University of El Oued for sharing their knowledge and fostering a stimulating learning environment during my academic journey.

Finally, I would like to thank all the members of the evaluation committee who agreed to assess my work.

Abstract

The digital world has witnessed a massive growth in data volume, creating a need for efficient techniques for storing and transmitting information. Due to limited resources, data compression techniques have been proposed to reduce the amount of data stored or transmitted. Data compression concepts contribute to the optimization of storage space and communication bandwidth, which has led to the development of numerous approaches from various perspectives.

This work focuses on the implementation of a lossless data compression method based on data block merging to improve compression ratios. The method merges the contents of two data blocks to achieve a better compression ratio. The proposed approach was tested on the Prague Corpus dataset, which includes various file types such as databases, software scripts, and medical images. The results showed significant improvements in compression ratios at the block level, but recorded negative values at the file level, highlighting the need for preprocessing algorithms before implementing the merging process.

This study opens new avenues for improving data compression in practical applications, with the potential to expand the use of artificial intelligence techniques to achieve higher compression ratios in the future.

Keywords: Data compression, Redundancy reduction, Compression algorithms,Compression datasets.

Résumé

Le monde numérique a connu une croissance considérable du volume de données, créant un besoin de technologies efficaces pour stocker et transmettre des informations. En raison de ressources limitées, des techniques de compression de données ont été proposées pour réduire la taille des données stockées ou transmises. Les concepts de compression de données contribuent à améliorer l'utilisation de l'espace de stockage et de la bande passante de communication, ce qui a conduit au développement de nombreuses méthodes sous plusieurs angles.

Ce travail se concentre sur la mise en œuvre d'une méthode de compression de données sans perte basée sur la fusion de blocs de données pour améliorer le taux de compression. La méthode combine le contenu de deux blocs de données pour obtenir un meilleur taux de compression. L'approche proposée a été testée sur un ensemble de données : Prague Corpus, qui comprend différents types de fichiers tels que des bases de données, des scripts logiciels et des images médicales. Les résultats ont montré une bonne amélioration des taux de compression au niveau du bloc, mais ont enregistré des valeurs négatives au niveau du fichier, soulignant la nécessité d'algorithmes de prétraitement avant d'effectuer le processus de fusion.

Cette étude ouvre de nouveaux horizons pour améliorer la compression des données dans les applications pratiques, avec le potentiel d'étendre l'utilisation des techniques d'intelligence artificielle pour atteindre des taux de compression plus élevés à l'avenir.

Mots-clés : Compression de données, Réduction de redondance, Algorithmes de compression, ensembles de données de compression.

الملخص

شهد العالم الرقمي نمواً هائلاً في حجم البيانات، مما أدى إلى الحاجة إلى تقنيات فعّالة لتخزين المعلومات ونقلها. ونظراً لمحدودية الموارد، تم اقتراح تقنيات ضغط البيانات لتقليل حجم البيانات المخزنة أو المنقولة. إذ تساهم مفاهيم ضغط البيانات في تحسين استغلال مساحة التخزين وعرض النطاق الترددي للاتصال، الأمر الذي أدى إلى تطوير العديد من الأساليب من زوايا متعددة.

يركز هذا العمل على تنفيذ طريقة ضغط بيانات من نوع بدون فقدان تعتمد على دمج كتل البيانات بهدف تحسين نسبة الضغط. تقوم الطريقة بدمج محتوى كتلتين من البيانات لتحقيق نسبة ضغط أفضل. تم اختبار النهج المقترح على مجموعة بيانات هي: Prague Corpus، والتي تتضمن أنواع من الملفات مثل قواعد البيانات، نصوص البرمجيات، والصور الطبية. أظهرت النتائج تحسناً جيداً في نسب الضغط على مستوى الكتل، لكنها سجلت قيماً سلبية على مستوى الملفات، مما يُبرز الحاجة إلى خوارزميات معالجة أولية قبل تنفيذ عملية الدمج.

تفتح هذه الدراسة آفاقاً جديدة لتحسين ضغط البيانات في التطبيقات العملية، مع إمكانية التوسع في استخدام تقنيات الذكاء الاصطناعي لتحقيق نسب ضغط أعلى في المستقبل.

الكلمات المفتاحية: ضغط البيانات، تقليل التكرار، خوارزميات الضغط، مجموعات بيانات الضغط.

Contents

List of Figures

List of Tables

Introduction

In the modern era of digital information, the exponential growth of data across various domains—ranging from multimedia and scientific measurements to enterprise systems—has created an urgent demand for efficient data storage and transmission. Data compression techniques play a pivotal role in addressing this challenge by reducing the size of data representations without significant loss of information. As data volumes continue to rise, the need for more effective and adaptive compression methods has become increasingly critical to address the following challenges:

- **Storage Capacity Challenge**: How can high compression ratios be achieved without sacrificing data quality, especially in sensitive areas such as medical images or financial records?
- **Processing Speed Challenge**: Is it possible to develop compression algorithms that balance execution speed (important in real-time applications such as live streaming) with compression efficiency?
- **Heterogeneous Data Challenge** : How can we manage the compression of diverse datasets (text, images, video, databases) within a single framework without losing the distinctive characteristics of each type?

Therefore, we propose this work to investigate the possibility of developing a method to improve data compression. This thesis proposes a novel compression method that leverages the concept of combining data blocks to enhance overall compression efficiency. Unlike traditional compression techniques that process data in isolation or rely heavily on predefined models, the proposed approach dynamically identifies and merges structurally or statistically similar data blocks, thereby exploiting redundancy at a higher abstraction level.
By focusing on inter-block relationships, this method aims to achieve better compression ratios while maintaining computational efficiency. The rest of this work outlines the theoretical basis of the method, details the design and implementation of the compression algorithm, and presents a comprehensive evaluation based on synthetic and real-world datasets. Through these investigations, the thesis demonstrates the potential advantages and limitations of block-based combination strategies in data compression.

Chapter 1

introduction to data compression

Introduction

Data compression is considered a fundamental concept in the field of information technology, aiming to reduce data size while preserving its integrity and content . The development of compression techniques began with Shannon's information theory, which introduced the concepts of entropy and redundancy [35].

Subsequently, various algorithms emerged to enhance the efficiency of data compression, particularly in files and images, contributing to better performance and significant size reduction. In the modern era, this field is witnessing rapid advancement through the integration of artificial intelligence and machine learning techniques into compression methods, with the goal of optimizing performance and minimizing data size as much as possible.

In this chapter, I have addressed the general concept of data compression, its various types — including lossy and lossless compression — as well as the fundamental principles it relies upon. Additionally, I reviewed the most prominent traditional and modern algorithms used in this field.

1.1 Definition of Data Compression

Data compression is the process of reducing the size of data by eliminating redundancies and encoding it more efficiently. This reduction is achieved using specialized algorithms that exploit patterns, repetitions, or unnecessary information in the data. Compression allows for both lossless and lossy methods, where the former preserves all original data, and the latter sacrifices some fidelity for higher compression rates [33].

1.2 Importance of Data Compression

Data compression plays a vital role in many domains, as it improves the efficiency of data storage, transmission, and processing. By reducing the size of data, compression enables more efficient storage and cheaper data transmission, making it easier to manage large volumes of information [33, 36]. Moreover, compression enhances system performance by reducing the amount of data that must be processed, enabling faster processing times and better resource utilization in fields like big data analytics and AI [14, 33].

1.2.1 Reducing Storage Requirements

One of the main advantages of data compression is its ability to significantly reduce the amount of disk space needed to store data. This is particularly important in environments that generate large datasets, such as data centers and scientific computing systems. As noted by Storer [36], reducing redundancy through compression minimizes the cost of storage infrastructure and helps prevent storage overload.
Compression also simplifies storage management by reducing the number of files or blocks that need to be handled by file systems, leading to improved organization and reduced fragmentation [33].

- **Example:** A company that stores raw surveillance footage totaling 10 TB monthly can apply video compression to reduce the storage needs to around 4–5 TB, saving costs and optimizing storage infrastructure.

1.2.2 Improving Data Transmission Speed

Data compression improves the speed and efficiency of data transmission by reducing the number of bits that must be sent over the network. According to a technical whitepaper by Netflix [6], advanced video compression algorithms enable high-quality streaming even on low-bandwidth connections, which enhances the user experience.
In addition, compressed data packages reduce the time required for uploads and downloads in cloud-based services and mobile applications, especially in bandwidth-constrained environments [33].

- **Example:** A 4K video originally sized at 20 GB can be compressed to 5 GB using H.265 encoding, allowing it to stream smoothly even on a 10 Mbps internet connection.

1.2.3 Optimizing System Performance

Compression directly impacts the performance of systems by reducing the volume of data that processors and memory have to handle. This enables faster input/output (I/O) operations and decreases the load on hardware resources. Sayood [33] explains that compression can make applications more responsive by lowering the processing burden during data retrieval and transfer.
Google reported that using compression across its storage and transmission pipelines helped reduce energy usage and improve throughput, especially in large-scale distributed systems [14].

- **Example:** In big data platforms like Hadoop, using compressed files (e.g., Snappy or Gzip) reduces the amount of data read from disk, speeding up MapReduce jobs by up to 40%.

1.2.3.1 Advantages

- **A. Reduced Storage Space:** Compressed data takes up less space on hard drives, cloud storage, or memory devices, allowing for more efficient storage of large datasets or files [33].
- **B. Faster Data Transmission:** Smaller file sizes mean that data can be sent more quickly over networks, improving the speed of downloads, uploads, and streaming [6].
- **C. Lower Bandwidth Usage:** By reducing the amount of data that needs to be transferred, compression helps decrease the load on network bandwidth, which is particularly important for mobile or limited-bandwidth environments [33].
- **D. Cost Savings:** Less storage and lower bandwidth usage can translate into reduced operational costs for businesses and individuals.

- **E. Improved Performance:** Applications and systems that handle compressed data can benefit from faster read/write operations and better performance, especially when dealing with large-scale data [33].
- **F. Enhanced File Management:** Compressed files can be more manageable, especially when grouping multiple files into a single compressed archive (e.g., ZIP files), aiding organization and portability.
- **G. Data Integrity and Security:** Some compression formats include checksums or encryption features, helping ensure data integrity and adding a layer of protection during transmission or storage.

1.2.3.2 Drawbacks

- **A. Processing Resource Consumption:** Compression and decompression processes require computational resources, which may slow down the performance of systems with limited capabilities [33].
- **B. Quality Loss (In Lossy Compression):** In some cases, compression may lead to irreversible data loss, reducing the quality of images, audio, or video.
- **C. Compatibility Issues:** Not all systems or applications support every compression format, which can lead to issues when sharing or archiving data.
- **D. Latency in Real-Time Applications:** Compression and decompression processes may introduce latency in applications that require immediate response, such as live streaming or online gaming.

1.3 Principles of Data Compression

Data compression relies on a set of core principles that enable reduction of data size while maintaining essential information. These principles form the foundation of both lossless and lossy compression techniques. The main principles include **redundancy reduction**, **entropy coding**, and **pattern recognition**. Each is discussed below in detail with examples, benefits, limitations, and a corresponding flowchart.

1.3.1 Redundancy Reduction

Redundancy refers to the repetition of data or patterns that do not contribute to new information. By removing redundant information, data size can be minimized without losing essential content. This principle is fundamental in lossless compression methods, where every bit of the original data is preserved [32].

1. Example: Consider a string such as `AAAAABBBCCDAA`. The character 'A' appears five times, and thus the sequence can be encoded more efficiently. Run-Length Encoding (RLE) is a simple algorithm that captures this redundancy. It compresses the string to `5A3B2C1D2A`, significantly reducing the size by replacing repetitions with counts.

2. Advantages:

A) Preserves all original data, making it ideal for lossless compression.

B) Very effective for data with repeated patterns (e.g., images, text).

C) Simple to implement and understand.

3. Disadvantages:

A) Less effective on random or already compressed data, such as encrypted files or highly complex content [32].

B) Compression ratios may not be as high for diverse or unpredictable data.

4. Applications: Run-Length Encoding (RLE) is commonly used in bitmap images (BMP) and in fax machine protocols.

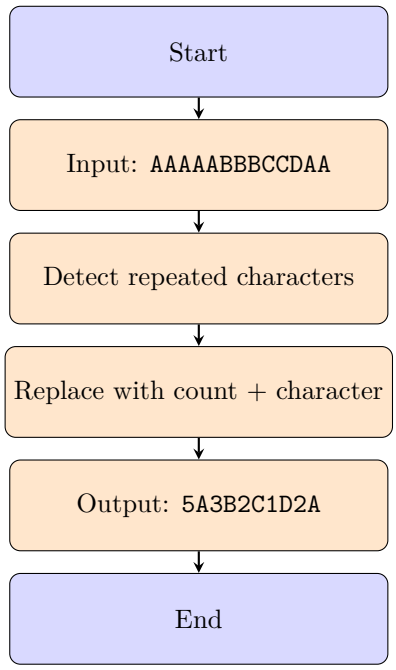

Figure 1.1: Flowchart of Redundancy Reduction Using RLE

1.3.2 Entropy Coding

Entropy coding is a lossless data compression technique based on the frequency distribution of symbols. The key concept is to assign shorter codes to more frequent symbols and longer codes to less frequent symbols. This principle reduces the average length of the code required to represent a dataset [32].

1. Example: In Huffman coding, symbols are assigned binary codes based on their frequency in the data. For instance, if 'A' appears 50% of the time and 'B' appears 25%, 'A' will get a shorter binary code than 'B', reducing the total number of bits required to represent the data [16].

2. Advantages:

A) Efficient for data with varying symbol frequencies [32].

B) Can be used with both lossless and lossy compression techniques [16].

C) Works well in a wide range of applications, including text and image compression [32].

3. Disadvantages:

A) May require complex encoding and decoding algorithms [32].

B) Efficiency can be reduced when symbol frequency is uniform or highly unpredictable.

4. Applications: Entropy coding is used in algorithms like Huffman coding (JPEG, ZIP) and Arithmetic coding (PDF, TIFF) [20].

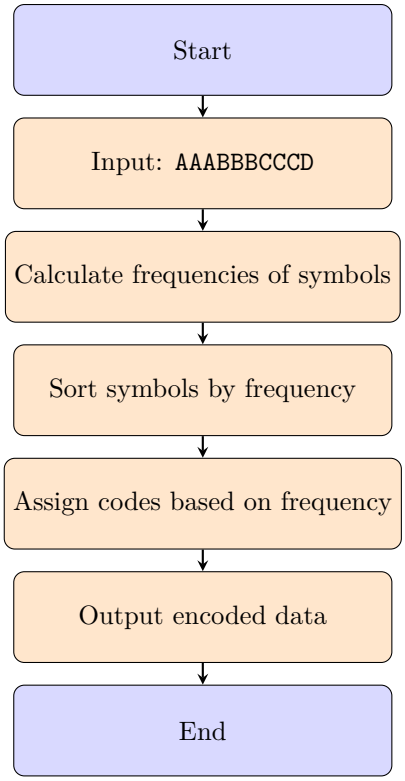

Figure 1.2: Flowchart of Entropy Coding Using Huffman Coding

1.3.3 Pattern Recognition and Prediction

Pattern recognition in data compression involves identifying recurring structures within the data and predicting the next values or sequences based on previous information. This is commonly applied in lossy compression methods like video compression (H.264, H.265) and in algorithms like LZ77 [47].

1. Example: In video compression, the difference between consecutive frames is predicted. Instead of encoding the entire frame, only the change from the previous frame is stored, reducing the amount of data required [34].

2. Advantages:

A) High compression ratios, especially for video and audio data [34].

B) Reduces storage and transmission costs by encoding only differences.

C) Useful for large datasets with predictable trends.

3. Disadvantages:

A) May result in quality loss (lossy compression) [34].

B) Requires complex algorithms for prediction and error correction.

C) Ineffective for data without recognizable patterns (e.g., random noise).

4. Applications: Pattern recognition is a core component of video codecs like HEVC (H.265) and audio compression algorithms like MP3 [34].

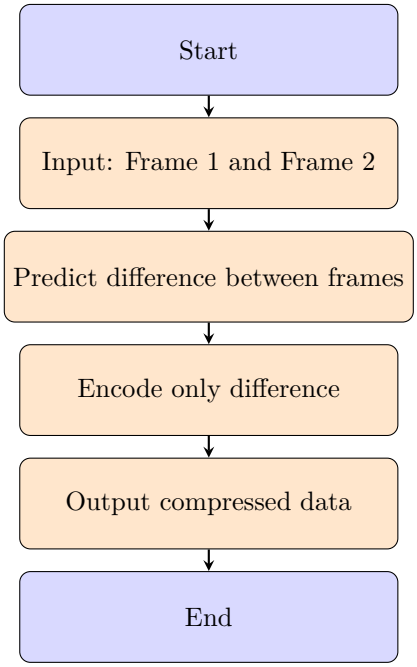

Figure 1.3: Flowchart of Pattern Recognition in Video Compression (HEVC)

1.4 Types of Data Compression

Compression comes in two main types: **lossless** and **lossy**. Each type has different uses depending on whether you need to keep all the original details or if you can afford to lose some information to save more space [48].

1.4.1 Lossless Compression

Lossless compression reduces the size of data without losing any information. When you decompress the file, it returns to its exact original form [48], enabling the image to take up less space without any discernible loss in picture quality [3].

- Techniques and Mechanisms of Data Compression : Lossless compression finds patterns or repeated parts in the data and replaces them with shorter representations.

For example, if a document has the word `"the"` many times, lossless compression might store `"the"` as a shorter code and replace each occurrence with that code [48].

- Applications and Areas of Data Compression : Lossless compression is used for text files, software programs, and any data where it's important to keep everything exactly the same, like ZIP files or PNG images [48].

Figure(1.4) presents a series of images demonstrating how lossless compression affects file size. The first image shows the original JPEG (already lossy), and subsequent versions illustrate lossless compression at 50%.

Figure 1.4: Lossless compression results [48].

1.4.2 Lossy Compression

Lossy compression reduces file size by removing some information that might not be noticed or needed [48]. Here, an algorithm scans image files and reduces their size by discarding information considered less important or undetectable to the human eye [3].

- Techniques and Mechanisms of Data Compression : Lossy compression removes parts of the data that are less important or that people are less likely to notice. For example, in an image, lossy compression might remove colors that are very similar, making the file smaller but slightly lowering the quality [48].

- Applications and Areas of Data Compression : Lossy compression is often used for images, audio, and videos, like JPEG photos, MP3 music files, and streaming videos. It's useful when saving space is more important than keeping every tiny detail [48].

Illustration: Figure(1.5) compares different compression techniques applied to the same image. The original file is 976 KB, while the lossless version is reduced to 834 KB and the lossy one to just 212 KB.

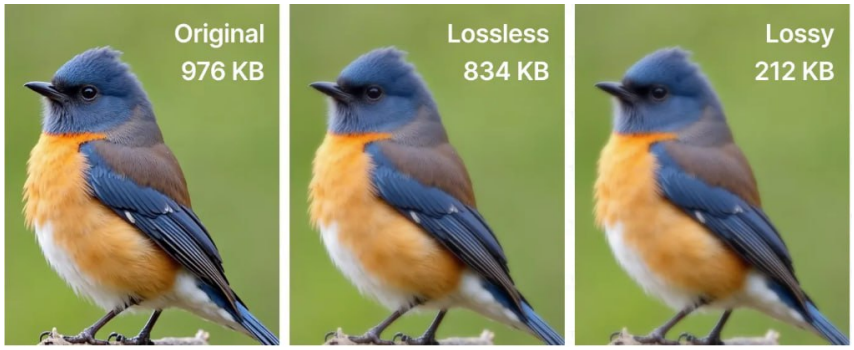

Figure 1.5: File size comparison: Original (976 KB) vs. compressed versions (834 KB lossless, 212 KB lossy) [48].

1.4.3 Comparison between Lossless and Lossy Compression

This comparison Table 1.1 highlights the fundamental differences between lossless and lossy compression techniques. The table systematically organizes key aspects including definitions, typical use cases, file

formats, algorithms, and the respective advantages and limitations of each approach. This analysis is particularly valuable for selecting the appropriate compression method based on specific requirements for data integrity versus file size reduction [38].

Compression Methods Comparison		
Aspect	**Lossless Compression**	**Lossy Compression**
Definition	Preserves all original data when decompressed	Permanently removes some data during compression
Use Cases	• Financial records • Medical imaging • Text documents	• Web images • Streaming media • Digital photography
Formats	• Images: PNG, BMP, GIF • Audio: FLAC, WAV • Archives: ZIP	• Images: JPEG, WebP • Audio: MP3, AAC • Video: H.264/265
Algorithms	• Run-Length Encoding • Lempel-Ziv-Welch (LZW) • Huffman Coding	• Discrete Cosine Transform • Wavelet Transform • Fractal Compression
Advantages	• Perfect reconstruction • No quality loss	• High compression ratios • Smaller file sizes
Limitations	• Larger files • Limited compression	• Quality loss • Compression artifacts

Table 1.1: Detailed comparison between lossless and lossy compression techniques [38]

1.4.4 Data Compression Algorithms

Data compression relies on a variety of algorithms and techniques, ranging from traditional methods to modern approaches for each type of compression, which benefit from advancements in artificial intelligence (AI). This section provides an overview of both traditional and modern algorithms related to each compression type, followed by a comparison in terms of efficiency, complexity, and suitability for various applications.

1.4.5 Lossless Data Compression Algorithms

In lossless compression, we will explore both modern and traditional algorithms.

A) Traditional Algorithms :

In traditional lossless data compression algorithms, the key techniques used by researchers to reduce data size while preserving its quality can be identified.

A.1) Huffman Coding : Huffman Coding is an algorithm used for lossless data compression [42],The idea is to assign variable-length codes to the input characters, with the length of the assigned codes being based on the frequency of occurrence of the characters in the original sequence [11].

1. **Steps for Constructing a Huffman Tree :**

   The construction of a Huffman tree involves several key steps, which are outlined below [11]:

   A) Create leaf nodes for each character: Each character in the message is treated as a leaf node, along with its frequency of occurrence. Nodes are sorted by ascending frequency.

   B) Extract two nodes with the lowest frequencies: Select and remove the two nodes with the smallest frequencies.

   C) Create a new internal node: Combine the two nodes into a new node, with frequency equal to their sum.

   D) Set child nodes: Assign the first node as the left child and the second as the right child.

   E) Reinsert the internal node: Add the new node back into the list.

   F) Repeat the process: Continue until only one node remains.

   G) Finalize the tree: The last remaining node becomes the root of the Huffman Tree.

2. **Huffman Coding Example:**

   To illustrate Huffman Coding, consider the input string: **AABACD**. First, we count the frequency of each character:

Character	**Frequency**
A	3
B	1
C	1
D	1

   The steps to build the Huffman Tree are as follows:

   A) Create leaf nodes for each character with its frequency.

   B) Combine the two least frequent nodes: C(1) and D(1) $\rightarrow$ new node with frequency 2.

   C) Combine B(1) with the new node $\rightarrow$ node with frequency 3.

   D) Combine this node with A(3) $\rightarrow$ final root node with frequency 6.

   The resulting Huffman codes are:

   - A $\rightarrow$ 1
   - B $\rightarrow$ 01
   - C $\rightarrow$ 000
   - D $\rightarrow$ 001

The string **AABACD** becomes: `1 1 01 1 000 001` (significantly compressed).

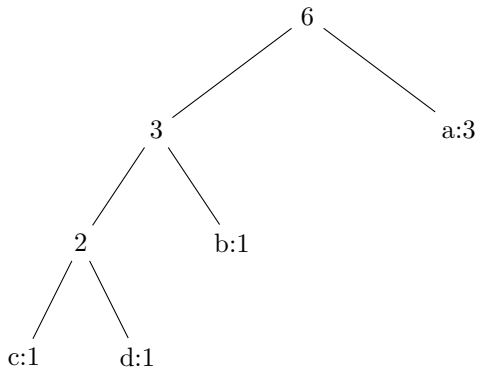

Figure 1.6: Example of Huffman Tree for the string `AABACD` [11].

3. **Flowchart of huffman coding algorithme** This flowchart outlines the Huffman coding process: first it counts each character's frequency and inserts them into a min-heap; then, while more than one node remains, it removes the two lowest-frequency nodes, combines them into a new internal node, and reinserts this node until only the root remains. The final node represents the completed Huffman tree, ready for code generation.

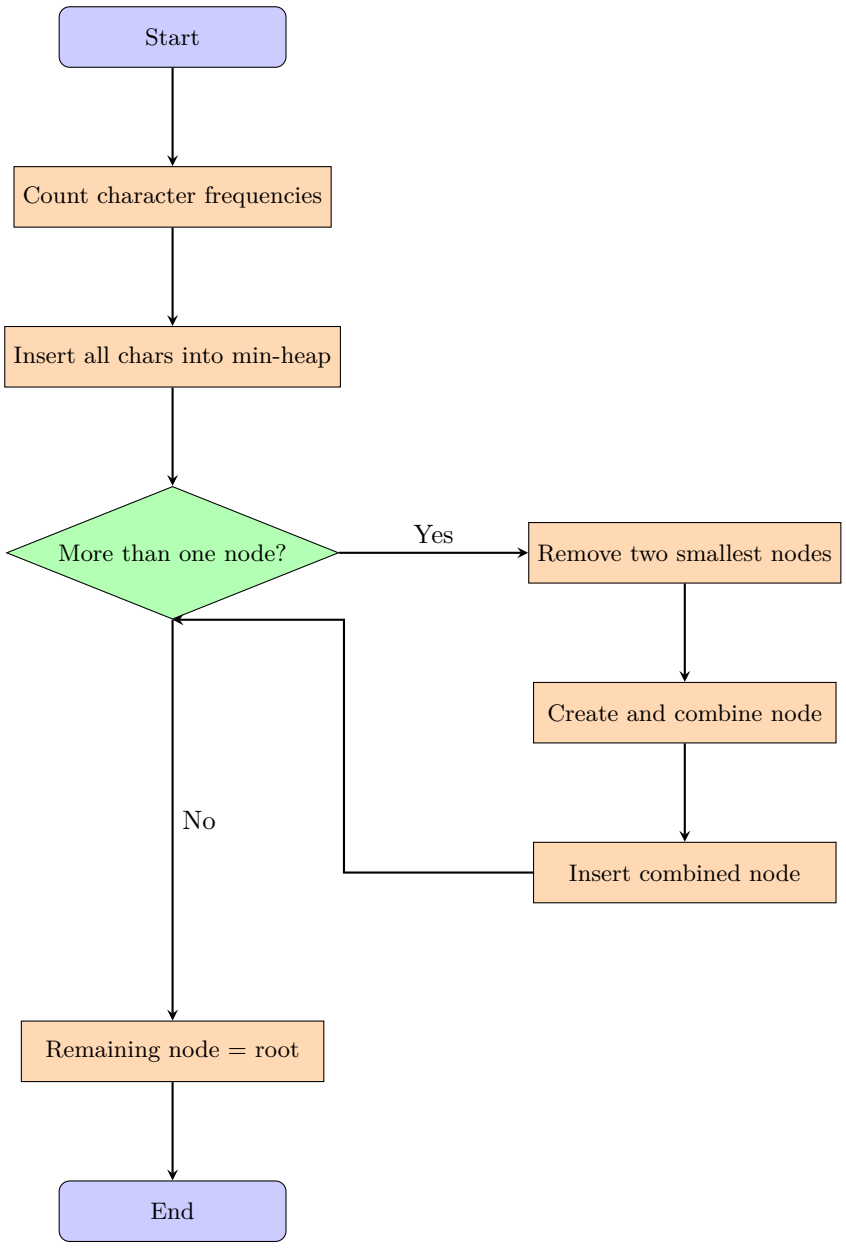

Figure 1.7: Flowchart of the Huffman Coding algorithm [11].

A.2) LZ77 Algorithm : The LZ77 algorithm is a lossless data compression algorithm that replaces repeated substrings with references to earlier occurrences in the data stream [43]. The main idea is to maintain a sliding window and find the longest match of a substring within the window. If a match is found, the substring is replaced with a reference to the earlier occurrence, otherwise, the symbol is directly added to the output.

1. **Steps for LZ77 Compression:**

   The LZ77 algorithm works by following these key steps, outlined below [43]:

   A) Initialize the sliding window: The algorithm begins with a sliding window that holds a certain portion of the text.

   B) Find the longest match: Search for the longest substring within the window that matches the current input.

C) Output a triplet: The algorithm outputs a triplet consisting of a pair (offset, length) representing the matched substring and the next symbol that follows the match.

D) Update the window: The sliding window is updated by moving forward, including the newly processed symbol.

E) Repeat the process: Continue searching for matches, outputting triplets, and updating the window until the entire input is processed.

2. **LZ77 Compression Example:**

To demonstrate LZ77, consider the input string: **ABABABABA**. The sliding window and the steps to compress the string are as follows:

Offset	**Length**	**Next Symbol**	**Triplet**
0	0	A	(0,0,A)
1	0	B	(0,0,B)
2	2	A	(1,2,A)
1	2	B	(1,2,B)
2	2	A	(1,2,A)

The string **ABABABABA** is compressed into the sequence: `(0,0,A) (0,0,B) (1,2,A) (1,2,B) (1,2,A)`.

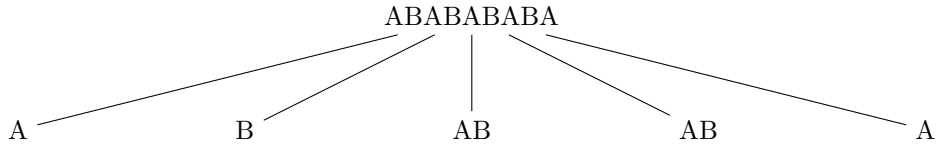

Figure 1.8: Example of LZ77 Compression for the string `ABABABABA` [43].

3. **Flowchart of LZ77 algorithm**

This flowchart illustrates the LZ77 compression process: the algorithm searches for the longest match in the sliding window, outputs a triplet (offset, length, symbol), and continues the process until the entire input is compressed.

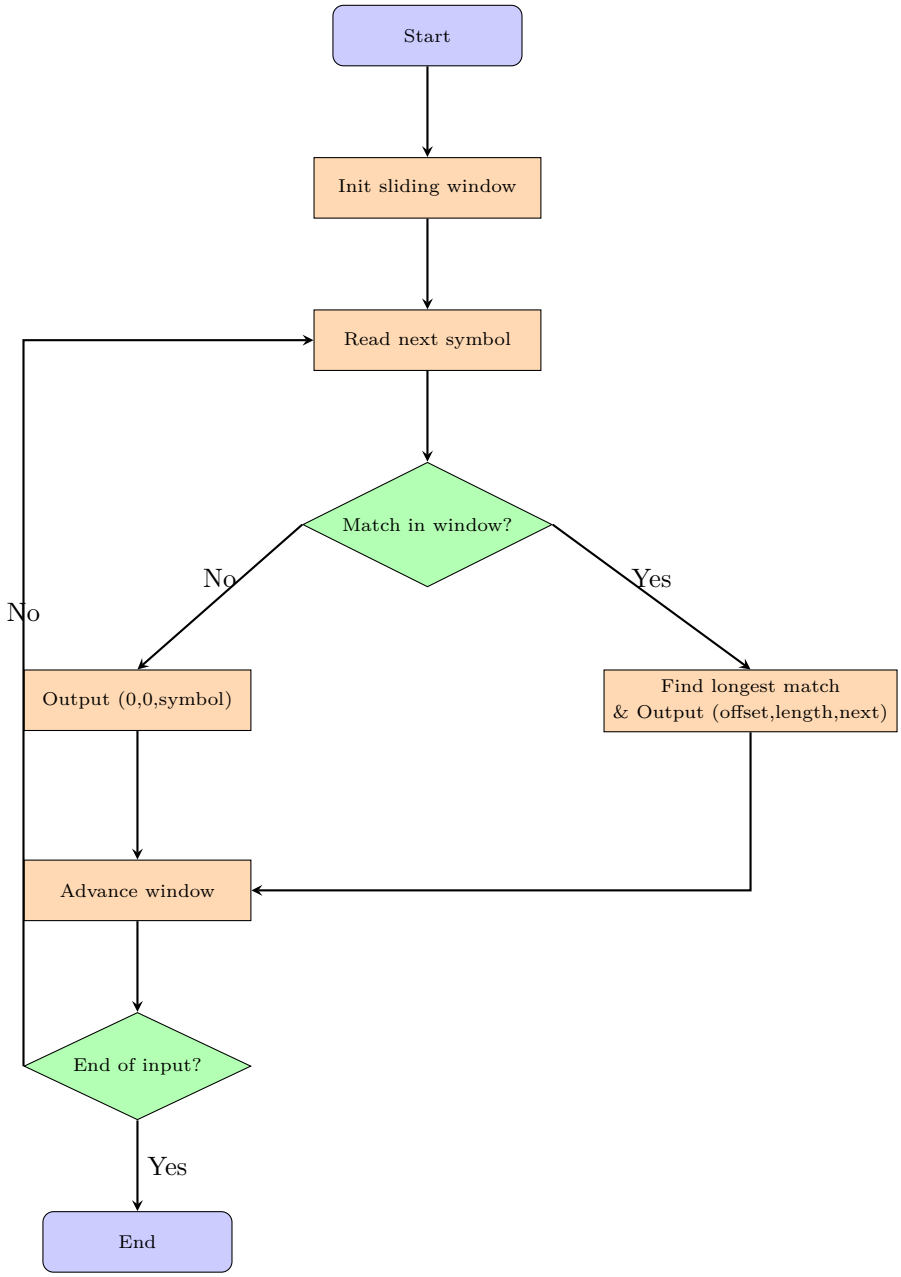

Figure 1.9: Flowchart of the LZ77 compression algorithm [12].

B) Modern Algorithms :

Modern data compression algorithms, such as LZMA, build upon traditional methods by integrating advanced techniques like dictionary encoding and entropy coding [44]. These approaches offer better compression ratios and efficiency, especially for large and structured data. Recent developments also include AI-based methods that adaptively learn data patterns, pushing the boundaries of compression performance.

B.1) LZMA (Lempel–Ziv–Markov chain algorithm): LZMA is a powerful lossless compression algorithm known for achieving high compression ratios by combining dictionary compression with sophisticated entropy coding methods [44]. It is widely used in formats such as 7z (7-Zip) and is known for its efficient handling of large files and repetitive data.

1. **Steps of LZMA Compression:**

   The LZMA algorithm consists of multiple steps that combine dictionary-based compression with range coding and filtering mechanisms:

   A) **Dictionary Matching:** The algorithm searches for the longest match of the current byte sequence in a sliding window (dictionary).

   B) **Position Encoding:** Matches are encoded by distance (how far back the match is) and length (how many bytes matched).

   C) **Literal Encoding:** If no match is found, the byte is encoded as a literal using context-based probability modeling.

   D) **Range Coding:** All data (matches and literals) is passed through an entropy encoder known as range coder, which achieves high compression efficiency.

   E) **Optional Filtering:** Filters such as BCJ (Branch Converter for Jump instructions) may be applied for specific file types like executables.

2. **LZMA Flowchart:**

   Below is a flowchart that summarizes the core structure of LZMA compression.

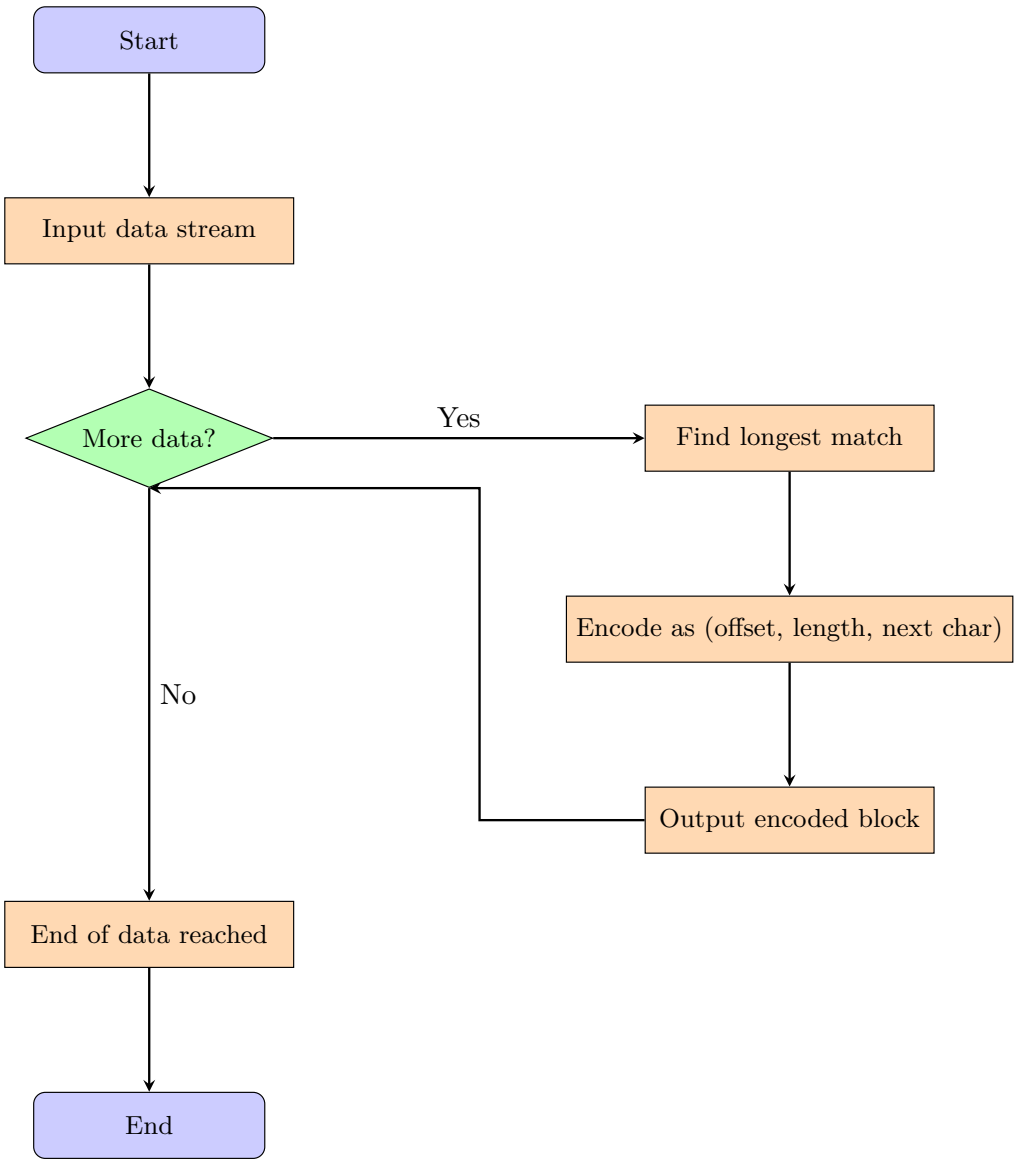

Figure 1.10: Flowchart of the LZ77 Compression Algorithm [43].

B.2) Brotli: Brotli is a modern lossless compression algorithm developed by Google, primarily used for compressing web content such as HTML, CSS, and JavaScript [15]. It combines dictionary-based compression (like LZ77), Huffman coding, and a context modeling approach to achieve high compression ratios and fast decompression speeds.

1. **Steps of Brotli Compression:**

   The Brotli compression process includes the following stages [15]:

   A) **Input parsing:** The input is parsed and divided into blocks for processing.

   B) **Backward references:** Brotli searches for repeated substrings using a sliding window, similar to LZ77.

   C) **Static dictionary:** Frequently used words and phrases are replaced using a prebuilt static dictionary.

   D) **Context modeling:** Context information is used to improve entropy coding decisions.

   E) **Huffman coding:** Final encoding is done using Huffman trees tailored to each block.

2. **Brotli Example:**

   A text file containing repetitive patterns and common phrases (e.g., HTML tags) can be significantly compressed using Brotli due to its dictionary and entropy coding optimizations.

3. **Benefits of Brotli:**

   A) Higher compression ratios than Gzip at similar speeds.

   B) Designed for web performance (used in HTTP compression).

   C) Efficient decompression for fast content delivery.

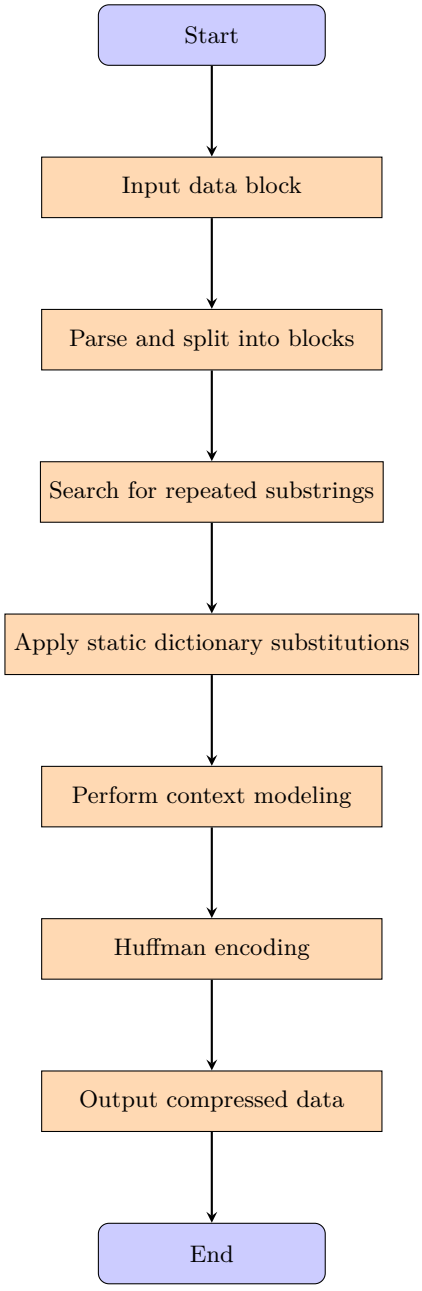

Figure 1.11: Flowchart of the Brotli compression algorithm [15].

1.4.6 Lossy Data Compression Algorithms

In lossy compression, we will explore both modern and traditional algorithms.

A) Traditional Algorithms :

In traditional lossy data compression algorithms, the key techniques used by researchers aim to significantly reduce data size by removing less perceptible information, while maintaining acceptable quality for the intended use.

A.1) JPEG:

JPEG (Joint Photographic Experts Group) is one of the most widely-used lossy image compression standards, introduced in the early 1990s. It is designed to efficiently compress photographic images by removing visually less important information, achieving substantial size reduction while maintaining acceptable visual quality [45]. JPEG is based primarily on the Discrete Cosine Transform (DCT) and remains a dominant format for digital images, especially on the web.

1. **Steps of JPEG Compression:**

   The JPEG compression process consists of the following steps [45]:

   A) **Color space conversion:** The input image is converted from RGB to YCbCr color space, separating luminance (Y) from chrominance (Cb and Cr) components.

   B) **Downsampling:** The chrominance channels (Cb and Cr) are downsampled because the human eye is less sensitive to color details than to brightness.

   C) **Block-based DCT:** The image is divided into 8×8 pixel blocks. Each block undergoes a Discrete Cosine Transform (DCT) to convert spatial pixel values into frequency coefficients.

   D) **Quantization:** The DCT coefficients are quantized using a quantization matrix, reducing the precision of less important high-frequency components to achieve data reduction.

   E) **Entropy coding:** The quantized coefficients are encoded using entropy coding techniques such as Huffman coding or arithmetic coding to achieve further compression.

2. **JPEG Example:**

   A standard digital photograph can be compressed using JPEG with a quality setting of 75%, achieving a file size reduction of up to 10:1 compared to the original, while preserving visual quality suitable for most display devices.

3. **Benefits of JPEG:**

   A) Simple and fast compression and decompression processes.

   B) High compatibility across web, mobile, and digital cameras.

   C) Effective compression for natural images and photographs.

   D) Adjustable compression level to balance between image quality and file size.

A.2) MP3 Compression Algorithm : MP3 (MPEG-1 Audio Layer III) is a widely used lossy audio compression standard that exploits perceptual psychoacoustic models to discard inaudible information while preserving sound quality [46].

1. **Steps of MP3 Compression:**

   The MP3 encoding process typically involves these stages [46]:

A) **Psychoacoustic analysis:** Analyze the input signal to determine masking thresholds.

B) **Filterbank/MDCT:** Split the signal into sub-bands (polyphase filterbank) and apply a Modified Discrete Cosine Transform (MDCT).

C) **Quantization:** Quantize the transform coefficients based on the psychoacoustic model.

D) **Huffman coding:** Losslessly entropy-encode the quantized coefficients.

E) **Bitstream formatting:** Package encoded data into MP3 frames with headers, side-information, and CRC.

2. **MP3 Example (simplified):**

   A 3-minute stereo WAV file (44.1 kHz, 16-bit) is about 31.8 MB uncompressed. When encoded to MP3:

   A) At 128 kbps $\rightarrow \approx 3$ MB (compression $\approx 10:1$)

   B) At 320 kbps $\rightarrow \approx 7.5$ MB (better quality, compression $\approx 4:1$)

3. **Benefits of MP3:**

   A) Good trade-off between quality and file size.

   B) Universally supported by audio players and devices.

   C) Adjustable bitrate and VBR for flexible encoding.

B) Modern Algorithms :

Modern lossy compression algorithms, such as JPEG 2000 and HEVC (H.265), build upon traditional transform-coding methods by integrating advanced techniques like wavelet transforms, motion-compensated prediction, and adaptive quantization [18, 19]. These methods achieve very high compression ratios by discarding perceptually irrelevant information, especially in image and video data. More recently, AI-driven codecs (e.g. end-to-end trained neural compressors) have emerged; they learn content-specific features and allocate bits optimally to push the limits of quality at low bitrates [5].

B.1) HEVC (H.265) :

HEVC (High Efficiency Video Coding), also known as H.265, is a modern lossy video compression standard designed to provide significantly better data compression at the same level of video quality as its predecessor, H.264 [19]. HEVC achieves higher compression ratios by using advanced techniques such as motion compensation, prediction, and more efficient entropy coding. It also uses larger coding block sizes and improved transform coding, allowing for better compression of high-resolution video.

1. **Steps of HEVC Compression:**

   The HEVC compression process includes the following steps [19]:

   A) **Prediction:** The encoder predicts pixel values based on previously encoded blocks to reduce redundancy.

   B) **Transformation:** The residuals (differences between predicted and actual values) are transformed into frequency coefficients.

   C) **Quantization:** These coefficients are quantized, discarding less important data to achieve compression.

   D) **Entropy coding:** A final step involves encoding the quantized coefficients using techniques like CABAC (Context-based Adaptive Binary Arithmetic Coding), which reduces the data size further.

2. **HEVC Example:**

   In HEVC, an input video frame is divided into blocks. Each block undergoes prediction, transformation, quantization, and entropy coding steps. The result is a highly compressed video stream, which is more efficient than previous codecs like H.264.

3. **Benefits of HEVC:**

   A) Provides up to 50% better compression efficiency than H.264.

   B) Supports higher resolutions (up to 8K).

   C) Widely used in modern video streaming platforms and video conferencing.

B.3) Neural Image Compression (NIC): NIC uses Convolutional Neural Networks (CNNs) and Autoencoders to compress images. The approach involves encoding images into a latent space while retaining essential features [23]. It applies CNNs to extract relevant features and then compresses them using an Autoencoder, resulting in high-quality compression.

1. **Steps of NIC Compression:**

   A) **Feature extraction:** The CNN extracts important image features such as edges and textures.

   B) **Compression:** The Autoencoder compresses these features into a latent space.

2. **NIC Example (simplified):** When compressing a high-resolution image, NIC first identifies key features (e.g., edges, textures) through the CNN, then compresses these features, ensuring that the image quality remains high with minimal loss.

3. **Benefits of NIC:**

   A) Superior compression ratios compared to traditional image formats like JPEG.

   B) High-quality image retention even after aggressive compression.

1.4.7 Comparison Between Traditional and Modern Algorithms

The table compares traditional and modern compression algorithms in terms of time period, key algorithms, strengths, limitations, and applications.

Category	Traditional Algorithms	Modern Algorithms
Time Period	1950s–1990s	2000s–Present
Representative Algorithms	• Huffman Coding • LZ77 • JPEG • MP3	• LZMA • Brotli • HEVC/H.265 • Neural Image Compression
Strengths	• Lower computational complexity • Wide compatibility • Hardware acceleration support	• Higher compression ratios • Content-aware optimization • Adaptive learning capabilities
Limitations	• Fixed compression approaches • Lower compression ratios • Less adaptive to new data types	• High computational requirements • Limited hardware support • Complex implementation
Typical Applications	• General-purpose file compression • Legacy systems • Low-power devices	• 4K/8K video streaming • Cloud storage optimization • AI-powered applications

Table 1.2: Comparison Between Traditional and Modern Compression Algorithms

1.4.8 Neural Compression

Neural compression refers to the use of neural networks, particularly deep learning models, to compress data. Unlike traditional compression methods, which rely on algorithms that reduce data size based on predefined rules and statistical properties, neural compression uses artificial neural networks (ANNs) to learn efficient representations of data for compression.

1.4.8.1 Key Aspects of Neural Compression

A) **Data Representation Learning:** Neural networks, especially convolutional autoencoders or variational autoencoders (VAEs), can learn efficient representations of data, capturing the essential features while discarding redundant information. This allows for more efficient compression than traditional methods like JPEG or MP3 [9].

B) **Lossless and Lossy Compression:**

1. *Lossless Compression:* Neural compression can be designed for lossless compression, where the original data can be exactly reconstructed after compression [49].
2. *Lossy Compression:* More commonly, neural networks are applied to lossy compression, where some of the data is discarded (e.g., high-frequency details in images or audio), achieving much higher compression rates at the cost of some data loss [40].

C) **End-to-End Learning:** A neural compression system can be trained in an end-to-end manner, where the network learns both how to compress (encode) and how to decompress (decode) data [4].

D) **Applications:**

1. *Image Compression:* Neural networks are used to compress images, and convolutional neural networks can improve compression beyond traditional methods like JPEG [22].
2. *Video Compression:* Neural networks can help compress video files by learning to predict and encode pixel sequences [31].
3. *Audio Compression:* For audio, neural networks like deep autoencoders can be used to compress audio files with high quality [27].
4. *Speech and Text Compression:* Neural compression can also be applied to compress speech or text data for efficient transmission [7].

1.4.8.2 Techniques in Neural Compression

A) **Autoencoders:** Autoencoders are neural networks trained to encode data into a lower-dimensional latent space and then decode it back to the original format. Variational autoencoders (VAEs) are commonly used in neural compression tasks because they add a probabilistic layer, making them good for generating compressed representations [21].

B) **Generative Models:** Models like Generative Adversarial Networks (GANs) are also explored for image and video compression. These models generate compressed representations of data and then decode it back using a generator model [13].

C) **Recurrent Neural Networks (RNNs):** For sequential data like time series, audio, or video frames, RNNs or Long Short-Term Memory (LSTM) networks can capture temporal dependencies, enabling efficient compression of sequential data [37].

D) **Transformers:** Transformer models have gained popularity in compression tasks, particularly for their ability to capture long-range dependencies in data, such as in text or video compression [41].

1.4.8.3 Advantages

A) **Higher Compression Ratios:** Neural compression can achieve higher compression ratios compared to traditional methods, particularly in image and video compression [31].

B) **Better Quality:** Neural networks can potentially offer better quality for the same compression ratio by learning more context-aware representations [49].

C) **Flexibility:** Neural compression can be customized for specific types of data (e.g., audio, video, or images) and can be trained for specific use cases or datasets [40].

1.4.8.4 Challenges

A) **Computational Resources:** Training neural compression models requires significant computational resources, especially when working with large datasets [27].

B) **Training Data Requirements:** Neural networks need large amounts of training data to learn good representations for compression, which might not always be available [4].

C) **Generalization:** While neural networks excel at learning specific patterns, ensuring they generalize well across different types of data is a challenge [9].

1.4.9 Comparison between Traditional and Neural Compression

Table presents a feature-wise comparison between traditional compression techniques, and modern neural compression methods [4, 9, 33].

Feature	**Traditional (e.g., JPEG, H.264)**	**Neural Compression**
Adaptability to Data	Fixed rules and hand-crafted algorithms tailored to specific formats.	Learns from data distributions; can be retrained for different data types [9].
Compression Efficiency	Moderate at low bitrates, with visible quality degradation.	High efficiency, especially perceptually, even at lower bitrates [4].
Computational Complexity	Low to moderate; suitable for real-time applications.	High computational demands during training and inference (partially mitigated with optimized architectures) [31].
Real-Time Performance	Excellent due to lightweight algorithms.	Improving with hardware acceleration and model optimization techniques [22].
Generalization	Hand-crafted for specific data types and domains.	Learns generalized features applicable to diverse data distributions [7].

Table 1.3: Feature Comparison: Traditional vs. Neural Compression

1.4.10 Performance Metrics for Data Compression

To evaluate the effectiveness of data compression methods, several key performance metrics are commonly used [4, 31, 33]:

A) **Compression Ratio:** Measures how much the data size is reduced after compression. It is typically expressed as the ratio of the uncompressed size to the compressed size. Higher ratios indicate better compression efficiency.

B) **Compression and Decompression Speed:** Refers to the time taken to compress and decompress data. Fast processing is critical for real-time applications such as video streaming and live data transmission [31].

C) **Quality Metrics:** Especially important in lossy compression, these metrics assess the perceptual or quantitative quality of the decompressed data compared to the original. Common examples include PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity Index) for images and videos [4].

D) **Trade-offs Between Space, Time, and Quality:** Compression systems often balance between achieving smaller file sizes (space), faster processing (time), and preserving data fidelity (quality). The optimal choice depends on the application context and resource constraints [33].

Conclusion

At the end of this chapter, it becomes clear that data compression is one of the fundamental and pivotal techniques in the modern data world, due to its effective role in enhancing storage efficiency and accelerating data transmission, while preserving the quality and integrity of information. Through this chapter, we reviewed the basic principles and commonly used algorithms in this field, which have significantly contributed to the development of various systems and applications across multiple sectors.

In the second chapter, we will present the general framework of this work by outlining the adopted structure of our study.

Chapter 2

System Design

Introduction

Data compression is a fundamental component of information processing, playing a significant role in reducing data size and accelerating its transmission and storage. In this chapter, we highlight the general structure of the method proposed in this research, which aims to efficiently combine compressible and incompressible data blocks before applying the compression process. We also detail the architecture of the proposed system, starting with the files used, the merging and compression stages, how they are implemented, the mechanism for restoring data to its original state, and how the results can be used to train the model.

2.1 General structure of compression

In this section we will present the general framework of the compression process (2.1) that supports our work, as well as a simplified explanation of each element in it.

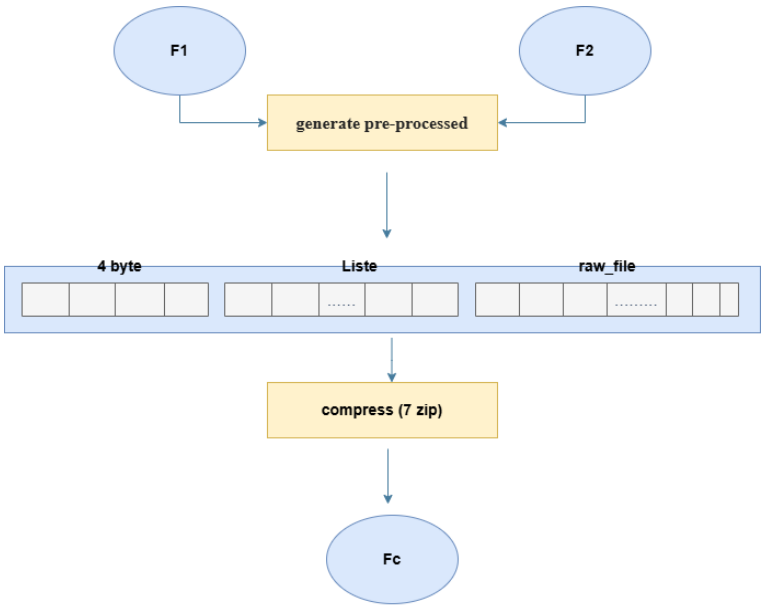

Figure 2.1: General structure of compression

The structure starts with a dataset: F1, which contains compressible files, and F2, which contains incompressible files.
A preprocessor is first created. This then creates a 4-byte array containing the length of the list, the list itself, which specifies the length of the fragment from file F2 (labeled b2), and a raw_file file, which combines the data from files F1 and F2. The data is then compressed using 7zip, producing the compressed file FC.

2.2 General structure of decompression

In this section we will present the general framework of the decompression process (2.2) that supports our work, as well as a simplified explanation of each element in it.

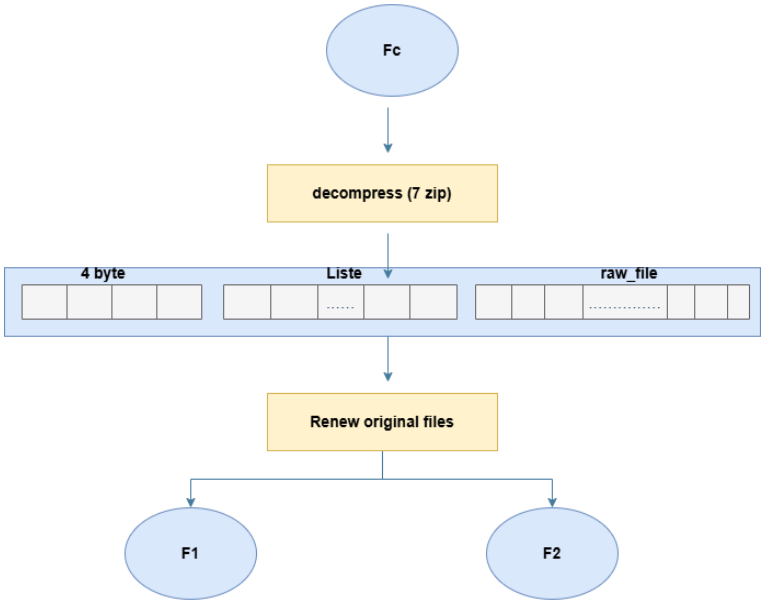

Figure 2.2: General structure of decompression

At the beginning of the diagram, we have the compressed file. We decompress it using 7zip. All information is extracted from the 4-byte array, which contains the length of the list, the length of the F2 portion stored in the list, and the raw_file file. Finally, the original files F1 and F2 are restored to their original state.

2.3 Main code structure

In this section we will show the Main code structure (2.3) of our work.

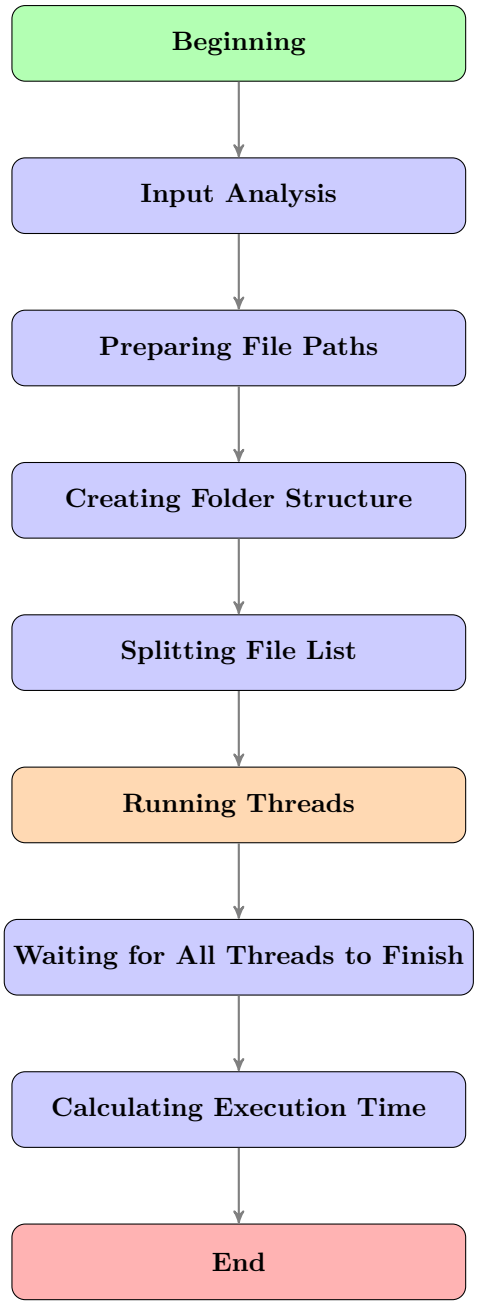

Figure 2.3: Main code structure

The main program acts as a basic organizer for parallel file processing. It first parses the input to determine the number of threads and preparing file paths .
It then initializes the processing environment by creating subfolder structures such as assembly files and CSV files using the make_dirs() function.
It then divides the file list into equal parts and distributes them to the threads running at the same time. Each thread executes the subroutine on its assigned set of files. Finally, the main program waits for all threads to finish using the thread.join() function and calculates the total execution time to measure performance.

2.4 Multithreaded structure

In this section we will introduce the Multithreaded model(2.4) architecture for our work.

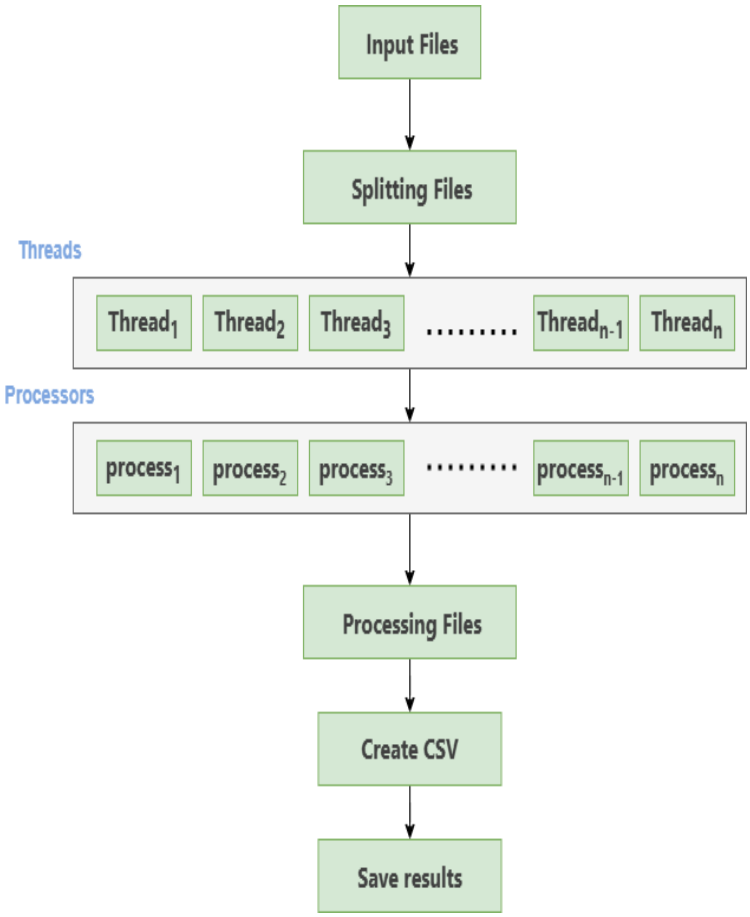

Figure 2.4: Multithreaded structure

The program follows a multi-stage system for highly efficient data processing. The process begins with the input phase for files to be read. This phase then moves to the file partitioning phase, where the files are distributed evenly across a set of threads (Thread1 to Threadn) to ensure optimal processing parallelism.

Each thread operates independently, running a secondary copy of the code (Process1 to Processn) that performs compression and merging operations according to specified algorithms. During the file processing phase, CSV files containing the processed data are created, with detailed results recorded for each compression operation, including performance metrics and data size before and after processing.

All system components work in tandem to ensure maximum utilization of available processing resources, while providing a clear mechanism for tracking progress and performance across all implementation phases.

2.5 Secondary code structure

In this section we will show the secondary code structure (2.5) of our work.

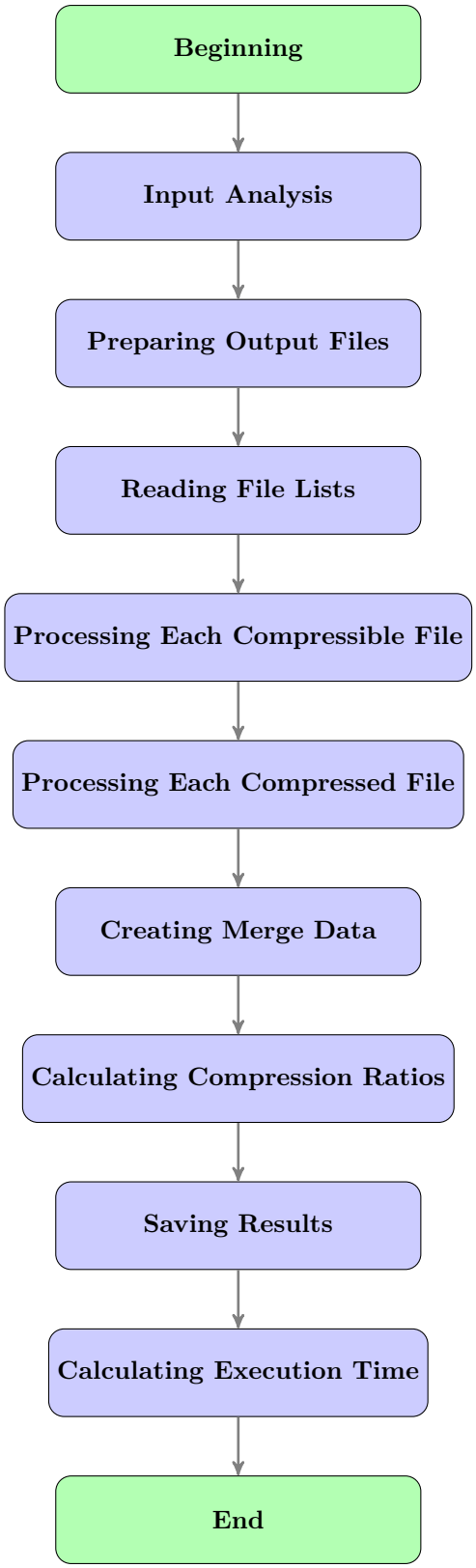

Figure 2.5: Secondary code structure

The subroutine acts as a specialized processing unit that follows a strict flow at the start of execution. It parses the inputs and then proceeds to prepare the output files by creating CSV files to store the bits, message files (msgfiles) for text logs, and collection files (combFilesList) to document the merge results.

In the file list reading phase, the program browses the contents of the specified directories (dataSet_path, compDataSet_path) via os.listdir(), extracting a subset according to the specified range. Two nested loops then run to process each compressible file and each compressed file. A merged data (ibuf) is created via the create_interleaved_buf() function, which merges the 512 bytes of the original file with the 256 bytes of the compressed file in an interleaved pattern.
Next, we apply the compression methods, which are gzip, LZ4, and zstd. Next comes the compression ratio calculation stage, which compares the combined compression size (len(compressed_data)) with the separate compression size (len(b1_compressed) + len(b2)) to determine efficiency. If the comparison is successful, the results are saved via the saveB1_B2() function, which formats the data in CSV format, while statistics are collected via saveFilesList().
Finally, the process concludes by calculating the execution time (end = time.time()) and printing a detailed report showing the total duration and average performance for each stage.

2.6 File merging and compression structure

In this section, we will mention the steps we took during merging and compressing (2.6) files and the results of this process.

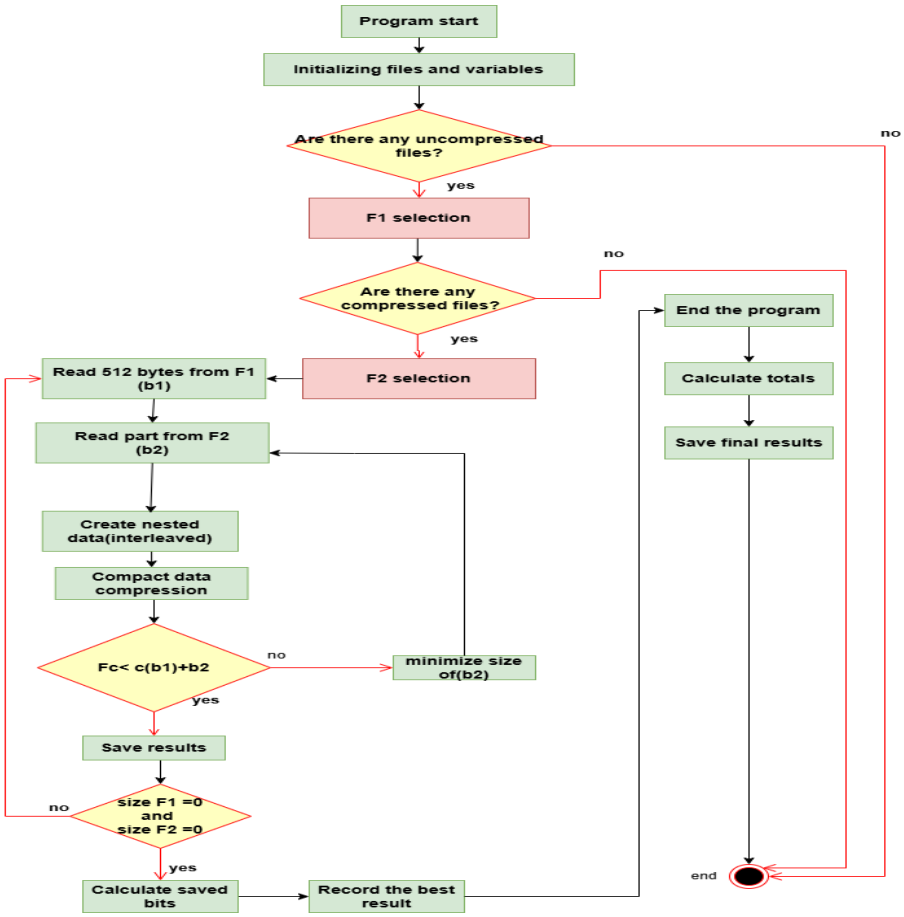

Figure 2.6: File merging and compression structure

We run the program and initialize the files and variables. Files are created to record the results, such as the combinations file, which contains data such as the uncompressed file name, the compressed file, the compression algorithm, and the compression rate. A CSV file is also created to save the results. Next, a condition is checked to check for uncompressed files. If true, the file F1 is selected, and then the following condition is checked to check for compressed files. If compressed files are present, the program reads 512 bytes from F1 (becoming b1) and takes a portion from F2 (becoming b2). It then inserts b1 and b2 using the create_interleaved_buf function, which creates an interleaved buffer by strategically mixing the data from the two files. It takes data blocks from the compressed file (comp_vec) and the uncompressed file (uncomp_vec) and distributes them in a pattern calculated based on the actual data size used (uncomp_size). The function calculates the step by dividing the length of the compressed file by the amount of data used, then places the bytes of the uncompressed file at specified positions, filling in the gaps with bytes from the compressed file. The data is then compressed using the gzip, Lz4, and zstd methods. We then check whether Fc $<$ c(b1) + b2 for each method. If the condition is met, the results are logged using the saveB1_B2 function, which acts as a logging system, converting the data blocks (b1 and b2) to a uniform CSV format while maintaining a constant line length by padding each line as needed. It begins by converting each element in b1 (which must be 512 bytes) and b2 (which must be 256 bytes) to a string, adding zeros to ensure the required length if the data is incomplete. The function writes the final result to a file, appending the value ub, which represents the number of bytes actually used. A counter ensures that the line length does not exceed 769 characters (512 for b1 + 256 for b2 + 1 for ub). After the function is complete, the following two conditions are verified (size F1 = 0 and size F2 = 0). If the condition is met, the saved bits are counted, the best result is recorded, and the program closes. If the condition is not met (Fc $<$ c(b1) + b2), the size of b2 is reduced, and the interleaving process is repeated. If the first two conditions are not met, the program closes.

2.7 merging structure

In this section we will mention the steps we took while merging the files (2.7).

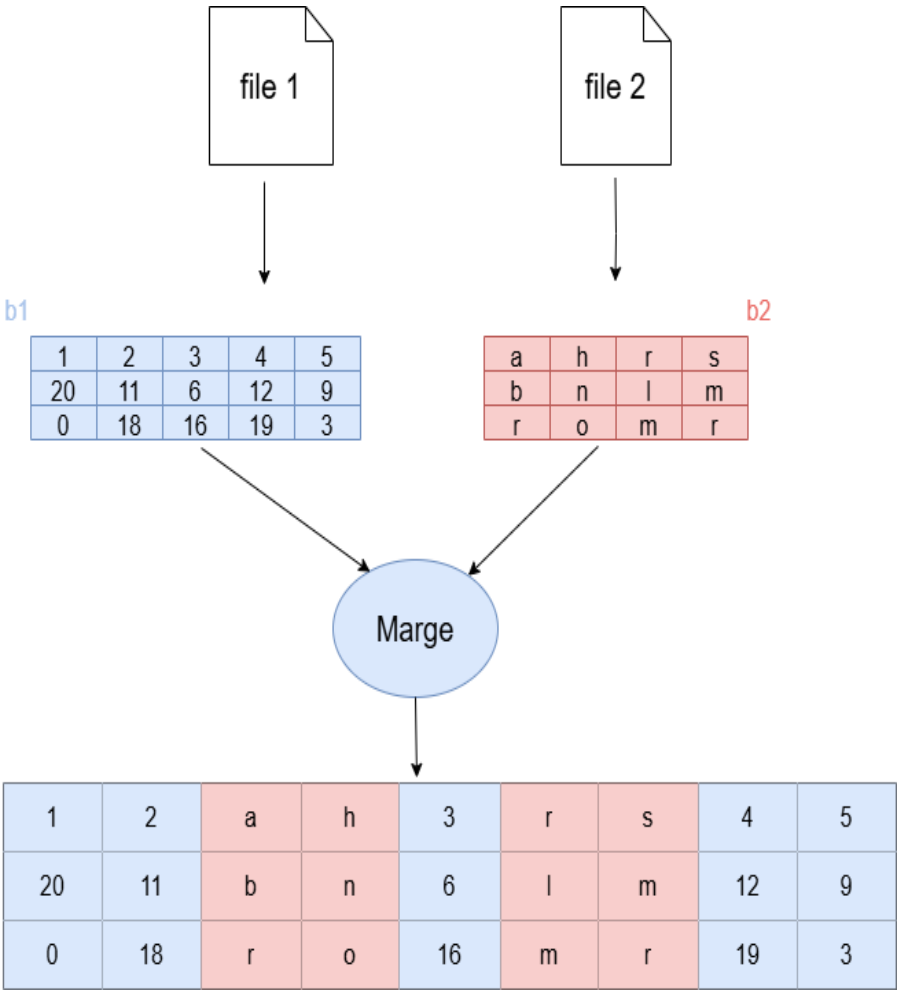

Figure 2.7: merging structure

The method shown in the figure represents the proposed merging method, where we take part b1 from file 1 and part b2 from file 2, then we merge the two parts together.

2.8 Model training

In this section we will mention the steps we took during training the model (2.8).

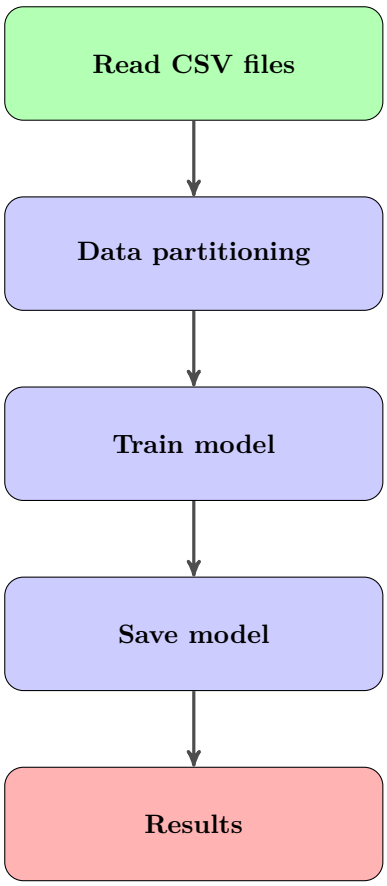

Figure 2.8: Model training structure

To train the model, we begin with the CSV file reading phase, where raw data is imported from the stored files. This is followed by the data splitting phase, where the dataset is divided into training and test sets at specific ratios, while maintaining a balance between the different classes in the data.

Next comes the model training phase, where machine learning algorithms such as SVM, CatBoost, and Random Forest are applied.

Finally, we save the trained model in special files for easy reuse later, while generating analytical results that demonstrate key performance metrics such as accuracy. This enables the user to comprehensively evaluate the model's performance and make informed decisions based on the results.

Conclusion

At the end of this chapter, we provide a comprehensive overview of the methodology used in this research, starting with the technical structure of the input files, proceeding through the successive merging and compression steps, and then providing a detailed explanation of each implementation step, ending with the use of the resulting data in training models. These steps form the practical foundation upon which the applied work in the final chapter is based, contributing to the construction of an effective compression method. This general framework paves the way for a discussion of the tools and algorithms used and a presentation of the experimental results obtained.

Chapter 3

Tests & Results

Introduction

This chapter focuses on the testing procedures carried out to evaluate the performance and reliability of the proposed system. Various test cases were designed to assess both functionality and efficiency under different conditions. The testing approach includes block level, and file level validation. Results obtained from these tests are presented and analysed in detail. Key performance indicators such as mean integrated bytes, mean gained bytes, mean wasted bytes are examined. Comparative analysis with existing benchmarks is also included.

3.1 Tools Used

In this section, we present the tools and software libraries that were used to implement and evaluate the proposed data compression system. Each tool played a specific role in the development process, from data analysis to machine learning model training and performance evaluation.

3.1.1 Python

Python is a high-level, interpreted, and open-source programming language known for its readability, simplicity, and versatility. It supports multiple programming paradigms, including object-oriented, procedural, and functional programming. Python has become widely popular in scientific computing, data analysis, artificial intelligence, and web development due to its rich ecosystem of libraries and active community support [1, 39].

3.1.2 Libraries Used

The following libraries and modules were utilized in the implementation and evaluation of the proposed data compression system:

1. `gzip`: A module for reading and writing GNU zip compressed files, used for compression and decompression operations [1].
2. `sys`: Provides access to some variables used or maintained by the Python interpreter, and to functions that interact with the interpreter. It is used for interacting with the system and managing input/output [1].

3. `os`: A library for interacting with the operating system, used for managing file paths and directories [1].

4. `numpy`: A fundamental package for scientific computing, providing support for large multi-dimensional arrays and matrices, along with a large collection of mathematical functions to operate on these arrays [26].

5. `math`: Provides mathematical functions such as trigonometric and logarithmic functions, which were useful for various calculations in the system [1].

6. `matplotlib.pyplot`: A popular library for plotting graphs and visualizations, useful for analyzing and presenting data in a graphical form [24].

7. `time`: Provides time-related functions. It was used for measuring the performance and execution time of compression algorithms [1].

8. `pickle`: A module used to serialize Python objects into byte streams and store them for later use. It is particularly helpful for saving trained machine learning models [1].

9. `pandas`: A data manipulation and analysis library. It was primarily used for handling data in a tabular form (DataFrames) and performing data wrangling tasks [24].

10. `sklearn`: A collection of machine learning algorithms and tools, such as `model_selection`, `LogisticRegression`, `svm`, and `accuracy_score`. It is used for building, training, and evaluating machine learning models [2].

11. `catboost`: A gradient boosting library used for machine learning tasks. The `CatBoostClassifier` was employed for classification tasks within the system [29].

12. `joblib`: A library for saving Python objects efficiently, often used to save machine learning models after training [1].

3.1.3 Models Used

The following models and algorithms were used in the implementation and evaluation of the proposed data compression system, with the aim of analyzing performance and predicting the effectiveness of different compression techniques based on the characteristics of the input data:

3.1.3.1 Support Vector Machine (SVM) Model

The Support Vector Machine (SVM) is a supervised learning algorithm widely used for classification and regression tasks. It works by finding the optimal hyperplane that best separates data points of different classes in a high-dimensional space. For non-linearly separable data, SVM can utilize kernel functions such as the radial basis function (RBF) to map the data into a higher-dimensional space where separation becomes feasible [10]. In this study, the SVM model was trained on features derived from block compression outcomes to classify optimal compression strategies or predict compression efficiency metrics.

3.1.3.2 Random Forest Model

Random Forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of their classifications or the mean prediction in regression cases. Its strength lies in reducing overfitting by averaging the outcomes of various uncorrelated trees, improving model robustness and accuracy [8]. In this work, Random Forest was employed to analyze the relationship between compression metrics and predict outcomes such as Mean Gained Bytes or Wasted Bytes.

3.1.3.3 CatBoost Model

CatBoost is a gradient boosting algorithm specifically optimized for categorical data and high-dimensional problems. It improves upon conventional gradient boosting by using ordered boosting and efficient handling of categorical features through permutation-driven techniques [28]. In this research, CatBoost was selected due to its ability to automatically handle categorical variables, minimize overfitting, and provide high predictive accuracy in scenarios with diverse file structures and compression behaviors. This section presents the evaluation results of the machine learning models applied in this study. Each model was trained and tested using the same dataset and evaluation protocol to ensure fair comparison. As shown in Table(3.8), the Random Forest model achieved the highest accuracy, followed by the Support Vector Machine (SVM) and CatBoost classifiers.

3.2 Dataset used

Under this title we will mention the groups that were used in the field of data compression, which are: The **Prague Corpus** (PDT) Table (3.1) is a richly annotated linguistic resource developed at **Charles University in Prague**. It represents one of the most significant corpora for syntactic and semantic analysis of the **Czech language**, based on the **Functional Generative Description (FGD)** theory.Over **180,000 sentences** from newspaper texts, translated literature, dialogues, etc [17, 25]. First released in **2001** and continuously updated. [17, 25]. Table (Table 3.1) contains information about Prague Corpus

File	Size[Bytes]	Description	Type
fireworks	1,440,054	Sound of fireworks	Audio
thunder	3,172,048	Sound of thunder	Audio
drkonqi	111,056	KDE crash handler	Binary
libc06	48,120	A dynamic-link library	Binary
mirror	90,968	A part of the software package	Binary
abbot	349,055	Part of interior design application	Binary
gtkprint	37,560	A shared object	Binary
wmvcrdt	328,550	A database file	Database

Continued on next page

File	Size[Bytes]	Description	Type
w01vlett	1,381,141	A database file	Database
emission	2,498,560	Waterbase emissions data	Database
bovary	2,202,291	Gustave Flaubert: *Madame Bovary*, in German	Documents
modern	388,909	Axel Lundegård, Ernst Ahlgren: *Modern. En berättelse*, in Swedish	Documents
ultima	1,073,079	Mack Reynolds: *Ultima Thule*, in English	Documents
lusiadas	625,664	Luís Vaz de Camões: *Os Lusíadas*, in Portuguese	Documents
venus	13,432,142	Ultraviolet image of Venus' clouds	Graphics
nightsht	14,751,763	A photo of a city at night	Graphics
flower	10,287,665	A photo of a flower	Graphics
corilis	1,262,483	CORILIS land cover data	Graphics
cyprus	555,986	Air Quality Monitoring in Cyprus	Markup
hungary	3,705,107	Air Quality Monitoring in Hungary	Markup
compress	111,646	Wikipedia page about data compression	Markup
lzfdmnt	22,922	C source code from a file archiver	Scripts
render	15,984	C++ source code from an action game	Scripts
handler	11,873	Java source code from the GPS tracking system	Scripts
usstate	8,251	Java source code from the GPS tracking system	Scripts
collapse	2,871	JavaScript source code from the project management framework	Scripts
xmlevent	7,542	PHP source code from the calendar generator	Scripts

Continued on next page

File	Size[Bytes]	Description	Type
mailfilder	43,732	Python source code from the ECM framework	Scripts
age	137,216	Age structure in the world	Spreadsheets
higrowth	129,536	Financial calculations	Spreadsheets
Total	**58,233,774**		

Table 3.1: The Prague Corpus files [30]

3.3 Results & Discussion

A series of tables is presented in the following section, summarizing the results obtained from applying the compression algorithms gzip, lz4, and zstd to two datasets: the Silesia Corpus and the Prague Corpus. This section presents an in-depth analysis of the compression performance using the **Prague Corpus** dataset. The evaluation is structured on two levels:

- **Block-Level Analysis:**
  - Mean Integrated Bytes by Block: Represents the average number of bytes combined when merging each pair of data blocks.
  - Mean Gained Bytes by Block:Refers to the average number of bytes effectively gained as a result of merging two data blocks.
- **File-Level Analysis:**
  - Minimum Wasted Bytes:Indicates the lowest number of bytes discarded when searching for the most suitable file candidate for block merging. This metric highlights the efficiency of inter-file merging decisions.

3.3.1 Gzip

Table(3.2) represents the Mean Integrated Bytes and Mean Gained Bytes for each block pair using the Gzip algorithm.

Block Pair	Mean Integrated Bytes	Mean Gained Bytes	Num Blocks
(xmlevent, collapse)	83	2	15
(usstate, collapse)	74	2	17
(handler, collapse)	66	2	19
(render, collapse)	70	3	18
(lzfindmt, collapse)	63	2	20
(abbot, age)	55	0	6
(gtkprint, collapse)	78	2	16
(mailflder, collapse)	70	2	18
(libc06, collapse)	12	0	94
(render, collapse)	8	0	190
(handler, collapse)	4	0	16
(modern, collapse)	46	0	760
(nightsht, collapse)	238	2	178
(flower, collapse)	153	2	2053
(thunder, collapse)	29	0	20
(firewrks, collapse)	255	1	760
(lusiadas, collapse)	5	1	20
(modern, collapse)	4	0	20

Table 3.2: Block-level results for Gzip

As shown in Table (3.2), the Gzip algorithm demonstrates excellent performance in terms of Mean Gained Bytes across most block pairs. The maximum recorded value does not exceed 3 bytes, observed only in the (render, collapse) pair. In contrast, several pairs such as (abbot, age), (libc06, collapse), and (modern, collapse) reported zero gained bytes, highlighting Gzip's stability and efficiency during block merging operations without causing noticeable size inflation.
Regarding the Mean Integrated Bytes, values varied significantly depending on the nature of the block pairs. The highest average was 255 bytes in the (firewrks, collapse) pair and 153 bytes in (flower, collapse), while the corresponding Mean Gained Bytes remained low at 1 and 2 bytes respectively. This indicates that Gzip effectively handles large or repetitive data blocks without introducing significant overhead.
It is also noticeable that pairs containing a large number of blocks — such as (modern, collapse) with 760 blocks and (flower, collapse) with 2053 blocks — achieved similarly efficient performance, confirming Gzip's capability to maintain compression consistency even when dealing with massive or highly repetitive blocks.

Table(3.3) represents the Minimum Wasted Bytes for each file after compression using the Gzip algorithm.

File	Minimum Wasted Bytes
abbot	496
xmlevent	1458
usstate	1655
handler	1831
render	1178
libc06	1485
lzfindmt	1775
gtkprint	1101
mailflder	1738
modern	1856
nightsht	2610
flower	378

Table 3.3: File-level results for Gzip

As shown in Table(3.3), the *flower* file exhibited the lowest Minimum Wasted Bytes value of 378, indicating a favorable balance between the original file size and the compression efficiency achieved by Gzip. In contrast, files such as *nightsht*, *modern*, and *handler* reported significantly higher wasted bytes, exceeding 1800 bytes in some cases.
This pattern suggests that Gzip's compression efficiency varies depending on the structure and content of the file. Files like *flower*, which likely contain more repetitive or compressible patterns, benefit more from Gzip's algorithm, while files with less redundancy tend to result in higher wasted bytes. Overall, these results highlight the importance of file content characteristics in determining the effectiveness of block-based compression operations.

3.3.2 LZ4

This table (3.4) presents LZ4's performance at the block level, showing the Mean Integrated Bytes and Mean Gained Bytes and Number of Blocks for each block pair.

Block Pair	Mean Integrated Bytes	Mean Gained Bytes	Num Blocks
(xmlevent, collapse)	83	3	15
(usstate, collapse)	73	5	17
(handler, collapse)	60	3	21
(render, collapse)	74	4	17
(lzfindmt, collapse)	74	4	17
(abbot, age)	55	0	6
(gtkprint, collapse)	66	2	19
(mailflder, collapse)	84	3	15
(libc06, collapse)	13	1	19
(render, collapse)	7	1	19
(handler, collapse)	4	0	21
(modern, collapse)	49	0	19
(nightsht, collapse)	255	1	17
(flower, collapse)	216	3	19
(thunder, collapse)	29	0	19
(firewrks, collapse)	255	1	19
(lusiadas, collapse)	5	1	19
(modern, collapse)	4	0	19

Table 3.4: Block-level results for LZ4

As shown in Table (3.4), the LZ4 algorithm exhibits a generally higher compression gain at the block level compared to Gzip. The Mean Gained Bytes values are slightly higher and more varied, indicating that LZ4 benefits more from block merging in certain cases.
The highest observed value was 5 bytes in the *(usstate, collapse)* pair — highlighted in yellow — which suggests that this specific pair contained a notable amount of redundant or compressible data across blocks. Additionally, other pairs such as (render, collapse) and (lzfindmt, collapse) recorded consistent gains of 4 bytes, confirming LZ4's effectiveness in identifying and eliminating inter-block redundancies.
Notably, pairs like (xmlevent, collapse), (handler, collapse), and (mailflder, collapse) achieved respectable gains of 3 bytes, supporting the algorithm's ability to detect structural similarities across a variety of block types.
In contrast, several block pairs — including (abbot, age), (modern, collapse), and (thunder, collapse) — recorded zero gained bytes, indicating that merging blocks in these cases had little to no impact on the compressed size. This might be due to limited redundancy or already optimized data structures within these specific blocks.
Regarding the Mean Integrated Bytes, the values remain relatively consistent, with certain pairs like (firewrks, collapse) and (nightsht, collapse) reaching a high of 255 bytes, showing that while the amount of integrated data can be large, LZ4 keeps the additional size (Mean Gained Bytes) relatively low, ensuring efficient compression.

This tablel (3.5) shows the Minimum Wasted Bytes after compressing each file using LZ4.

File	Minimum Wasted Bytes
abbot	253
xmlevent	1057
usstate	1341
handler	1558
render	734
libc06	976
lzfindmt	1250
gtkprint	993
mailflder	1034
modern	1387
nightsht	1290
flower	1486

Table 3.5: File-level results for LZ4

Most files show wasted bytes below 1500, with *abbot* being the best performer.
As illustrated in Table(3.5), the file-level evaluation reveals that most files had **Minimum Wasted Bytes** under 1500, indicating that LZ4 efficiently matched files for block merging. The file *abbot* achieved the lowest value at 253 bytes, suggesting that it was the most compatible with the merging mechanism and had minimal redundancy overhead. Conversely, files such as

3.3.3 ZSTD

Table(3.6) represents ZSTD's performance at the block level, showing the Mean Integrated Bytes and Mean Gained Bytes for each block pair.

Block Pair	Mean Integrated Bytes	Mean Gained Bytes	Num Blocks
(xmlevent, collapse)	83	2	15
(usstate, collapse)	73	4	17
(handler, collapse)	52	5	16
(render, collapse)	52	1	18
(lzfindmt, collapse)	63	3	20
(abbot, age)	55	0	18
(gtkprint, collapse)	78	3	19
(mailflder, collapse)	66	3	18
(libc06, collapse)	11	2	94
(render, collapse)	7	1	190
(handler, collapse)	4	0	21
(modern, collapse)	47	0	19
(nightsht, collapse)	255	1	17
(flower, collapse)	216	3	19
(thunder, collapse)	28	0	19
(firewrks, collapse)	255	1	19
(lusiadas, collapse)	5	1	19
(modern, collapse)	4	0	19

Table 3.6: Block-level results for ZSTD

As shown in Table **(3.6)**, the ZSTD algorithm provides a balanced performance at the block level, with a noticeable variation in the **Mean Gained Bytes** values.
The highest gain of 5 bytes appears in the pair *(handler, collapse)*, indicating strong compression synergy between these blocks. Other pairs, such as *(usstate, collapse)* and *(gtkprint, collapse)*, also achieve respectable gains ranging from 3 to 4 bytes, demonstrating ZSTD's good ability to detect redundancy or structural similarity between these blocks.
On the other hand, some block pairs like *(abbot, age)* and *(modern, collapse)* show zero gain, suggesting limited redundancy or structural similarity that can be exploited in these cases.
These results highlight that ZSTD performs particularly well with larger or more structured blocks, offering moderate but consistent compression gains across different data types. This makes it a suitable choice for applications requiring a balance between compression quality and final file size while benefiting from the structural properties of the data.

Table(3.7) represents the Minimum Wasted Bytes for each file after compression using ZSTD.

File	Minimum Wasted Bytes
abbot	240
xmlevent	1175
usstate	1366
handler	1624
render	889
libc06	1414
lzfindmt	1601
gtkprint	748
mailflder	1333
modern	1455
nightsht	1290
flower	1486

Table 3.7: File-level results for ZSTD

Table(3.7) shows that ZSTD generally maintains acceptable efficiency at the file level, with **Minimum Wasted Bytes** staying under 1700 across all tested files. The best result is observed in the *abbot* file, with only 240 wasted bytes—indicating excellent compatibility between the file structure and the ZSTD compression model. However, larger files like *handler*, *lzfindmt*, and *flower* exhibit higher wasted byte values, ranging from 1400 to 1600. This suggests that, although ZSTD can effectively compress diverse file types, its performance may vary depending on file complexity and internal redundancy. Notably, while ZSTD outperformed Gzip in some instances, it still resulted in relatively high overhead in structurally complex files such as *usstate*.

3.3.4 Comparison Between Algorithms

(Figure 3.1) summarizes the performance of the three algorithms in terms of average integrated bytes , average gained bytes (best case), and minimum wasted bytes.

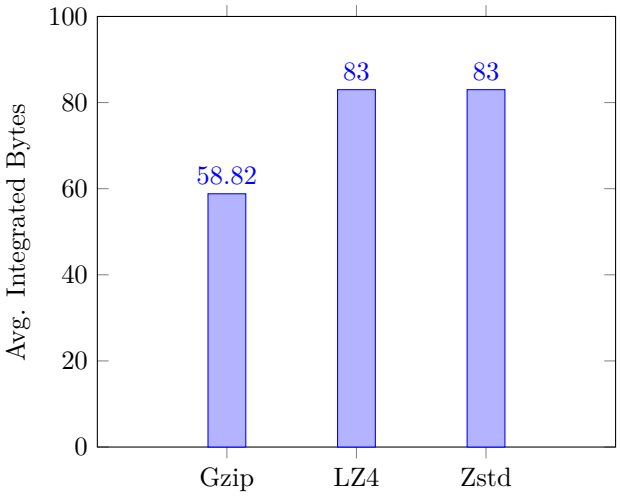

Figure 3.1: Mean Integrated Bytes by Block

As illustrated in **Figure(3.1)**, both LZ4 and ZSTD achieve the highest mean integrated bytes per block (83 bytes), significantly outperforming Gzip, which averages only 58.82 bytes. This suggests that LZ4 and ZSTD are more effective at integrating larger data segments during compression, likely due to their more advanced block processing mechanisms.

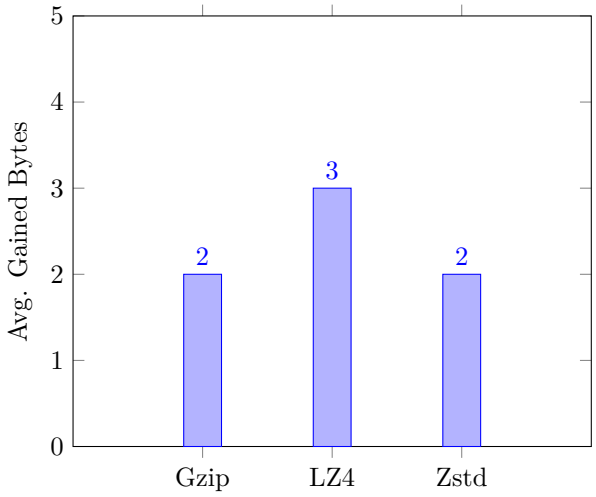

Figure 3.2: Mean Gained Bytes (best case)

Figure(3.2) shows that LZ4 leads in terms of average gained bytes with a value of 3, while Gzip and ZSTD are tied at 2. Gained bytes reflect the efficiency improvements from merging block pairs. Thus, LZ4 demonstrates greater adaptability in capitalizing on structural redundancies between blocks.

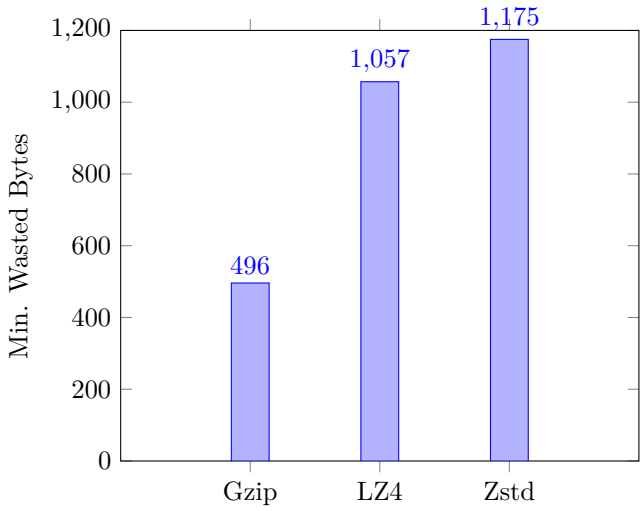

Figure 3.3: Min. Wasted Bytes

According to **Figure(3.3)**, Gzip clearly outperforms the other algorithms in terms of minimizing wasted bytes, with only 496 bytes compared to 1057 in LZ4 and 1175 in ZSTD. This metric indicates the overhead resulting from mismatches in block merging; hence, Gzip shows better alignment between file structure and compression model in its best case.

Overall Comparison: Each algorithm excels in different aspects. LZ4 and ZSTD demonstrate superior integration capabilities at the block level **Figure(3.1)**, while LZ4 also gains the most in efficiency **Figure(3.2)**. However, when it comes to minimizing redundancy and wasted space in file-level compression, Gzip performs best **Figure(3.3)**. This indicates that algorithm selection should depend on whether the priority is block-level integration or file-level compactness.

3.3.5 Treaning Modeles Results

In this study, machine learning models were trained with the objective of predicting the number of **used bytes** after the compression process. A set of features derived from compression operations was utilized to assess their effectiveness as reliable predictive indicators for estimating the actual number of bytes utilized following compression. This predictive capability is essential for enhancing the evaluation of compression algorithm efficiency and supporting decision-making in systems where continuous storage optimization is critical.

To achieve this, several machine learning models were employed, including *Support Vector Machine (SVM)*, *Random Forest*, and *CatBoost*. The dataset was divided into two subsets: **80%** was allocated for training the models, while the remaining **20%** was reserved for testing their ability to predict the used bytes. The classification accuracy results for each of the models are presented in the following table(3.8):

In this study, machine learning models were trained with the objective of predicting the number of used bytes after the compression process. A set of features derived from compression operations was utilized to assess their effectiveness as reliable predictive indicators for estimating the actual number of bytes utilized following compression. This predictive capability is essential for enhancing the evaluation of compression algorithm efficiency and supporting decision-making in systems where continuous storage optimization is critical.

To achieve this, several machine learning models were employed, including *Support Vector Machine (SVM)*, *Random Forest*, and *CatBoost*. The dataset was divided into two subsets: **80%** was allocated

for training the models, while the remaining **20%** was reserved for testing their ability to predict the used bytes. Specifically, the training set contained **47,724 samples** , while the testing set included **11,932 samples**. The classification accuracy results for each of the models are presented in the following table(3.8):

Model	**Accuracy**
Support Vector Machine (SVM)	91.64%
Random Forest	94.03%
CatBoost	61.54%

Table 3.8: Classification accuracy of different machine learning models

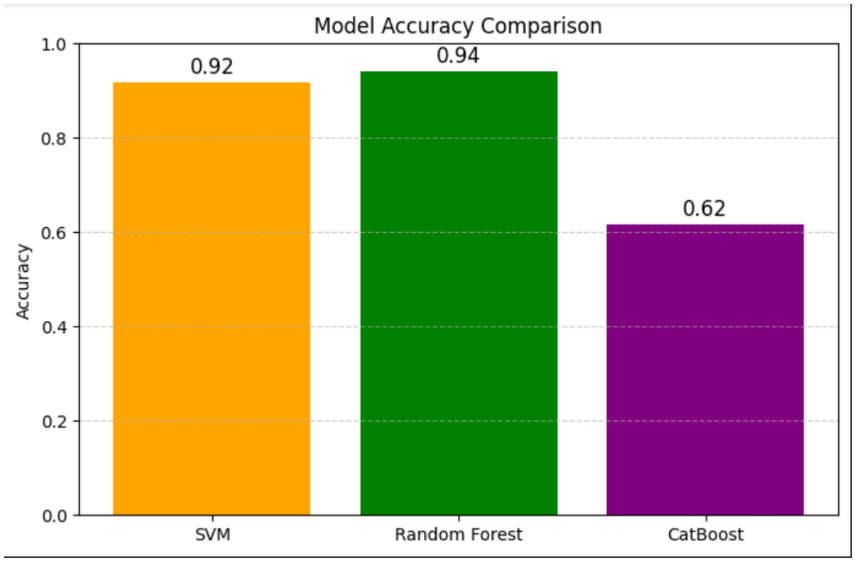

Figure 3.4: Model accuracy comparison between SVM, Random Forest, and CatBoost

As shown in Figure (3.4), the Random Forest model achieved the highest accuracy in predicting the *used bytes* value, reaching **94%** in predicting the values resulting from the compression process. This result highlights the superior performance of ensemble-based methods in capturing complex patterns within the dataset. The Support Vector Machine (SVM) model came in second with an accuracy of **92%**, demonstrating its solid capability in prediction tasks, particularly when dealing with high-dimensional feature spaces. In contrast, the CatBoost model recorded a lower accuracy of**62%** , which may indicate sensitivity to the nature of the data or a need for further hyperparameter tuning.
These results demonstrate the effectiveness of both Random Forest and SVM in modeling the relationship between input features and the predicted compression values, with Random Forest maintaining a slight advantage. This suggests that ensemble-based methods, which combine the decisions of multiple decision trees, are better at capturing the intricate and nonlinear relationships present in compression-derived data compared to individual models like SVM or gradient boosting approaches such as CatBoost.

3.4 Improved method

Given the results we obtained from the method we used to merge the B2 data into B1 using a step calculation to determine where to place the B2 data, we found a loss across the files. We therefore implemented a new method, placing the B2 data before B1, and then placing the data together. We preprocessed the F2 file, from which we would extract B2, using a function that restructured the file data and added some details while preserving the original file data. We then evaluated the success of the proposed method by calculating the percentage of bits gained and the total size c(F1)+F2, and recording it. We then calculated the percentage of bits gained and the size of F2 after applying the second proposed method we mentioned earlier.

We applied the three compression methods: ppmd, lzma, and deflate, and recorded the results in the following tables:

Uncompressed File	Compressed File	Gained Bytes	Merger percentage (%)	percentage of original f2(%)
abbot	emission.7z	178.4	23.8	82.8
age	modern.7z	83.8	26.6	63.5
bovary	firewrks.7z	527.7	18.4	44.0
collapse	handler.7z	1.5	21.8	52.8
compress	hungary.7z	36.0	19.5	43.3

Table 3.9: File-level results for ppmd

Uncompressed File	Compressed File	Gained Bytes	Merger percentage (%)	percentage of original f2(%)
abbot	emission.7z	193759	26.0	89.9
age	modern.7z	93008	29.8	70.5
bovary	firewrks.7z	697154	23.5	58.2
collapse	handler.7z	1600	22.8	55.8
compress	hungary.7z	48611	26.3	58.5
corilis	firewrks.7z	950798	31.7	79.3
cyprus	emission.7z	154575	34.6	71.8
drkonqi	emission.7z	21949	8.3	73.4

Table 3.10: File-level results for LZMA

Uncompressed File	Compressed File	Gained Bytes	Merger percentage (%)	percentage of original f2(%)
abbot	abbot.7z	296.3	31.4	94.3
age	modern.7z	90.5	28.1	68.6
bovary	drkonqi.7z	21.6	2.9	72.1
collapse	handler.7z	1.5	21.3	51.6
compress	hungary.7z	36.3	19.5	43.7
corilis	firewrks.7z	446.0	14.5	37.2
cyprus	emission.7z	184.1	24.6	85.5
drkonqi	emission.7z	81.8	7.4	38.0

Table 3.11: File-level results for Deflate

3.4.1 Execution time comparison of ppmd, lzma and deflate algorithms

Algorithm	Time (ms)	Time (sec)	Time (mins)	Time (hours)
ppmd	216017.62	216.017	3.600	0.060
lzma	200597.12	200.597	3.343	0.055
deflate	165739.51	165.739	2.762	0.046

Table 3.12: Execution Time Comparison of Compression Algorithms

Conclusion

This chapter provided a comprehensive evaluation of the proposed data compression system through a series of structured tests conducted at both the block and file levels. The testing methodology was designed to assess the effectiveness of different compression algorithms—namely Gzip and LZ4—by analyzing key performance indicators such as Mean Integrated Bytes, Mean Gained Bytes, and Minimum Wasted Bytes across various file types in the Prague Corpus dataset.

The results indicate that while Gzip offers consistent compression behavior with minimal data inflation during block merging, LZ4 demonstrates greater adaptability in identifying and exploiting structural redundancies, yielding higher compression gains in specific scenarios. At the file level, performance varied depending on the file content and format, reinforcing the importance of data characteristics in determining compression efficiency.

Ultimately, the comparative analysis highlights the strengths and limitations of each algorithm within the context of block-aware compression strategies. These insights not only validate the functionality of the proposed system but also provide a foundation for further optimization and algorithmic enhancement in future work.

Conclusion & Perspectives

Data compression is a crucial technique used to reduce the size of data for more efficient storage and transmission. With the exponential growth of digital data, compression methods help mitigate the challenges of limited storage space and bandwidth constraints. These methods can be divided into lossless, where data is preserved exactly, and lossy, where some data is discarded for higher compression ratios. Compression algorithms identify patterns and redundancies in data, optimizing its representation. As data continues to expand across industries, the demand for more efficient compression solutions grows. Research in this area aims to balance compression efficiency with computational resources. Optimizing these techniques remains a key focus for improving digital communication and storage.

This thesis introduces a novel data compression method based on the concept of combining data blocks to improve compression efficiency. The proposed approach was motivated by the observation that traditional compression techniques often overlook higher-level structural redundancies that exist across adjacent or similar data segments. By identifying and merging such blocks, the method aimed to reduce overall data redundancy and improve compression ratios.

The design and implementation of the method demonstrated that combining blocks can lead to a notable gain in compression performance, especially in datasets where repetitive or structured content is prevalent. Experimental results confirmed that the approach achieves competitive compression rates compared to standard algorithms but also maintains reasonable computational complexity on block level, especially when the size is small. However, the same gain cannot be obtained in file level with the current implementation, which shows the need for applying preprocessing algorithms for uncompressible data smoothing.

Building upon the findings and outcomes of this work, several promising research directions can be explored to enhance the proposed compression approach and broaden its applicability:

1. **Adaptive Block Sizing and Partitioning**: Future work can investigate dynamic or content-aware block partitioning techniques. Rather than using fixed-size blocks, adaptive strategies based on data characteristics could improve similarity detection and overall compression performance.

2. **Optimization of Similarity Detection Algorithms**: The efficiency of the proposed method heavily relies on the ability to quickly and accurately identify similar blocks. Future research could focus on integrating faster, scalable similarity metrics—potentially leveraging hashing, clustering algorithms, or machine learning techniques.

Bibliography

[1] Python programming language – official website, 2024.

[2] Scikit-learn documentation, 2024.

[3] Adobe. Lossy vs. lossless compression. https://www.adobe.com/uk/creativecloud/photography/discover/lossy-vs-lossless.html, 2024. Accessed: 2025-04-16.

[4] B. Balle et al. End-to-end optimized image compression with neural networks. *IEEE Transactions on Image Processing*, 2018.

[5] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2017.

[6] Netflix Technology Blog. How netflix delivers high-quality streaming. 2020.

[7] B. Bodden et al. Compression for speech and text data using neural networks. *IEEE Transactions on Speech Processing*, 2020.

[8] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] X. Chen et al. Neural image compression. *IEEE Transactions on Image Processing*, 2020.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[11] DataScientest. Codage de huffman : tout savoir, 2023. Accessed: 2025-04-16.

[12] DataScientest. Algorithme lz77 : principes et mise en œuvre, 2024. Accessed: 2025-04-19.

[13] I. Goodfellow et al. Generative adversarial nets. In *Neural Information Processing Systems (NeurIPS)*, 2014.

[14] Google. Data center efficiency: How google does it. *Google Sustainability*, 2015.

[15] Google Inc. Brotli compression algorithm, 2015. https://github.com/google/brotli.

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

[17] Charles University in Prague. Prague dependency treebank (pdt). https://arxiv.org/abs/2006.03679, 2020. Accessed: 2025-04-14.

[18] ISO/IEC. Jpeg 2000 image coding system, 2004. ISO/IEC 15444-1.

[19] ITU-T and ISO/IEC. High efficiency video coding (hevc), 2013. ITU-T Rec. H.265 / ISO/IEC 23008-2.

[20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2002.

[21] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2014.

[22] A. Krikun et al. Convolutional networks for efficient image compression. *Journal of Signal Processing*, 2020.

[23] David Lee and Sarah Black. Neural image compression: A deep learning approach. *Journal of Image Processing*, 28(3):33–44, 2023.

[24] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, 2010.

[25] Institute of Formal and Charles University Applied Linguistics. Prague dependency treebank documentation (pdt 2.0). https://ufal.mff.cuni.cz/pdt2.0/doc/manuals/en/intro/index.html, 2006. Accessed: 2025-04-14.

[26] Travis E. Oliphant. *NumPy Cookbook*. O'Reilly Media, 2011.

[27] A. V. D. Oord et al. Neural audio compression using deep autoencoders. *IEEE Transactions on Audio, Speech, and Language Processing*, 2018.

[28] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

[29] Olga Prokhorenkova et al. *CatBoost: Gradient Boosting with Categorical Features Support*. arXiv preprint arXiv:1807.01668, 2018.

[30] Marc Reznicek. Corpus for comparing compression methods and an extension of a excom library. Master's thesis, Charles University in Prague, 2010. Focuses on corpus-based comparison of compression methods and the extension of the ExCom library.

[31] O. Rippel et al. Recurrent neural networks for video compression. *International Journal of Computer Vision*, 2017.

[32] David Salomon. *Data Compression: The Complete Reference*. Springer, 2007.

[33] Khalid Sayood. *Introduction to Data Compression*. Elsevier, 5th edition, 2017.

[34] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 5th edition, 2017.

[35] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[36] James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, 1988.

[37] I. Sutskever et al. Sequence to sequence learning with neural networks. In *NeurIPS*, 2014.

[38] TechTarget. What is data compression?, 2023. Accessed: 2023-11-15.

[39] Guido van Rossum and Fred L Drake. *Python 3 Reference Manual*. CreateSpace, 2009.

[40] V. Vasilev et al. Learning to compress with deep networks. *Journal of Machine Learning*, 2020.

[41] A. Vaswani et al. Attention is all you need. In *NeurIPS*, 2017.

[42] W3Schools. Huffman coding - data structures and algorithms reference, 2023. Accessed: 2025-04-16.

[43] W3Schools. Lz77 compression algorithm, 2023. Accessed: 2023-04-19.

[44] W3Schools. Lzma compression algorithm, 2023. Accessed: 2025-04-19.

[45] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1992.

[46] Wikipedia contributors. Mp3, 2025. Accessed: 2025-04-19.

[47] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[48] WsCube Tech. Types of data compression. https://www.wscubetech.com/resources/dsa/compression-algorithms, 2020. Accessed: 2025-04-16.

[49] Y. Zhu et al. Lossy compression with neural networks. *IEEE Transactions on Signal Processing*, 2021.