People's Democratic Republic Of Algeria

Ministry of Higher Education and Scientific Research

University ECHAHID HAMMA LAKHDAR - EL OUED

Fculty OF Exact Sciences

Computer Science department

End of study memory

Presented for the Diploma of

ACADEMIC MASTER

Domain: **Mathematics and Computer Science**

Industry: **Computer Science**

Specialty: **Distributed Systems and Artificial Intelligence**

Theme

Development of a data compression method by substitutions

Presented by:

Hamied Abd Djalal

Debbar Abderrahmane

DR,	Faouzi-Zaiz	Supervisor Univ. El Oued
DR.	Benbarika mohamedkamel	president Univ. El Oued
DR.	Mohieddin Khabbache	Examiner Univ. El Oued

Academic year

2023/2024

Dedication

First of all we dedicate this work to our parents which they support and tired for us for developing our selfes in this road of education, also without forgetting all teacher we have meeting them before ,specially in this year of university and all friends ..... thanks all.

Acknowledgements

We would like to express our heartfelt gratitude to everyone who stood by us and supported us throughout the course of this project. Your efforts were not just words of encouragement, but a driving force behind every achievement we made. We also want to extend our sincere thanks to our families and friends for their unwavering support and understanding during our challenging times. Thanks to you, we were able to successfully achieve our goals and overcome obstacles with confidence and faith

Abstract

The volume of data transmitted over the Internet is escalating rapidly. With network bandwidth remaining a persistent limitation, the imperative lies in leveraging efficient compression algorithms to streamline the swift and effective exchange of data across networks. This study undertakes a comparative analysis of various lossless compression techniques. Moreover, it introduces a novel file pre-processing methodology designed to enhance compressibility. The research also incorporates a block-switching process, determined by calculating the minimum distance between blocks. Additionally, a predictive model is developed to ascertain the necessity of the switching process for each block.The performance evaluation encompasses several widely-used compression algorithms, including Deflate, Deflate64, Bzip2, LZMA, and PPMd, with an in-depth examination conducted on the Prague Corpus dataset.

Keywords: Compression,Prague Corpus, lossless compression, Silesia, compression ratio, compression speed, permutation, predictive model.

Résumé

Le volume de données transmises sur Internet augmente rapidement. Avec une bande passante réseau qui reste une limitation persistante, il est impératif de tirer parti d'algorithmes de compression efficaces pour faciliter l'échange rapide et efficace de données à travers les réseaux.

Cette étude entreprend une analyse comparative de diverses techniques de compression sans perte. De plus, elle introduit une nouvelle méthodologie de pré-traitement des fichiers conçue pour améliorer la compressibilité. La recherche intègre également un processus de commutation de blocs, déterminé par le calcul de la distance minimale entre les blocs. En outre, un modèle prédictif est développé pour déterminer la nécessité du processus de commutation pour chaque bloc. L'évaluation des performances englobe plusieurs algorithmes de compression largement utilisés, y compris Deflate, Deflate64, Bzip2, LZMA et PPMd, avec un examen approfondi effectué sur l'ensemble de données Prague Corpus.

Mots-clés : Compression, Prague Corpus, compression sans perte, Silesia, taux de compression, vitesse de compression, permutation, modèle prédictif.

ملخص

تتزايد حجم البيانات المرسلة عبر الإنترنت بسرعة. مع استمرار قيود عرض النطاق الترددي الشبكي، فإن الضرورة تكمن في استغلال خوارزميات الضغط الفعالة لتبسيط التبادل السريع والفعال للبيانات عبر الشبكات. يقوم هذا البحث بتحليل مقارن لمختلف تقنيات الضغط اللاخساري. علاوة على ذلك، يقدم منهجية مبتكرة لمعالجة الملفات مصممة لتعزيز قابلية الضغط. كما يشمل البحث عملية تبديل الكتل، التي تتم بحساب المسافة الدنيا بين الكتل. بالإضافة إلى ذلك، يتم تطوير نموذج تنبؤي لتحديد ضرورة عملية التبديل لكل كتلة. تشمل التقييم الأداء العديد من خوارزميات الضغط المستخدمة على نطاق واسع، بما في ذلك Deflate, Deflate64, Bzip2, LZMA, PPMd **مع إجراء فحص عميق على مجموعة البيانات** Prague Corpus.

Contents

List of Figures

List of Tables

Introduction General

Since the early days of computing, we've faced the limitation of computers' capacity to store and transmit information. Recognizing that most of the data we handle isn't random, we've developed data compression methods. These methods aim to convert data into a more compact form (compression), even if it becomes unreadable, and then restore it to its original form later (decompression).

By using a more compact representation, we can save storage space or bandwidth during transmission through various communication means. Despite constant progress in computer speed and storage capacity, the clutter associated with the amount of information we handle doesn't decrease, as we're constantly dealing with increasingly larger volumes of data. Thus, the utility of data compression methods remains as relevant as ever. Various approaches to data compression can be observed, some being general-purpose while others are specific to particular data types, such as images, for example. There are methods that achieve compression without alteration, while others involve compression with a loss of quality.

Initial methods aim to restore the original data without any alteration after undergoing compression and decompression steps. More recent approaches have the ability to modify data after compression and decompression, provided that the reconstructed data remains relatively similar to the original data. For example, in many cases, audio or video documents can undergo alterations as long as they remain within certain tolerable limits.

Data compression methods vary in terms of compression ratio and processing speed.

In this work, we will present a comparative study of the most well-known lossless compression techniques. Additionally, we propose a method for preprocessing files to improve the compression ratio. The method involves rearranging different data segments of the target file to make the segments closer (in terms of distance) in adjacent positions.

The first chapter provides a state-of-the-art overview of different data compression techniques.

The second chapter presents lossless data compression methods in much more detail.

The third chapter will focus on the various modules and processing stages of the proposed system.

The fourth chapter presents the validation of results. Within this, we'll see how our system functions overall. Among these steps, we present the preprocessing algorithm. Additionally, we present the obtained results and discussions.

We conclude the paper with a summary of the results obtained from the methods used and the prospects of this work.

Chapter 1

Introduction to Data Compression

Introduction

Data compression encompasses all methods and principles that allow reducing the size of data without losing essential information. In simple terms, it's a technique that employs a pair of functions, one for compressing data and the other for decompressing it.

The objective of compression is to represent information more concisely than the original, so that the compressed result occupies less space than the original data. Data can be compressed with or without loss of information.[1]

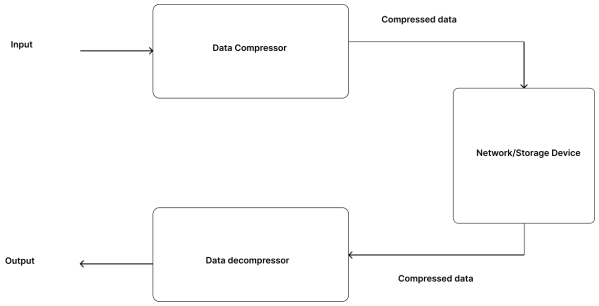


Figure 1.1: details screen image

1.1 Compression Overview

Data compression is a crucial technique in the modern digital world. It allows for the efficient storage and transmission of data by reducing file sizes. Compression can be achieved through various algorithms and methods, each with its own advantages and trade-offs. Understanding the differences between lossless and lossy compression helps in choosing the right method for different applications. While lossless compression ensures data integrity and is used where exact data reproduction is critical, lossy compression is more suitable for media files where some loss of quality is acceptable in exchange for smaller file sizes.

By utilizing these compression techniques, we can optimize storage space and improve data transmission efficiency, making it possible to handle larger amounts of data more effectively.

1.2 Types of compression

Essentially, there are two types of techniques available for compression: with loss and without loss. Each technique shares the same goal of reducing the file size.

1.2.1 Lossless compression

In this type of compression, each bit undergoes compression and is restored with total precision during the decompression stage. A set of bits X is compressed to obtain a shorter compressed version Y, which is then stored or transmitted. During the decompression of Y, the original set of bits X will be exactly recovered[1] .

1.2.2 Lossy compression

This type of compression plays a distinctly different role compared to lossless compression. The principle of lossy compression involves removing certain data (deemed non-essential) from the information to achieve more efficient compression. This means that during the decompression of the compressed file, an approximation of the original file will be obtained. Starting with information X that we wish to compress, the compression result is Y. After decompressing Y, we obtain a result Z, which is an approximation of X. This compression method is often used for compressing data such as images, sound, video, etc., as for this type of data, some data can be removed without affecting the essential information contained in the file[2].

1.3 Information theory

Information theory gained significance with the development of its mathematical theory by Shannon and Weaver, defining concepts such as information quantity, entropy, and redundancy[3].

1.3.1 quantity of information

The quantity of information can be mathematically defined through probability theory. The amount of information contained in a data stream is directly related to its probability of occurrence.

Let S be an information source (alphabet) $S = (X_1, X_2, X_3, ..., X_k)$ of k symbols. Let $P(X_i)$ be the probability of occurrence of X_i. The quantity of information (I_i) contained in a data stream is a function of the inverse of the probability of occurrence of this symbol, given by the following relationship:

$$I_i = \log_2\left(\frac{1}{P(X_i)}\right) = -\log_2(P(X_i))$$

1.3.2 Entropy

Generally, entropy is a measure of the degree of disorder in a given system. In information theory, entropy is used to quantify the randomness in a data stream or file. A file with low entropy, where all elements are entirely predictable in advance, will be more compressible. For example, a file filled with zeros has zero information content, lacks randomness, and has minimal entropy.

On the other hand, a compressed (already compacted) file contains bytes that appear random and have no relationship to each other. The amount of information therein is maximal, almost by definition, as it is a compressed file where all redundancies have been eliminated. Moreover, it contains a maximal dose of randomness, i.e., maximal entropy.

The entropy H of a file is the average information contained per symbol, given by the relationship:

$$H = \sum_i P(x_i)\log_2(P(x_i)) = -\sum_i P(x_i)\log_2(P(x_i))$$

Properties:

$$0 \leq H \leq \log_2(n)$$

$$H = 0 \text{ when } P(x_i) = 1$$

$$H = \log_2(n) \text{ when } P(x_i) = \frac{1}{n}, \quad i = 1..n$$

1.3.3 Redundancy

Data compression relies on identifying and eliminating redundancies present in a data stream. Since computer information, regardless of its form (text, sound, image, etc.), is logically represented by a combination of 256 different bytes (if we consider the byte as the basic unit), called the alphabet, this means that any data stream whose size exceeds that of the alphabet (the number of symbols) necessarily contains redundancies.

Example: Let A be the alphabet and F be the data stream. $A = \{0, 1\}$ of size 2 symbols. $F = 1001...10$ The data stream F exhibits redundancies between the symbols 0 and 1 as soon as its size exceeds that of the alphabet, which is 2.

Thus, we can conclude that large data streams (several megabytes) are merely the result of redundancy among the symbols of the alphabet, which is a limited set.

1.4 Main Compression Techniques

In today's digital world, where the creation, storage, and sharing of data are ubiquitous, data compression techniques play a central role. Data compression is the art of reducing the size of files or data without significant loss of information. This discipline is essential for optimizing the use of storage space, accelerating data transmission over networks, and saving valuable resources.

1.4.1 Encoding

Encoding is the process of converting information from one format to another, often used in the context of computer programming to create software and applications. Here are some of the most commonly used encoding techniques[4]:

Binary Encoding: Binary encoding uses only two symbols (0 and 1) to represent information.

ASCII Encoding: The American Standard Code for Information Interchange (ASCII) is a coding method used to represent alphanumeric characters.

Encryption Encoding: Encryption algorithms are used to secure data by transforming it into an unreadable form without the corresponding decryption key.

Compression Encoding: Compression algorithms are used to reduce the size of data.

1.4.2 Encoding Techniques and Algorithms

Data compression is frequently referred to as "**encoding**" due to its fundamental goal of finding a concise way to represent information. The terms "**encoding**" and "**decoding**" are used to denote compression and decompression respectively. There are numerous compression algorithms that transform data to make it more concise. Some of the most significant ones include:

- **Shannon-Fano Coding**: This technique is one of the earliest attempts to create efficient codes for data compression.
- **Huffman Coding**: It assigns variable-length codes to symbols based on their frequency of occurrence, allowing efficient compression of textual data.
- **Arithmetic Coding**: Arithmetic coding associates intervals with each symbol based

on its probability of occurrence, thus enabling an efficient representation of symbol sequences.

- **Run-Length Encoding (RLE)**: This technique replaces repeated sequences of symbols with a single occurrence of the symbol followed by the number of repetitions. It is effective for compressing data with repetitive sequences.
- **Move-To-Front (MTF) Coding**: The Move-to-Front (MTF) algorithm is a data compression technique that aims to exploit local redundancy in a character sequence.
- **Lempel-Ziv (LZ) Coding**: LZ family compression algorithms, such as LZ77 and LZ78, identify and replace repeated data sequences with references to previously encountered data, reducing the size of the data.

1.4.3 Compression Measures

The interest of compression is to minimize the size of original information without altering its content. To this end, several criteria are adopted to measure the performance of a compression method:

- **Compression Ratio**
- **Compression Gain**
- **Compression Savings Percentage**

Compression Ratio: The compression ratio is widely used to measure the quality of a compression algorithm. It is defined as the ratio of the size of the compressed file to the size of the original file and expressed as a percentage.

$$\text{Compression Ratio} = \frac{\text{size after compression}}{\text{size before compression}}$$

Compression Gain: The gain represents the space saved after compression. It is expressed as:

$$\text{Compression Gain} = \frac{\text{size before compression} - \text{size after compression}}{\text{size before compression}}$$

Compression Savings Percentage: It is expressed as:

$$\text{Compression Savings Percentage} = \frac{\text{size before compression} - \text{size after compression}}{\text{size before compression}}$$

Compression and Decompression Speed: It is important that compression and decompression be of acceptable speed. By speed, we refer to the time required for compression or decompression.

1.5 Machine Learning and Deep Learning

Machine Learning (ML) and Deep Learning (DL) are fundamental areas within artificial intelligence that enable machines to learn from data and perform complex tasks[5].

1.5.1 Machine Learning

Machine Learning (ML) or automatic learning is a subsection of the field of artificial intelligence in computer science, which aims to teach machines to perform a task without being explicitly programmed, using algorithms referred to as models and data.

Methods of Machine Learning:

- **Supervised Learning**: Supervised learning takes labeled data and creates a model using this data to make predictions that generalize to new data. Types of supervised learning include:
  - **Classification**: Predicts a label for a model (0 or 1). It can be binary or multiclass.
  - **Regression**: Predicts a continuous value.
- **Unsupervised Learning**: Unsupervised learning is a type of machine learning algorithm used to make inferences from datasets consisting of input data without labeled responses.
- **Semi-supervised Learning**: Semi-supervised learning combines both labeled data and unlabeled data of the same type.
- **Reinforcement Learning**: Reinforcement learning is a technique in which an agent is immersed in an environment where it interacts with its environment to learn.

1.5.2 Deep Learning

Deep Learning (DL) is a branch of Machine Learning entirely based on artificial neural networks.

1.5.3 Machine Learning vs Deep Learning

The major difference between these two concepts comes from the way data is presented to the system (model):

- Machine Learning algorithms almost always require structured data, whereas deep learning networks rely on layers of artificial neural networks (ANNs).
- There is also a difference in the architecture of the models they comprise; deep learning models tend to be deeper than traditional machine learning models.
- Deep learning exclusively uses neural networks, whereas for machine learning, neural networks are just one approach to model design among many others.

1.6 Reasons for Using Traditional Machine Learning over Deep Learning

Traditional machine learning offers several advantages over deep learning, making it preferable in certain contexts[6]:

Simplicity and Interpretability

Traditional machine learning algorithms, such as decision trees, logistic regression, and support vector machines, offer simplicity and interpretability, which are valuable in domains where understanding the underlying mechanisms of compression algorithms is important.

Data Efficiency

Deep learning models typically require large amounts of labeled data to train effectively. In contrast, traditional machine learning algorithms can often perform well with smaller datasets, which may be more practical in scenarios where data collection is limited.

Computationally Efficiency

Deep learning models, especially large neural networks, require significant computational resources for training and inference. Traditional machine learning algorithms are often computationally more efficient and may be more suitable for deployment on resource-constrained devices.

Domain-specific Knowledge

Traditional machine learning algorithms allow domain experts to incorporate domain-specific knowledge into the feature engineering process, which can lead to more effective models. In contrast, deep learning models often rely on automatically learned features, which may not capture domain-specific nuances as effectively.

Robustness to Noise and Outliers

Traditional machine learning algorithms can be more robust to noisy or incomplete data compared to deep learning models, which may be prone to overfitting in such scenarios.

Interpretation and Debugging

Traditional machine learning models provide clear insights into feature importance, model coefficients, and decision boundaries, facilitating model interpretation and debugging. This interpretability can be valuable in understanding the behavior of compression algorithms and diagnosing any issues that arise.

Conclusion

In this chapter, we have explored the significance of data compression and its crucial role in optimizing the utilization of storage and transmission resources by minimizing the amount of required data. We have also examined in detail the two main categories of compression methods, namely those focusing on faithful recovery of the original source and those based on data modeling and encoding. Additionally, we have provided a general overview of artificial neural networks, exploring their layers and some of the activation functions used.

Chapter 2

Data Compression Algorithms

Introduction

Data compression is widely adopted by most users because it offers the opportunity to save storage space and speed up data transfer between individuals. This process relies on compression techniques that are not limited to textual data but also include images and videos. Data compression technology is divided into two main categories, namely lossless compression and lossy compression. Several compression methods are available, including Huffman, Shannon-Fano, Lempel-Ziv Welch, and run-length encoding. Each of these methods has distinct compression capabilities. We will examine these techniques in detail to assess how they reduce the size of a compressed file compared to the original.

2.1 Shannon-Fano Coding

Shannon-Fano coding is a lossless data compression technique that was independently developed by Claude Shannon and Robert Fano in the 1940s. It is an entropy coding method, which means it assigns variable-length codes to symbols based on their probability of occurrence. This technique is one of the earliest attempts to create efficient codes for data compression [4].

2.1.1 Principle of the Algorithm

- **Probability Calculation:** For each symbol in the dataset to be compressed, the Shannon-Fano coding calculates its probability of occurrence. More frequent symbols typically receive shorter codes, while less frequent symbols receive longer codes.
- **Symbol Sorting:** The symbols are sorted in descending order of probability. This rearranges them so that the most frequent symbols are associated with shorter codes.
- **Recursive Division:** The symbols are then divided into two groups in a way that the sum of probabilities in each group is approximately equal. Then, prefixes 0 and 1 are assigned to the two groups.
- **Repetition:** The process of recursive division is repeated for each group of symbols until each symbol is associated with a unique code.

Example:

Consider the following figure representing the frequency of occurrence of symbols in an alphabet $A = \{A, B, C, D, E\}$ [7]. The statistics are extracted from a text:

> "Traces are found in many different places that differ according to their physical nature, geographical location, and climate,...etc. Therefore, these places are classified according to their characteristics into three categories" 2.1.

Symbol	**A**	**B**	**C**	**D**	**E**
Occurrences	15	7	6	6	5
Frequency	$\frac{15}{39}$	$\frac{7}{39}$	$\frac{6}{39}$	$\frac{6}{39}$	$\frac{5}{39}$

Table 2.1: Symbol Occurrence Table

The application of the algorithm gives us the following tree**??**:

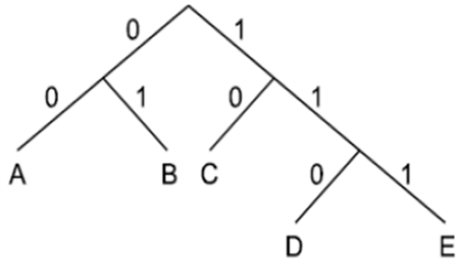


Figure 2.1: Application of the Shannon-Fano Algorithm

From the tree, we can extract the code for each symbol. This gives us the following table 2.4:

Symbole	**Mot**
A	00
B	01
C	10
D	110
E	111

Table 2.2: Code Shannon-Fano

Advantages	**Disadvantages**
Produces prefix codes, significant reduction in the size of the compressed file	The size of the codes increases with the number of symbols in the alphabet used

Table 2.3: Advantages and Disadvantages of the Shannon-Fano Algorithm

2.2 Huffman Coding

Huffman coding is a lossless data compression technique invented by David A. Huffman in 1952. It is an entropy encoding method that assigns variable-length codes to symbols based on their probabilities of occurrence. The most frequent symbols are associated with shorter codes, while less frequent symbols receive longer codes. This method is widely used for data compression, particularly in the fields of text processing, data transmission over networks, and image compression [2].

2.2.1 Principle of the Algorithm

- Each node of the Huffman tree has an associated frequency representing the number of occurrences of that symbol in a given sequence of symbols.
- In the Huffman tree, the leaves represent individual symbols with their occurrence frequencies in a sequence. Internal nodes of the tree have a frequency equal to the sum of the frequencies of their children.

Once the tree is constructed, we can obtain the codeword for each symbol of the alphabet. To do this, we simply accumulate the bits in the root-to-leaf order (each leaf represents a symbol). Therefore, we can retrieve the codeword associated with a symbol in linear time relative to the number of bits it contains[7].

Algorithm 1: Huffman Coding Algorithm

Input: Symbols with their probabilities

Output: Encoded data using Huffman Coding

Sort symbols by probability in descending order;

while *At least two symbols remain* **do**

> Pair up the two least probable symbols;
>
> Assign '0' to one symbol and '1' to the other;
>
> Combine them into a composite symbol with sum of probabilities;

end

Assign codewords to symbols based on tree traversal;

Example:

Taking the example from the previous section, the result of applying Huffman coding yields the following tree2.2:

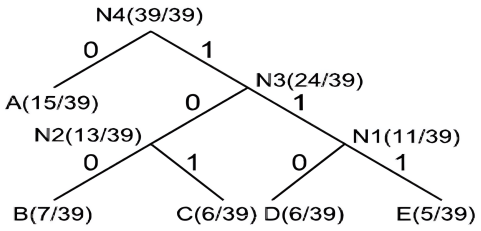


Figure 2.2: Application of the Huffman Algorithm.

Here is a table representing the codewords for each symbol and their probabilities of occurrence :

Symbole	**Mot**
A	0
B	100
C	101
D	110
E	111

Table 2.4: Code Shannon-Fano

Here's a comparison between the number of bits required to encode the text using Shannon-Fano coding and Huffman coding based on the results obtained in the previous example :

Number of bits with Shannon-Fano = $P(A)\times2+P(B)\times2+P(C)\times2+P(D)\times3+P(E)\times3 = 2.28$ bits.

With Huffman coding:

Number of bits with Huffman = $P(A)\times1+P(B)\times3+P(C)\times3+P(D)\times3+P(E)\times3 = 2.23$ bits.

Where $P(X)$ represents the probability of symbol X. According to the results obtained, we can observe that Huffman coding is more optimal than Shannon-Fano coding.

Advantages	**Disadvantages**
Huffman coding provides optimal character-by-character compression, meaning that shorter binary codes cannot be obtained for characters. It also has the advantage of being easy to implement programmatically, and the execution time is rather fast.	Certaines méthodes de compression qui encodent des mots plutôt que des caractères peuvent offrir un taux de compression plus intéressant. Ainsi, le codage de Huffman est souvent utilisé en conjonction avec d'autres techniques de compression, telles que LZW.

Table 2.5: Advantages and Disadvantages of the Shannon-Fano Algorithm

2.2.2 Adaptive Huffman Coding

Huffman coding has the disadvantage of requiring prior knowledge of the symbol frequencies to construct the tree used in generating the codewords. This requirement involves a two-pass procedure during encoding: one pass to collect statistics and a second pass to perform the actual data encoding. This characteristic of Huffman coding can be problematic in situations where data retention is difficult or when used in real-time applications.

Adaptive Huffman coding is a variant of Huffman coding that avoids the need to know the symbol frequencies in advance. Instead, the Huffman tree used to generate the codewords is updated as the data is read. This means that there is no longer a need for two separate steps to encode the data; a single pass is sufficient. Additionally, a decoder no longer needs to be informed about the Huffman tree used to generate the codewords, as it can construct the Huffman tree itself using the same approach as the encoder. Of course, this technique is a bit more complex and we won't delve into it in detail here. However, the basic idea is as follows[8] 2.

Algorithm 2: Adaptive Huffman Coding Algorithm

Input: Data to be encoded

Output: Encoded data using Adaptive Huffman Coding

Initialize a Huffman tree containing all symbols of an alphabet, assigning a default probability to each symbol;

while *There is data to be encoded* **do**

  Read the next symbol s to be encoded;

  Encode the symbol;

  Update the Huffman tree to account for the appearance of a new symbol s;

end

In the context of adaptive Huffman coding, the Huffman tree undergoes updates each time new symbols are encountered. Fortunately, more efficient techniques exist for updating the Huffman tree and the associated codewords.

2.3 Arithmetic Coding

Arithmetic coding operates by encoding a sequence of symbols using a fractional representation within the real interval [0,1]. It employs a recursive procedure assigning a subrange to each symbol, with the main characteristic being the reduction of the real interval's width based on the probability associated with the symbol being encoded. The reduction of the interval is smaller for more probable symbols.

What sets arithmetic coding apart from other source encodings is that it encodes the entire message and represents it with a single floating-point number r, unlike other encodings that segment the input message into individual symbols and then encode each symbol with a codeword.

For each symbol in the source alphabet, an interval is associated within [0,1]. The size of the interval is proportional to the probability of the symbol, expressed as

$$a_i \in A \rightarrow [r_i, r_i + P(a_i)] \textbf{with } r_0 = 0.$$

To encode a message, which is a string of symbols, the first symbol determines the code's interval (the lower bound is chosen). This interval is then subdivided using the same principle (probabilities) for each subsequent symbol read, and the subrange of the next symbol is chosen accordingly (see Algorithm) 3.

Algorithm 3: Arithmetic Coding Algorithm

Input: Sequence of symbols to be encoded

Output: Encoded representation of the sequence

Initialize $Inf = 0$ and $Sup = 1$;

while *there are symbols in the sequence* **do**

$Sup = Inf + (Sup - Inf) \times BornSup[C]$;

$Inf = Inf + (Sup - Inf) \times BornInf[C]$;

$C =$ ReadNextSymbol();

end

At the end, the value of Inf represents the code of the message, which can also be any value in the interval $[Inf, Sup[$ [9].

2.3.1 Exemple de codage arithmétique

Soit $A = \{a, b, c, d\}$ avec une probabilité de chaque symbole $0.2, 0.5, 0.2, 0.1$.

- $a \rightarrow [0, 0.2[$
- $b \rightarrow [0.2, 0.7[$
- $c \rightarrow [0.7, 0.9[$
- $d \rightarrow [0.9, 1[$

Ainsi, pour coder a, on peut utiliser n'importe quelle valeur dans $[0, 0.2[$. La même chose pour b, c et d autant que symboles.

Pour coder maintenant un message, on applique l'algorithme. Par exemple, pour coder *cbaabd*:

- Le premier symbole est c, on choisit l'intervalle $[0.7, 0.9[$.
- Le symbole suivant est b, donc le nouvel intervalle est:

$$[0.7 + (0.9 - 0.7) \times 0.2, 0.7 + (0.9 - 0.7) \times 0.7[= [0.74, 0.84[$$

- Le symbole suivant est a, donc le nouvel intervalle est:

$$[0.74 + (0.84 - 0.74) \times 0, 0.74 + (0.84 - 0.74) \times 0.2[= [0.74, 0.76[$$

Symbole	**Borne Inf**	**Borne Sup**
C	0.7	0.9
B	0.74	0.84
A	0.74	0.76
A	0.74	0.744
B	0.7408	0.741
D	0.7426	0.7428

Table 2.6: Codage Arithmétique par table

Le code de **cbaabd** est donc 0.7426.

More details can be found in the following figure 2.3.

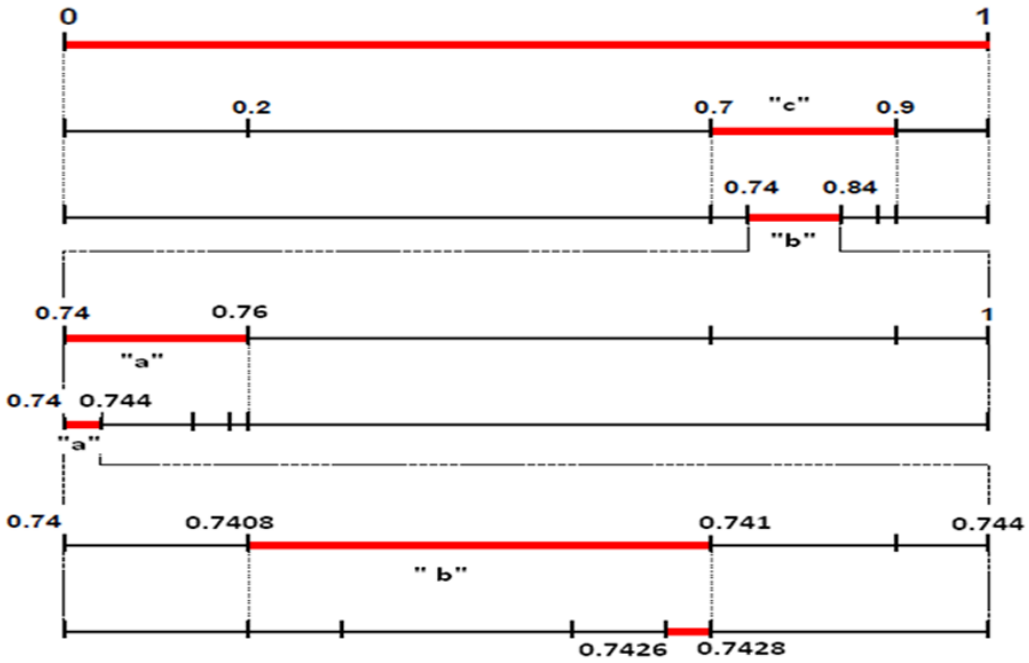


Figure 2.3: Arithmetic Coding Diagram

2.3.2 Arithmetic Decoding Algorithm

In the application to the example, we find:

$$
\begin{aligned}
r &= 0.7426 \in [0.7, 0.9[\rightarrow c \\
r &= \frac{0.7426 - 0.7}{0.2} = 0.213 \in [0.2, 0.7[\rightarrow b \\
r &= \frac{0.213 - 0.2}{0.5} = 0.026 \in [0, 0.2[\rightarrow a \\
r &= \frac{0.026 - 0}{0.2} = 0.13 \in [0, 0.2[\rightarrow a \\
r &= \frac{0.13 - 0}{0.2} = 0.65 \in [0.2, 0.7[\rightarrow b \\
r &= \frac{0.65 - 0.2}{0.5} = 0.9 \in [0.9, 1[\rightarrow d
\end{aligned}
$$

2.4 Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is a data compression technique that reduces the size of repetitive sequences. The RLE algorithm replaces consecutive repeated characters or symbols with a count of the repetition followed by the character itself [10].

Algorithm 4: Arithmetic Decoding Algorithm

Input: Encoded value R

Output: Decoded message

Initialize Message $=''$;

Repeat;

if $r_i \in [r_i, r_i + P(a_i)[$ **then**

  Message = Message + a_i;

  $R = \frac{r_i}{P(a_i)}$;

end

until the entire message is decoded;

2.4.1 Steps of the RLE Algorithm

1. Start with an input data sequence.
2. Initialize an empty output sequence.
3. Traverse the input sequence from left to right.
4. Each time a character is encountered, count the number of consecutive occurrences of that character.
5. Add the count followed by the character to the output sequence.
6. Move to the next character in the input sequence and repeat steps 4 and 5 until the entire input sequence is processed.
7. The resulting output sequence is the compressed representation of the input data using RLE encoding.

Example: Consider the following character sequence: `AAAAAAARRRRROOO`, which is encoded in 15 bytes. Applying the RLE encoding function yields the following sequence: `7A5R3O`, which encodes in 6 bytes, resulting in a savings of 9 bytes.

2.4.2 Advantages and Disadvantages

- **Advantages:**

  - **Simplicity:** The RLE algorithm is simple to implement and understand.
  - **Effective Compression for Repetitive Data:** It efficiently compresses repetitive data.
  - **Fast Compression and Decompression Times:** Compression and decompression are performed quickly.
  - **Reduced Memory Usage:** It consumes less memory.
  - **Data Integrity Preservation:** It preserves data integrity.

- **Disadvantages:**
  - **Low Efficiency for Non-Repetitive Data:** RLE is less efficient for non-repetitive data.
  - **Loss of Efficiency for Small Repetitive Sequences:** It loses efficiency for small repetitive sequences.
  - **Sensitivity to Transmission Errors:** RLE is sensitive to transmission errors.
  - **Pre-Processing Requirement:** Pre-processing is needed to optimize compression.

2.5 Move To Front (MTF)

The Move To Front (MTF) algorithm, also known as "move-to-front," is a data reorganization technique used to improve the efficiency of data compression. It is commonly used in conjunction with other compression methods such as Huffman coding. The principle of the MTF algorithm is relatively simple and involves rearranging symbols in a list based on their usage, so that the most frequently used symbols are placed at the front of the list[11].

2.5.1 Algorithm Principle

- **List Initialization:** Initially, the list is usually initialized with symbols in a fixed order, such as the order of appearance in the character set.

- **Initial Coding:** When a symbol is encountered in the data to be compressed, it is searched for in the list. Once found, its current position in the list is used for coding it (for example, using the number of movements from the beginning of the list).
- **List Reorganization:** After coding a symbol, it is moved forward (upward in the list). Other symbols are also rearranged based on the order in which they were used. Symbols that are used frequently thus end up closer to the beginning of the list, while those that are less frequently used are towards the back of the list.
- **Subsequent Coding:** The following symbols are coded based on their new position in the rearranged list.

Example: The sequence "EEEEEA" would be transformed into the sequence "400001"; the table would evolve as follows:

Index	0	1	2	3	4	5	6	...	25
Character	A	B	C	D	E	F	G	...	Z

Modified Table by the First "E":

E	A	B	C	D	F	G	...	Z

Table Retained by the Following "E":

E	A	B	C	D	F	G	...	Z

Modified Table by the "A":

A	E	B	C	D	F	G	...	Z

The decoding process is also simple: using the same initial matrix, you just need to take the character corresponding to the index and rearrange the matrix by placing this character in the first position. The matrix transforms in the same way as during the encoding phase.

In summary, the MTF algorithm offers simplicity of implementation and efficient compression for certain types of data. However, it may be less effective for variable data, has a relatively high time complexity, and lacks adaptability to changes in character frequencies.

2.6 Lempel-Ziv77 (LZ77)

In 1977, Jacob Ziv and Abraham Lempel introduced the lossless compression algorithm known as LZ77. Its principle is to compress a sequence by exploiting repetitions of subsequences within that sequence. The algorithm traverses the sequence from left to right. The left part, already traversed, is assumed to be already compressed, while the right part is the part remaining to be compressed. A pointer is positioned at the first symbol of the uncompressed part. The algorithm then searches for the longest substring matching the beginning of the uncompressed part in the left part. When a match is found, its position (relative to the current pointer), its length (the match), and the first symbol that does not match in the right part are encoded. Thus, we obtain tuples of the form P_i, L, Q_i, where P represents the position, L represents the length of the match, and Q represents the next character.
For decompression, we read tuple by tuple, and add the designated segment to the decompressed sequence[12].

Example: Let's apply the LZ77 compression algorithm to the character sequence `cabracadabrarrarrad`.

1. **Initialization:** Begin with an empty search buffer and a lookahead buffer containing the entire sequence.

2. **Search for Matches:** Starting from the beginning of the lookahead buffer, find the longest matching substring in the search buffer.

3. **Encoding Matches:** Encode each match found as a triple (P, L, C), where:

   - P: Position of the matching substring in the search buffer.
   - L: Length of the match.
   - C: The next character after the match.

4. **Updating Buffers:** Move the search buffer forward by the length of the match, and add the next character after the match to the search buffer.

5. **Repeat:** Repeat steps 2-4 until the lookahead buffer is empty.

Search Buffer	Lookahead Buffer	Encoding
-	cabracadabrarrarrad	-
cabr	acadabrarrarrad	(0,0,'c')
abrac	adabrarrarrad	(3,1,'a')
cadabr	arrarrad	(2,1,'d')
abrarr	arrad	(7,4,'r')
arrar	rad	(3,5,'d')

Table 2.7: LZ77 Compression Steps

2.7 Lempel-Ziv78 (LZ78) Algorithm

The Lempel-Ziv78 (LZ78) algorithm is another lossless compression algorithm that builds a dictionary of substrings encountered in the input data. Unlike LZ77, which uses a sliding window approach, LZ78 builds its dictionary dynamically as it processes the input sequence[13].

Basic Steps of LZ78:

1. **Initialization:** Start with an empty dictionary.
2. **Parsing Input:** Parse the input sequence from left to right.
3. **Building the Dictionary:** As each new substring is encountered, add it to the dictionary.
4. **Encoding:** Encode each substring encountered as a pair (P, C), where:
   - P: Index of the longest prefix of the substring already in the dictionary.
   - C: The next character after the prefix.
5. **Repeat:** Repeat steps 2-4 until the entire input sequence is processed.

In this example, we start with an empty dictionary. As we parse the input sequence "abracadabra" from left to right, we add substrings encountered to the dictionary and encode each substring using the index of the longest prefix already in the dictionary along with the next character. The table above shows the compression steps for the input sequence "abracadabra" using LZ78.

Dictionary Index	Next Character	Encoding
0	a	(0, 'a')
0	b	(0, 'b')
0	r	(0, 'r')
0	a	(0, 'a')
4	c	(4, 'c')
0	d	(0, 'd')
4	a	(4, 'a')
6	b	(6, 'b')

Table 2.8: LZ78 Compression Steps for Input Sequence "abracadabra"

2.8 Comparison between LZ77 and LZ78 Algorithms

In the realm of data compression, the LZ77 and LZ78 algorithms represent two pioneering approaches that have significantly impacted the field[14].

2.8.1 LZ77 Algorithm

1. Operates on past data.
2. Slower compared to LZ78.
3. Output format: Triplet <o, l, c>, where o=offset, l=length of match, c=next symbol to encode.
4. Applications: Open source and widely used in ZIP, PNG, TIFF, PDF, etc. Also used in gzip, Squeeze, LHA, PKZIP, ZOO.

2.8.2 LZ78 Algorithm

1. Attempts to work on future data.
2. Faster than LZ77.
3. Output format: Pair <i, c>, where i=index and c=next character.

4. Applications: Various applications in information theory such as random number generation, hypothesis testing, string analysis, etc. Used in compression, GIF, CCITT (modems), ARC, PAK.

The LZ77 algorithm operates on past data, while the LZ78 algorithm attempts to work on future data. LZ78 analyzes the input buffer ahead and compares it to a dictionary it manages. It scans through the buffer until it finds no match in the dictionary. At this point, it displays the location of the word in the dictionary, if any, the length of the match, and the character that caused the match failure. The resulting word is then added to the dictionary.

Conclusion

In this chapter, we have detailed some lossless compression algorithms. These lossless compression algorithms play a crucial role in the field of data management, file transmission, and storage space conservation. They also enable efficient data handling in many computer applications. Their ability to reduce data size while preserving information integrity makes them invaluable in our ever-evolving digital world.

Chapter 3

System design

Introduction

In this chapter, we propose a preprocessing method for data compression by modifying the position of the data through a block exchange process between them, based on the minimum distance between one block and another. Then, we present the prediction model for validating block permutation.

3.1 Preprocessing Method

The system starts by reading the file block by block (each block contains values in the range [-128..127]). Then, for each block b, it computes an occurrence matrix Occ_mat. This latter is used to generate a permutation list L. Subsequently, this list is used to generate b', which represents a much more compressible block than b. After that, b' is compressed, and the compression result is written into the target compressed file [15].

3.1.1 Processing Steps

We have the file f with size t.

Step 1: We divide the file f into a set of blocks b. The size of each block is 512×512 bytes.

Step 2: Each block is then divided into sub-blocks. The size of each sub-block is 512 bytes.

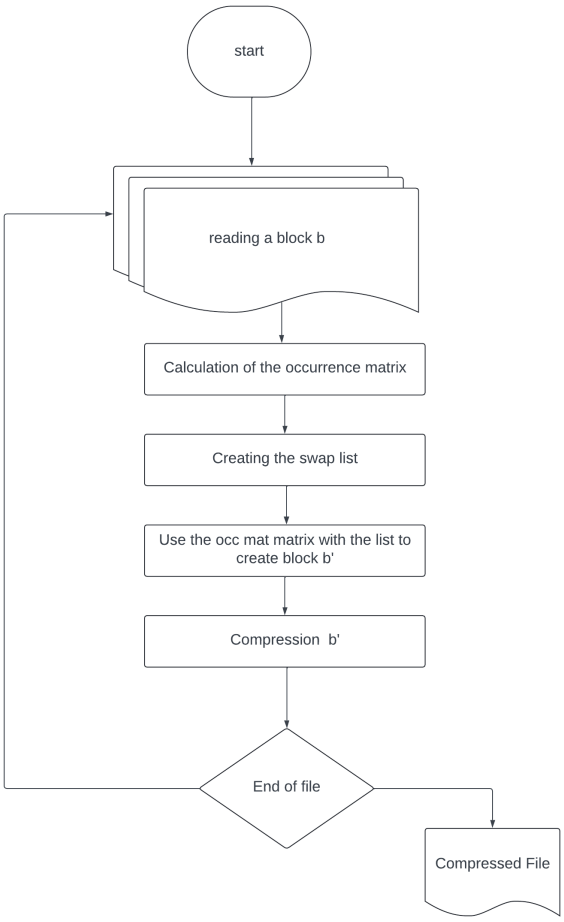


Figure 3.1: General system diagram

Each sub-block contains values from -128 to 127.

Example sub-block

Step 3: Creation of the Occurrence Matrix:

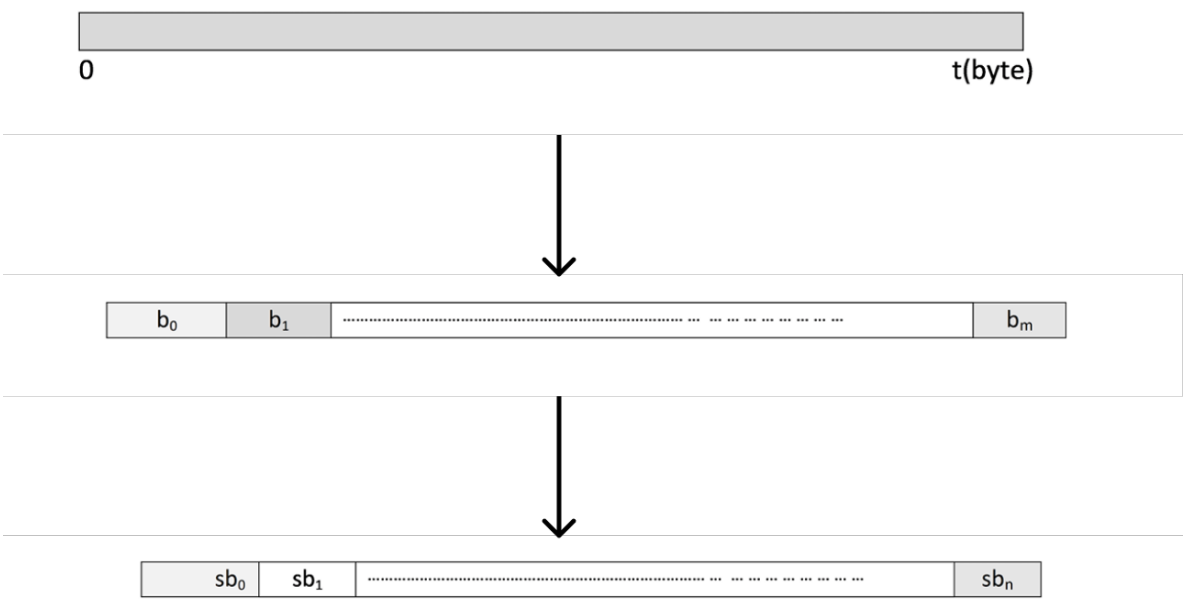


Figure 3.2: Division of the file into blocks.

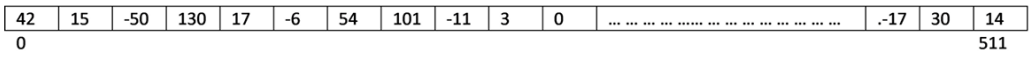


Figure 3.3: Example of a sub-block.

The matrix contains one block:

- Rows of the Occurrence Matrix (`Occ_mat`) = number of sub-blocks = 511
- Columns of the Occurrence Matrix (`Occ_mat`) = 129

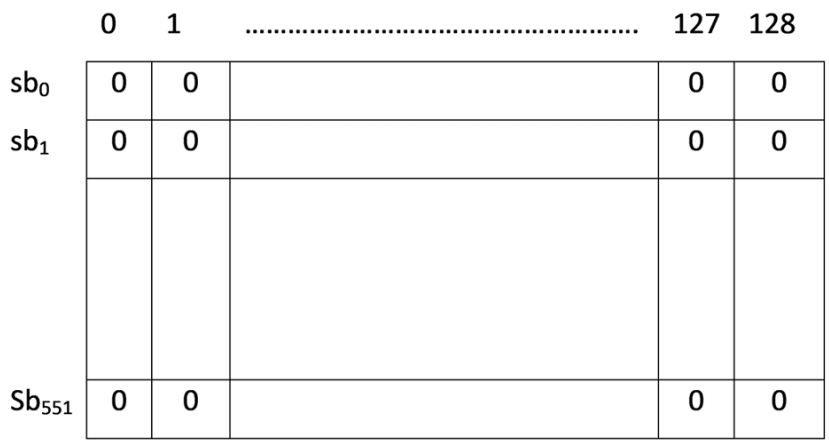
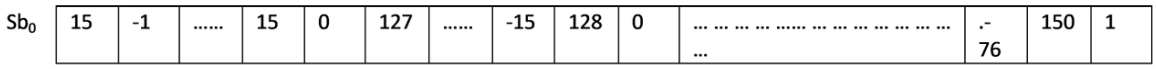


Figure 3.4: Initial Occurrence Matrix

Suppose we have two sub-blocks.



and

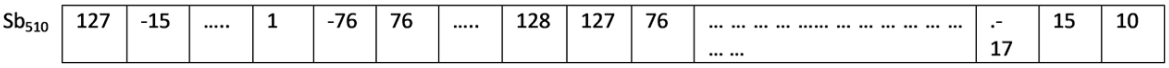


Figure 3.5: for two sub-blocks.

The matrix is as follows:

	0	1	....	15	....	76		....	150	151	... ...		127	128
Sb_0	2	2	.....	3	.....	1	0	.....	1	0	... ...	....	1	1
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
Sb_{510}	0	1	.....	2	.....	3	.....	.....	0	0	... ...	...	2	1
Sb_{511}	....	....	.....	....	....	....	.....	.....	....	....	... ...	....	....	....

Figure 3.6: The matrix after filling the sub-block.

Step 4: Creation of the permutation list L: Using the occurrence matrix, we can calculate the distance between the partial blocks in order to rearrange them and create another sequence (permutation list). Calculate the distance between the sub-blocks:

$$d(sb_0, sb_1) = \sqrt{\sum_{j=0}^{128} (\text{occ\_mat}[sb_{0j}, sb_{0j}] - \text{occ\_mat}[sb_{1j}, sb_{1j}])^2}$$

$$d(sb_0, sb_2) = \sqrt{\sum_{j=0}^{128} (\text{occ\_mat}[sb_{0j}, sb_{0j}] - \text{occ\_mat}[sb_{2j}, sb_{2j}])^2}$$

...

$$d(sb_0, sb_{255}) = \sqrt{\sum_{j=0}^{128} (\text{occ\_mat}[sb_{0j}, sb_{0j}] - \text{occ\_mat}[sb_{255j}, sb_{255j}])^2}$$

We choose the minimum distance:

$$\text{Min}(D(sb_0, sb_1), D(sb_0, sb_2), \ldots, D(sb_0, sb_{255}))$$

Suppose sub-block 15 has the minimum distance. We repeat this process until the end.

0	15
1	sb_k
2	..
3	..
...	..
15	40
...	..
40	..
sb_k	sb_p
...	..
sb_p	..
...	..
254	110
511	

Figure 3.7: List of permutations.

Step 5: In this step, rearrange the sub-blocks according to the permutation list.

	0	1	.....	15	.....	76		.....	150	151	... ...		127	128
Sb_0	2	2	.....	3	.....	1	0	.....	1	0	... ...	....	1	1
15	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
40	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
sb_k	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
Sbp	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
sbp	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
	....	....	.....	....	.....	....	.....	.....	....	....	... ...	....	....	....
Sb_{510}	0	1	.....	2	.....	3	.....	.....	0	0	... ...	...	2	1
Sb_{511}	....	....	.....	....	....	....	.....	.....	....	....	... ...	....	....	....

Figure 3.8: Rearrange the sub-blocks.

Step 6: Compress and save the block b' with the permutation list L.

3.1.1.1 Calculation of the Occurrence Matrix

To calculate the occurrence matrix, we need to input the block b containing values in the range [-128..127], and the size of sub-blocks is $sbs = 512$. The columns of this matrix represent the numbers of the temporal distance described earlier, and the rows represent the ID (order) of the sub-block within the block b. This process involves dividing the block b into a set of sub-blocks of size sbs. We count the number of occurrences of each number and assign it to the current column and row 3.9.

createOccMat.py

```
def createOccMat(buf,buf_len, mat, sbs ):
    sbn = len(buf) / sbs
    mat= np.zeros((len(mat), len(mat[0])))
    for i in range(0, int(sbn)):
        s = i * sbs
        e = ((i + 1) * sbs)
        if e > buf_len:
            e = buf_len
        for j in range(s, ((i+1) * sbs)):
            if j < buf_len:
                mat[i][abs(buf[j]-128)] = mat[i][abs(buf[j]-128)] + 1
    return mat
```

Figure 3.9: Calculation of the Occurrence Matrix

3.1.1.2 Generation of the Permutation List

We have the occurrence matrix. Using the occurrence matrix, we can calculate the distance between the sub-blocks in order to rearrange them and create another sequence (permutation list), and obviously another block b'. Therefore, generating the permutation list requires as input the occurrence matrix and the use of two functions: one that calculates the distance between two sub-blocks and the other that generates the permutation list 3.10.

createMatchList.py

```
def createMatchList(mat, mat_len):
    old_dist = 9223372036854775808
    idx = -1
    temp_list = []
    list=[-1 for i in range(mat_len)]
    ci=0
    for iterations in range(0, mat_len):
        idx = -1
        old_dist = 9223372036854775808
        for i in range(1, mat_len):
            if(i not in temp_list)and(ci != i):
                new_dist = doubleEuclideanDistance(mat, ci, i)

                if (new_dist<old_dist):
                    old_dist = new_dist
                    idx = i
        if(idx in temp_list):
            sys.exit("Element fourd...!")
        else:
            temp_list.append(idx)
        list[ci]=idx
        ci=idx
    return list
```

Figure 3.10: Generation of the Permutation Lis

3.1.1.3 Function to Calculate the Distance Between Two Rows

The function `doubleEuclideanDistance` calculates the Euclidean distance between two rows of a given matrix. The detailed steps involved in this function are as follows:

1. Initialize a variable `sum` to zero. This variable will store the sum of the squared differences between the corresponding elements of the two rows.
2. Iterate over each element in the rows specified by `p1` and `p2`.
3. For each element, calculate the difference between the elements in the two rows.
4. Square the difference and add it to `sum`.
5. After completing the iteration, calculate the square root of `sum` to obtain the Euclidean distance.
6. Return the calculated Euclidean distance.

The function can be implemented as follows:

EuclideanDistance.py

```
def doubleEuclideanDistance(mat, p1, p2):
    sum = 0

    for i in range(0, len(mat[p1])):

        d = mat[p1][i] - mat[p2][i]
        sum += d * d
    return  math.sqrt(sum)
```

Figure 3.11: General process of calculating the Euclidean distance between two rows

3.1.1.3 Creating a Prime Buffer

The function `createPrimeBuffer` creates a new list by copying specific segments from a source list (`src_buf`). The size of each segment is determined by the variable `sbs` (segment size). Another list (`list`) specifies the order of the segments to be copied.

The function iterates a number of times based on the length of the source list (`buf_len`) divided by the segment size (`sbs`). In each iteration, it copies a segment from the source list to the new list. It uses the `list` parameter to determine the next segment to copy. This process continues until all required segments are copied.

Finally, the function returns the new list, which contains the copied segments from the source list 3.12.

createPrimeBuffer.py

```
def createPrimeBuffer(src_buf, buf_len, list, sbs):
    buf = []
    i = 0
    k = 0
    sbn = buf_len / sbs

    for n in range(0, int(sbn)):
        s = i * sbs
        for j in range(s,((i+1) * sbs)):
            if j < buf_len:
               buf.append(src_buf[j])
               k =k + 1
        i = list[i]
    return buf
```

Figure 3.12: Creating a Prime Buffer

3.2 machine learning model

the model we're building is indeed for prediction. Specifically, it's a binary classification model that predicts whether a given text document from the Silesia dataset is compressed or uncompressed.

Once trained, the model takes as input the numerical features derived from the text data (using TF-IDF vectorization) and predicts the compression status of new, unseen documents. The prediction is a binary outcome, where the model classifies a document as either compressed (1) or uncompressed (0) based on its learned patterns from the training data.

3.2.1 Choosing the Classification Method

The Random Forest Classifier model was selected. This model is suitable for classification tasks due to its ability to handle imbalanced data and provide good performance without the need for extensive parameter tuning.

3.2.2 Detailed Steps

Step 1: Data Splitting and Preparation

The contents of the text files were read and converted into a list called `file_contents`. The data was split into a training set (`X_train`) and a test set (`X_test`) using the `train_test_split` function from the `sklearn.model_selection` module, with 20% of the data allocated for testing. The sample balance was maintained by using `random_state=42` to ensure reproducibility 3.13.

LoadDataset.py

```
import os

file_paths = ["abbot", "bovary", "corilis", "cyprus", "emission", "firewrks", "flower", "hungary", "lusiadas", "modern"..]

compression_labels = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

file_contents = []
for file_path in file_paths:
    with open(file_path, 'r', encoding='latin-1') as file:
        content = file.read()
        file_contents.append(content)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(file_contents, compression_labels, test_size=0.2, random_state=42)
```

Figure 3.13: Data Splitting and Preparation

Step 2: Text Vectorization Using TF-IDF

We used `TfidfVectorizer` to convert the texts into a numerical representation based on Term Frequency-Inverse Document Frequency (TF-IDF). This method helps to enhance the important words in the documents and reduce the impact of common words 3.14.

TF-IDF Vectorization.py

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

Figure 3.14: Text Vectorization Using TF-IDF

Step 3 : Model Creation and Training

The Random Forest model was created with balanced class weights (`class_weight='balanced'`). This is useful for handling imbalanced data to ensure that the model does not bias towards the most frequent class 3.15.

train.py

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(class_weight='balanced', random_state=42)
clf.fit(X_train_tfidf, y_train)
```

Figure 3.15: Model Creation and Training

Step 4 : Prediction and Model Evaluation

The trained Random Forest model was used to make predictions on the test set (`X_test`), and its performance was evaluated using various metrics such as accuracy, precision, recall, and F1-score 3.16.

Prediction.py

```
from sklearn.metrics import classification_report, accuracy_score

y_pred = clf.predict(X_test_tfidf)

print(classification_report(y_test, y_pred))

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Figure 3.16: Prediction and Model Evaluation

Step 6: Save Model and Vectorizer

The trained Random Forest model (RandomForestClassifier) and the text vectorizer (TfidfVectorizer) were saved to files using pickle, enabling them to be reused later without the need for retraining. This approach provides a robust and efficient method for text classification based on its content, using a trusted machine learning model and common text processing tools. This process ensures that once a model is trained and a vectorizer is fitted to the data, they can be easily stored and loaded for future use without the overhead of retraining, making the system highly efficient and scalable. The implementation of this method is depicted in Figure 3.17.

Save Model and Vectorizer.py

```python
import pickle

with open('model_compression.pkl', 'wb') as file:
    pickle.dump(model, file)

with open('vectorizer.pkl', 'wb') as file:
    pickle.dump(vectorizer, file)
```

Figure 3.17: Save Model and Vectorizer

The general design of the system is depicted in the following figure (3.18). This figure provides an overview of the model and its components.

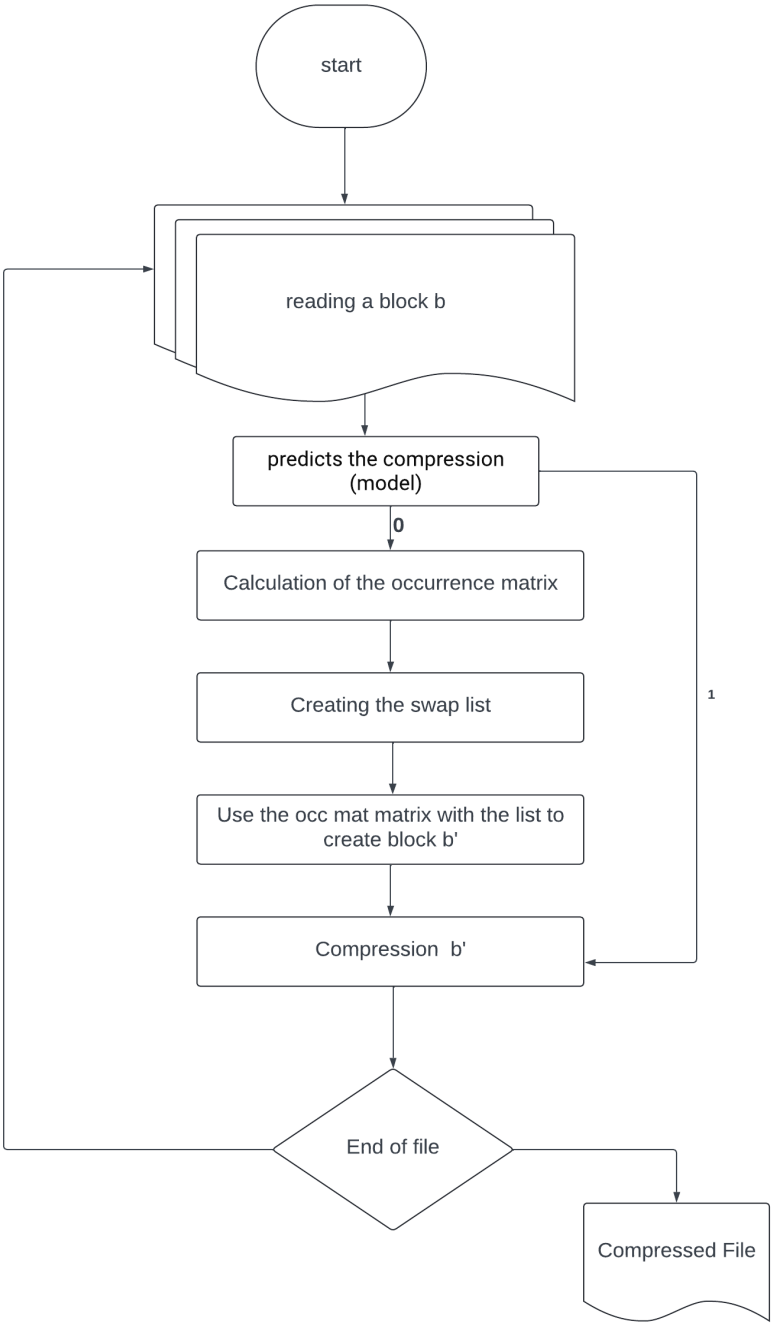


Figure 3.18: General system diagram with the model

Conclusion

In this chapter, we presented the general structure of the preprocessing method proposed system, which is based on block data permutation. Additionally, we outlined the general structure of the model for predicting blocks before applying the permutation method.

Chapter 4

Test and Results

Introduction

In this chapter, we will provide a brief description of the development tools used, such as the chosen programming language and integrated libraries. Additionally, we will give an overview of the dataset used. Furthermore, we will present the test results in both cases.

4.1 Utilization of Python

Python is utilized in this study for various purposes[16]:

- Data Processing: Preprocessing and manipulation of datasets.
- Algorithm Implementation: Implementing compression algorithms and methodologies.
- Model Development: Developing predictive models.
- Experimentation and Evaluation: Conducting experiments and evaluating algorithm performance.

4.2 Prague Corpus

We acknowledge all the sources, pages, and projects used for research purposes.

Name	Original name	Size	Source	Description
abbot	abbot.jar	349,055 B	sweethome3d.eu	Part of a free interior design application Sweet Home 3D
bovary	15711-pdf.pdf	2,202,291 B	gutenberg.org	Gustave Flaubert's first published novel Madame Bovary written in German
corilis	corilis06 r5l1 c3b.tif	1,262,483 B	eea.europa.eu	CORILIS land cover data, from CORIne and LISsage (smoothing in French).
cyprus	CY meta.xml	555,986 B	eea.europa.eu	Data from AirBase (The European air quality database) for Cyprus
emissions	Waterbase Emissions v1.mdb	2,498,560 B	eea.europa.eu	This database contains data on emissions of nutrients and hazardous substances to water, aggregated within River Basin Districts (RBDs)
firewrks	27647 hanstimm FS fw2.f10.aiff	1,440,054 B	freesound.org	Sound of fireworks
flower	ower foveon.ppm	10,287,665 B	imagecompression.info	A photo of a flower, the resolution is 2268 x 1512
hungary	HU meta.xml	3,705,107 B	eea.europa.eu	Data from AirBase (The European air quality database) for Hungary
modern	15703-8.txt	388,909 B	gutenberg.org	A book Modern by Ernst Ahlgren and Axel Lundegard written in Swedish (ISO-8859-1 encoding)

Name	Original name	Size	Source	Description
thunder	24003 Erdie mega thunder.wav	3,172,048 B	freesound.org	Sound of thunder
w01vett	w01vett.dbf	1,381,141 B	nature.berkeley.edu	A database file containing information about West Nile Virus Surveillance (Veterinary cases) from 2001
ultima	30334-pdf.pdf	1,073,079 B	gutenberg.org	A Science-Fiction book by Mack Reynolds written in English
venus	pvo-uv 790226.tiff	13,432,142 B	nssdc.gsfc.nasa.gov	Ultraviolet image of Venus clouds as seen by the Pioneer Venus Orbiter (Feb. 26, 1979), the resolution is 2048 x 2188

4.3 Definition of Algorithms and Comparative

In this section, we provide definitions and comparative analysis of various compression algorithms[11].

1. **Deflate**:

   - **Description**: Deflate is an evolutionary algorithm developed by Phillip W. Katz. It employs both LZ77 and Huffman coding to compress data.
   - **Usage**: Originally designed for the .zip file format, various versions of the original Deflate algorithm are now used in different file formats and software, such as GZIP, 7-zip, etc.

2. **Deflate64**:

   - **Description**: Deflate64 is fundamentally the same as the Deflate algorithm with some enhancements.

- **Differences**: It increases the dictionary size from 32 KB to 64 KB, extends the distance codes to 16 bits to address a 64 KB range, and extends the length code to 16 bits.

3. **Bzip2**:

   - **Description**: Bzip2 is a file compression software developed by Julian Seward in 1996. It is an advanced multi-layered compression program.
   - **Characteristics**: Bzip2 provides excellent compression results at the cost of increased space and processing time. It divides the data into blocks between 100 KB and 900 KB, then compresses each block individually using the Burrows-Wheeler Transform.

4. **LZMA**:

   - **Description**: The Lempel-Ziv-Markov chain algorithm (LZMA) is a variant of the LZ77 algorithm, developed in 1998.
   - **Usage**: Initially used in the 7zip application as a compression algorithm, LZMA is known for its high compression ratio.

5. **Prediction by Partial Matching (PPM)**:

   - **Description**: The PPM algorithm, developed by Cleary and Witten in 1984, is a data compression algorithm.
   - **Characteristics**: PPM is a sophisticated technique that predicts the next symbol in a sequence based on the context provided by the preceding symbols, offering high compression efficiency at the cost of slower processing speeds.

4.4 Algorithm performance after applying the permutation method

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Compression Speed (bytes/sec)	Decompression Speed (bytes/sec)
abbot.dat	350079	315698	1.11	5954024.56	8341643.23
bovary.dat	2210483	584281	3.78	771292.92	41844282.57
corilis.dat	1266579	607782	2.08	3682038.41	13473785.92
cyprus.dat	558034	22997	24.27	163099.53	24264609.56
emission.dat	2507776	243122	10.31	758644.59	64085632.78
firewrks.dat	1445174	1189781	1.21	6287574.43	19264932.47
flower.dat	10327601	3840662	2.69	1469782.46	50635319.67
hungary.dat	3719443	132790	28.01	308677.50	131172483.65
lusiadas.dat	627712	149776	4.19	1724253.99	27294087.27
modern.dat	389933	133481	2.92	1668469.34	17724935.70
nightsht.dat	14809107	12595212	1.18	786648223.13	77358176.31
thunder.dat	3184336	1926104	1.65	3795935.57	34959149.69
ultima.dat	1077175	658368	1.64	4475795.16	28708859.91
venus.dat	13484366	8539385	1.58	2447044.70	36205462.33
w01vett.dat	1386261	75265	18.42	451819.23	63004822.64

Table 4.2: Performance of the LMZA Algorithm on Prague Corpus after applying the permutation method

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Compression Speed (bytes/sec)	Decompression Speed (bytes/sec)
abbot.dat	350079	319954	1.09	6146221.05	8069918.17
bovary.dat	2210483	700468	3.16	1186935.26	53925340.33
corilis.dat	1266579	658391	1.92	4770034.12	12082490.56
cyprus.dat	558034	28401	19.65	159098.04	27880123.39
emission.dat	2507776	322424	7.78	906674.07	71714073.73
firewrks.dat	1445174	1312508	1.10	11052801.03	41333945.39
flower.dat	10327601	5048810	2.05	3799089.74	72339722.52
hungary.dat	3719443	182925	20.33	440557.41	112703905.89
lusiadas.dat	627712	181437	3.46	1565283.97	20925249.98
modern.dat	389933	150547	2.59	1359522.81	9240777.81
nightsht.dat	14809107	12595212	1.18	524828111.75	59093790.38
thunder.dat	3184336	2391045	1.33	9578347.80	57894689.21
ultima.dat	1077175	683434	1.58	5297842.18	25644078.60
venus.dat	13484366	9471787	1.42	6254342.79	75106908.85
w01vett.dat	1386261	97056	14.28	551177.91	39226057.54

Table 4.3: Performance of the Deflate Algorithm on Prague Corpus after applying the permutation method

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Compression Speed (bytes/sec)	Decompression Speed (bytes/sec)
abbot.dat	350079	318892	1.10	4759435.89	16670879.79
bovary.dat	2210483	681407	3.24	1340388.88	38019042.20
corilis.dat	1266579	648351	1.95	5075906.16	33936701.82
cyprus.dat	558034	27894	20.01	321733.78	16943912.07
emission.dat	2507776	315354	7.95	834343.47	78376600.43
firewrks.dat	1445174	1312551	1.10	10981243.75	43789997.90
flower.dat	10327601	5003668	2.06	3507222.07	68356922.45
hungary.dat	3719443	176515	21.07	409436.36	143067181.32
lusiadas.dat	627712	176212	3.56	1630271.17	27289843.61
modern.dat	389933	144772	2.69	1419330.04	17724359.43
nightsht.dat	14809107	12595212	1.18	787210885.03	64410117.41
thunder.dat	3184336	2386960	1.33	7440602.61	55201106.09
ultima.dat	1077175	679817	1.58	3862405.64	34750366.59
venus.dat	13484366	9464621	1.42	5725034.46	75077895.72
w01vett.dat	1386261	95157	14.57	464068.94	51343547.68

Table 4.4: Performance of the Deflate64 Algorithm on Prague Corpus after applying the permutation method

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Compression Speed (bytes/sec)	Decompression Speed (bytes/sec)
abbot.dat	350079	320697	1.09	3624426.29	7605800.16
bovary.dat	2210483	558095	3.96	2493260.53	22328277.59
corilis.dat	1266579	607711	2.08	2104368.43	16028969.86
cyprus.dat	558034	18504	30.16	86080.14	14386474.06
emission.dat	2507776	249322	10.06	917750.97	31696650.27
firewrks.dat	1445174	1357748	1.06	3633698.03	6429811.55
flower.dat	10327601	3885415	2.66	7049933.28	63122926.27
hungary.dat	3719443	106026	35.08	141804.95	18746019.18
lusiadas.dat	627712	172619	3.64	686853.66	8995725.46
modern.dat	389933	122708	3.18	786157.71	8110776.57
nightsht.dat	14809107	12595212	1.18	628509964.81	76570857.47
thunder.dat	3184336	2164681	1.47	10456166.86	30147516.21
ultima.dat	1077175	654194	1.65	2722011.70	13577840.92
venus.dat	13484366	8771290	1.54	12432978.44	39711733.48
w01vett.dat	1386261	73223	18.93	358958.52	22728836.58

Table 4.5: Performance of the BZip2 Algorithm on Prague Corpus after applying the permutation method

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Compression Speed (bytes/sec)	Decompression Speed (bytes/sec)
abbot.dat	350079	320876	1.09	1849149.23	1713626.20
bovary.dat	2210483	508741	4.35	1902137.83	6737439.52
corilis.dat	1266579	579391	2.19	1445681.82	3207861.40
cyprus.dat	558034	28824	19.36	586631.88	12410663.38
emission.dat	2507776	227731	11.01	1232859.95	13996264.75
firewrks.dat	1445174	1357069	1.06	1670942.38	1748829.52
flower.dat	10327601	3957146	2.61	2354849.43	5392711.85
hungary.dat	3719443	170599	21.80	1609150.27	31541471.03
lusiadas.dat	627712	165041	3.80	1366797.03	6406203.12
modern.dat	389933	109549	3.56	1422068.81	5344501.22
nightsht.dat	14809107	12595212	1.18	629386056.91	61488071.14
thunder.dat	3184336	2139136	1.49	1757754.75	2662025.86
ultima.dat	1077175	649818	1.66	2212232.21	3367625.33
venus.dat	13484366	8470306	1.59	2452514.27	3524210.45
w01vett.dat	1386261	82432	16.82	1081964.08	15387099.06

Table 4.6: Performance of the PPMd Algorithm on Prague Corpus after applying the permutation method

4.5 Comparative Analysis of Compression Algorithms on Prague Corpus

Compression Ratios

- **Highest Compression Ratio:**
  - *hungary.dat*: 35.08 (BZip2)
  - *cyprus.dat*: 30.16 (BZip2)
  - *w01vett.dat*: 18.93 (BZip2)

- **Lowest Compression Ratio:**
  - *firewrks.dat*: 1.06 (BZip2 and PPMd)
  - *nightsht.dat*: 1.18 (All algorithms)

Compression Speeds (bytes/sec)

- **Fastest Compression Speed:**
  - *nightsht.dat*: 787210885.03 (Deflate64)
  - *nightsht.dat*: 786648223.13 (LMZA)
  - *nightsht.dat*: 629386056.91 (PPMd)
- **Slowest Compression Speed:**
  - *cyprus.dat*: 86080.14 (BZip2)
  - *cyprus.dat*: 159098.04 (Deflate)
  - *cyprus.dat*: 141804.95 (BZip2)

Decompression Speeds (bytes/sec)

- **Fastest Decompression Speed:**
  - *hungary.dat*: 143067181.32 (Deflate64)
  - *hungary.dat*: 131172483.65 (LMZA)
  - *hungary.dat*: 112703905.89 (Deflate)
- **Slowest Decompression Speed:**
  - *abbot.dat*: 1713626.20 (PPMd)
  - *firewrks.dat*: 1748829.52 (PPMd)
  - *abbot.dat*: 16670879.79 (Deflate64)

Summary for Key Metrics Across Algorithms

- **LMZA:**
  - Good balance between compression ratio and speed.
  - Highest ratio for *hungary.dat* (28.01).
  - Very high decompression speed for many files.
- **Deflate:**
  - Generally fast compression and decompression speeds.
  - High decompression speed for *nightsht.dat* and other files.
- **Deflate64:**
  - Fastest compression speed for *nightsht.dat*.
  - High decompression speed overall.
- **BZip2:**
  - Highest compression ratios for many files, especially *hungary.dat* and *cyprus.dat*.
  - Generally slower compression speeds but still performs well in decompression.
- **PPMd:**
  - High compression ratios for several files.
  - Slower decompression speeds for some files, especially *abbot.dat*.

Overall Best Performers

- **Best Compression Ratios:**
  - BZip2 for highest ratios, particularly for *hungary.dat* and *cyprus.dat*.

- **Best Compression Speeds:**
  - Deflate64 for fastest speeds, particularly for *nightsht.dat*.
- **Best Decompression Speeds:**
  - LMZA and Deflate64 for overall high decompression speeds.

Conclusion

Each algorithm has its strengths:

- **BZip2** excels in achieving high compression ratios.
- **Deflate64** offers the fastest compression speeds.
- **LMZA** and **Deflate** provide a good balance with high decompression speeds.

Conclusion General

In our practical investigation, we delved into the nuanced trade-offs among various lossless compression algorithms: BZIP, Deflate, Deflate64, LEMA, and PPMD, using the Prague Corpus dataset as our benchmark. Through meticulous analysis across diverse file types, we unearthed distinct strengths among compression algorithms, some excelling in compression and decompression efficiency, while others prioritizing speed. This underscores the importance of tailoring compression algorithm selection to the specific file type and application requirements for optimal performance.

Furthermore, we explored the application of block permutation techniques to the Silesia dataset, coupled with training a predictive model to discern the necessity of such permutation operations. Subsequently, we juxtaposed this approach against conventional compression methods. However, our findings revealed a gap in performance, attributed to the limitations of model training on a relatively modest dataset. This highlights the potential for future research to address this shortfall by leveraging larger and more diverse datasets for model refinement and improved predictive accuracy.

Bibliography

[1] S. Pigeon. "Contribution to Data Compression". PhD thesis. Université de Montréal, 2001.

[2] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the Institute of Radio Engineers* 40 (Sept. 1952).

[3] Claude E. Shannon and Warren Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, 1949.

[4] C. E. Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27 (July 1948).

[5] Glassdoor. *Machine Learning Engineer Salaries*. https://www.glassdoor.com/Salaries/machine-learning-engineer-salary-SRCH_KO0,25.htm. Accessed March 20, 2024. Unknown.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science Business Media, 2009.

[7] V. Beaudoin. "Development of New Lossless Data Compression Techniques". PhD thesis. Université du Québec, 2008.

[8] Jeffrey S. Vitter. "Design and analysis of dynamic Huffman codes". In: *Journal of the ACM (JACM)* 34.4 (1987), pp. 825–845.

[9] Jorma Rissanen. "Generalized Kraft inequality and arithmetic coding". In: *IBM Journal of Research and Development* 20.3 (1976), pp. 198–203.

[10] Jon Louis Bentley. "Programming pearls: Writing correct programs". In: *Communications of the ACM* 29.5 (1986), pp. 364–367.

[11] D. W. Michael Burrows. *A Block-Sorting Lossless Data Compression Algorithm*. Tech. rep. 124. Digital Systems Research Center, 1994.

[12] B. A. Fouty and Gary Fouty. "A Mathematical Model for Estimating the Effectiveness of Bigram Coding". In: *Information Processing and Management* 12 (1976), pp. 111–116.

[13] J. Ziv and A. Lempel. *A universal algorithm for sequential data compression*. 1977. URL: https://doi.org/10.1109/TIT.1977.1055714.

[14] J. Ziv and A. Lempel. "Compression of Individual Sequences via Variable Rate Coding". In: *IEEE Transactions on Information Theory* (1978).

[15] Sheng Chen, Yang Shu, and Hesheng Wang. "Fast lossless compression of scientific floating-point data". In: *International Conference on Computational Science*. Springer. 2006, pp. 927–934.

[16] Wes McKinney. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2017. URL: https://www.oreilly.com/library/view/python-for-data/9781491957653/.