Exploring the Performance of Data Compression Algorithms with Varying Data Sortedness

Student Name

ABSTRACT

Data compression is a critical component of data transmission. This article introduces classic data compression algorithms, including Huffman encoding, arithmetic encoding, run-length encoding, and Lempel-Ziv, and categorizes and compares them based on input formats.
The implemented data compression algorithms in C++ are tested for compression ratio, compression time, and other metrics on different input data and types. The experimental results are compared to evaluate the performance of different compression algorithms on various input data.
By analyzing the experimental findings, the strengths and weaknesses of each algorithm are identified, and the applicable scenarios for different compression algorithms are summarized.

Keywords: data compression algorithms, comparession performance, data sortedness

1 INTRODUCTION

1.1 Motivation

Data compression is an important technology with wide-ranging applications in data transmission and storage. With the rapid development of the Internet and the advent of the big data era, the volume of data continues to increase, making efficient compression and transmission of data particularly important.
Data compression reduces storage space and transmission bandwidth, improves data transmission efficiency, and lowers storage and transmission costs. Therefore, data compression techniques have significant research value and practical applications in the fields of computer science and communication.
There are many algorithms for data compression, each with its own applicable scenarios and advantages. Therefore, this article aims to classify and compare data compression algorithms and explore and analyze their performance to help readers choose the appropriate algorithm for practical applications.

1.2 Problem Statement

Data compression is crucial, but it is often challenging to determine which compression algorithm is most suitable for different data types, sizes, and sortedness. There are various classification methods for data compression algorithms, such as lossless compression and lossy compression based on the compression approach, and different types based on the input data.
Due to differences in principles and implementation approaches, different compression algorithms have distinct application scenarios. Moreover, even with the same compression algorithm, the compression ratio, compression time, and other performance metrics may vary depending on factors like data sortedness and size. Therefore, to address the performance of different algorithms on input data with varying sortedness and sizes, this article will analyze and experiment with different compression algorithms, comparing metrics such as compression ratio and compression time.
Through the conclusions drawn from the experiments, we can gain a deeper understanding of the strengths and weaknesses of each algorithm and provide recommendations for selecting compression algorithms.

1.3 Contributions

The main contributions of this article are as follows:
1. In the past, the main classification of compression algorithms was based on lossy or lossless compression. This article classifies compression algorithms from the perspective of different input data types and compares them.
2. This article implements different compression algorithms on input data with varying degrees of sortedness and compares the compression ratios of the algorithms on input data with different degrees of sortedness, filling the research gap in the sortedness of input data left by previous studies.

2 BACKGROUND

One of the classification criteria for compression algorithms is based on the compression method, which can be divided into lossless compression and lossy compression. Lossless compression refers to the compression where the data before and after compression are exactly the same, without losing any information. Lossy compression, on the other hand, means that the compressed data differs from the original data and loses some information.
In 2010, S.R. KODITUWAKKU conducted a study on lossless compression algorithms for text data and tested their performance.[3] This provides us with some insights into how to perform performance testing for compression algorithms.
In 2007, M Al-Laham et al. compared the performance of various compression algorithms, including HUFFMAN, LZW, LZ77, and others[1]. In 2012, M Hosseini surveyed multiple compression algorithms such as Huffman, Run-Length Coding, Lempel-Ziv, and Arithmetic Coding and also divided the algorithms into lossless and lossy compression[2].
In previous work, the performance of many algorithms was evaluated on specific datasets without considering the sortedness. Therefore, in this article, we will conduct experiments considering different aspects such as the degree of sorting in the input datasets to compare the performance of different algorithms.

3 CLASSIFICATION OF COMPRESSION ALGORITHMS

3.1 Classification Based on Input Data

Compression algorithms can be classified based on the input data types. Here we introduce three types of input data: numerical, strings, and byte.

- Numerical data: This type of data is composed of numbers, such as integers and floating-point numbers, and is often used in scientific computing and engineering.
- String data: This type of data is composed of characters, such as letters, numbers, and symbols, and is often used in text data.
- Byte data: This type of data is composed of bytes, which is the smallest unit of data in a computer, and is often used in image, audio, video and other types of data.

We will classify the compression algorithms based on the input data types and give a brief introduction to each type of algorithm.

HUFFMAN Encoding. Huffman encoding is usually used for **string data**.It is a lossless compression algorithm that constructs a Huffman tree to represent characters with higher frequencies using shorter codes and characters with lower frequencies using longer codes, thereby achieving compression.
In the code, we use a priority queue data structure to construct the Huffman tree. The process of constructing the Huffman tree is as follows:

- Count the frequency of each character in the input string.
- Construct a priority queue based on the frequency of each character.
- Pop the two nodes with the lowest frequency from the priority queue and combine them into a new node. Then push the new node back into the priority queue.
- Repeat the above step until there is only one node left in the priority queue, which is the root of the Huffman tree.
- Traverse the Huffman tree to generate the Huffman code for each character, the left child node is 0, and the right child node is 1.

Here is an example of Huffman encoding:
The input string data is **aababcabab**,and the frequency and code of each character is as follows:

	frequency	code
a	5	0
b	4	10
c	1	11

So, the compressed data is **001001011010010** in bits, and the compression ratio is 23.4375%.

According to the principle of Huffman coding, we can infer that because the number of distinct characters in string data is limited, the number of leaf nodes in the Huffman tree is also limited.
Therefore, the depth of the tree is not very deep, and the length of each character's encoding is not very long. As a result, Huffman coding is suitable for compressing string data.

Howerver, Huffman encoding also has some limitations, it cannot approach Shannon's entropy limit. According to Shannon's entropy calculation formula:

$$H(X) = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i) \tag{1}$$

the entropy of the input string data is:

$$H(X) = -\sum_{i=1}^{3} p(x_i) log_2 p(x_i) = 1.361 \tag{2}$$

In the example, we averagely use 1.5 bits to encode each character, which is not very close to the entropy limit.

The reason is that Huffman uses integers for symbol encoding, which is not precise enough. For example, the probabilities of c and d are different, but he uses the same length of encoding.

RunLength Encoding. RunLength encoding is usually used for **byte data and text data**. It is a lossless compression algorithm that replaces repeated data with a count and a single data value.
The process of RunLength encoding is as follows:

- Scan the input data from left to right.
- If the current data is the same as the previous data, increase the count by 1.
- If the current data is different from the previous data, output the count and the previous data, and then reset the count to 1.
- Repeat the above steps until the end of the input data.

Here is an example of RunLength encoding:
The input data is **aaabbbcc**, and the RunLength encoding is **3a3b2c**.
RunLength encoding is suitable for compressing byte data and text data with a high degree of sorting. The more repeated data, the higher the compression ratio.

Howerver, if the input data is unsorted, the compression ratio will be low, even worse than the original data. For example, the input data is **abcabcabc**, and the RunLength encoding is **1a1b1c1a1b1c1a1b1c**, which is longer than the original data.

Lempel-Ziv Encoding.

REFERENCES

[1] Mohammed Al-Laham and Ibrahiem MM El Emary. 2007. Comparative study between various algorithms of data compression techniques. *IJCSNS* 7, 4 (2007), 281.
[2] Mohammad Hosseini. 2012. A survey of data compression algorithms and their applications. *Network Systems Laboratory, School of Computing Science, Simon Fraser University, BC, Canada* (2012).
[3] SR Kodituwakku, US Amarasinghe, et al. 2010. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering* 1, 4 (2010), 416–425.