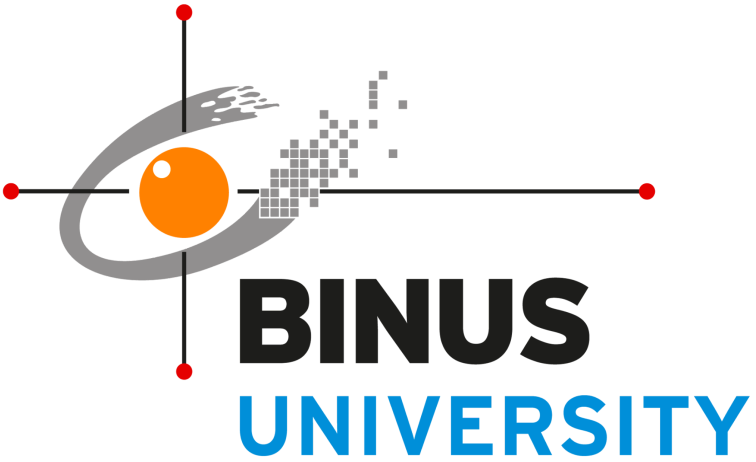
PROJECT REPORT

Comparative Study of Data Compression Algorithms

Algorithm Design and Analysis



Arranged by:

ALYSHA PUTI MAULIDINA (2502005906)

KIMBERLY MAZEL (2502022250)

RACHEL ANASTASIA WIJAYA (2502009646)

CHELLSHE LOVE SIMROCHELLE (2502043040)

Faculty of Computing and Media

BINUS INTERNATIONAL UNIVERSITY

Table of Contents

Background

Data compression refers to the conversion of data into a more compact form of representation that has fewer storage requirements \cite{b1}. Fundamentally, the process involves eliminating and interpreting excessive or redundancy in data. The goal of data compression is to represent the data in fewer bits as possible while maintaining the minimum requirements of the original. For example, in a successful text compression, unnecessary characters are removed and replaced with a single character as a reference for a string of repeated characters. With the right function, compression algorithms can effectively reduce the size of a text file by 50% or more, significantly lower than its original size \cite{b2}.

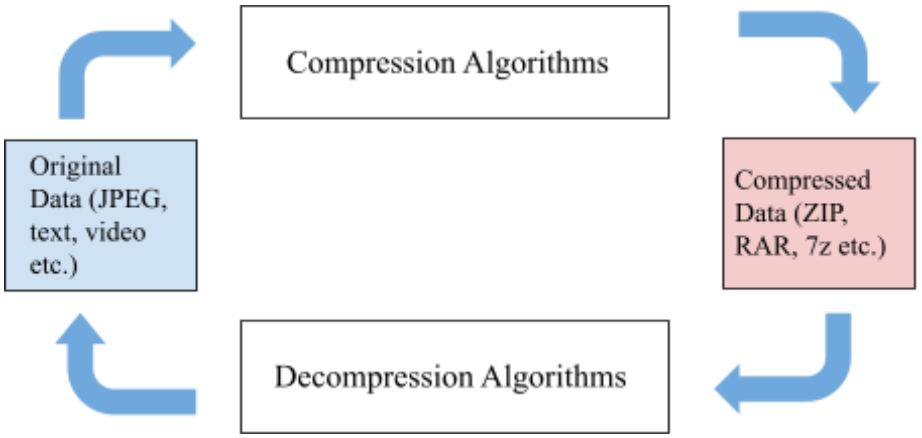


Fig. 1: Pictorial framework of data compression

In the era of the digital multimedia revolution and rapid communication, where huge amounts of data are generated each day, data compression algorithms have been critical in enabling the ease of data transmission. There are many data compression software available to compress files of different formats, with each vendor providing different methods to optimize compression and offering different trade-offs. This research aims to focus on comparing the performance of three different compression algorithms, examining each strength, best-case uses, and efficiency. Unlike previous studies, which have focused on one specific algorithm, this study provides a comprehensive comparison of these algorithms and evaluates their performance on a variety of data types.

Problem Analysis

The main problem that this paper aims to address is the lack of a comprehensive comparison of different data compression algorithms. While each algorithm has its own unique characteristics, it is difficult to determine which algorithm is the most effective overall. Additionally, many papers only focus on one or a few specific algorithms, making it difficult to compare and contrast different algorithms in a single study.

To address this problem, we will conduct a thorough comparison and analysis of different data compression algorithms. We will compare the most popular algorithms, namely Zstd, zlib, and LZ4, and evaluate them in terms of compression ratio, compression and decompression speed, and memory usage. We will use a variety of data sets, including text, images, and audio, to ensure that our results are representative of a wide range of data types.

Zstandard, also known as zstd, is a fast lossless compression algorithm that targets real-time compression scenarios at the zlib-level and better compression ratios. According to the developers of Facebook, Zstandard provides high compression ratios as well as great compress and decompression speeds and allegedly offers the best-in-kind performance in many conventional situations [3]. However, there are two alternatives to Zstandard, namely zlib and LZ4.

Similarly to Zstandard, zlib and LZ4 claim to offer the same benefits of high compression ratios and speeds. Although these algorithms have the same purpose, no two algorithms are identical. Each algorithm will have its strengths and weaknesses, and this project aims to measure and compare them to find out what they are.

Measurements/benchmark

There are 3 standard metrics for comparing data compression algorithms, which are as follows:

- **Compression ratio**
  The original size of the data compared with the compressed size. Measured in unit-less data as a size ratio of 1.0 or greater

- **Compression speed**
  The rate at which we can make the data smaller. Measured in MB/s of input data consumed.

- **Decompression speed**
  The rate at which we can reconstruct the original data from the compressed data. Measured in MB/s for how quickly data is produced from compressed data.

However, we also decided to add one metric of our own for additional analytical purposes:

- **Space complexity**
  The amount of space/memory taken up by an algorithm to run input to completion. Measured in Big-O notations.

The reason we chose these metrics is that we found that using the three standard metrics of compression ratio, compression speed, and decompression speed is standard practice when comparing data compression algorithms. Alongside the addition of space complexity, we can use the collected information to decide when to use specific algorithms. For example, one algorithm might be able to compress data very quickly, but at the cost of taking up a large amount of space.

Methodology

To properly evaluate the different algorithms, certain variables have to be controlled to ensure that the measured variables are only affected by the algorithms alone and not external factors. For example, the test data and device should be the same for each algorithm. Since all three algorithms are open-sourced, the code can be easily found online.

The respective algorithms can be found in the links below:

a) Zstandard - https://github.com/facebook/zstd
b) zlib https://github.com/madler/zlib
c) LZ4 - https://github.com/lz4/lz4/

A corpus, or dummy data, is used to test these algorithms. This research has used the Silesia corpus which is a set of files of different characteristics to test how the aforementioned algorithms perform on common data types used every day. The data sets vary in size as this will allow the algorithms to be tested on both small and large data sets. The list of data types is listed as follows.

1. Text (.txt)
2. PDF
3. CSV
4. JSON
5. JPG (small and big).

The corpus is then used to test each of the algorithms. To improve accuracy, the testing for each algorithm will be done a set number of times, and the average of each algorithm will be used instead of the pure value for one trial. The results will be documented in tables and represented in various graphs to be able to compare the difference more clearly.

After the comparison, we can properly conclude each algorithm's strengths and weaknesses and present our results.

Theories/concepts of the algorithm

In theory, data compression algorithms work by reducing the redundancy of data, thereby allowing it to be stored or transmitted more efficiently. Zstd, zlib, and LZ4 use different theories and techniques to compress data.

A common approach used by compression algorithms is to identify and remove redundant patterns or sequences of data. For instance, the Zstandard algorithm uses a combination of techniques, including dictionary compression and entropy encoding, to achieve high levels of compression. Dictionary compression is a lossless technique that works by creating a dictionary of commonly occurring patterns in the data and replacing these patterns with shorter references to the dictionary, resulting in a smaller representation of the data [4]. This method is particularly effective for types of data that contain repetitive patterns, such as natural language text or source code. Dictionary compression algorithms are typically faster than other compression techniques; however, research by Lasch et. al. revealed that such dictionaries consume a significant amount of memory due to the need to store the dictionary [5]. Dictionary-based algorithms, such as LZ4, are often used in combination with other compression techniques, such as Entropy coding, to further improve the compression ratio. Zstd is designed to be highly versatile and can be used in a variety of applications, such as file compression, network communication, and data storage.

Entropy coding is a technique that uses statistical properties of the data to encode it more efficiently. It adopts the theory that symbols that occur more frequently on the data should be assigned shorter code words than symbols that occur less frequently. This is based on the principles of entropy, which states that the amount of information in a message is inversely proportional to the probability of each symbol [6]. Huffman coding and arithmetic coding are typical algorithms of this kind.

zlib, in particular, is a data compression library that uses the DEFLATE algorithm – a combination of the LZ4 algorithm and Huffman encoding [7]. It works by using a sliding window technique that replaces the repeated sequences of data with a reference to the original data. The following figure demonstrates an example of a sliding window, where the search buffer contains the dictionary (recently encoded data) and the look-ahead buffer contains the next portion of the input data sequence to be encoded [7]. zlib is widely used in many applications, such as the PNG image format and HTTP compression.

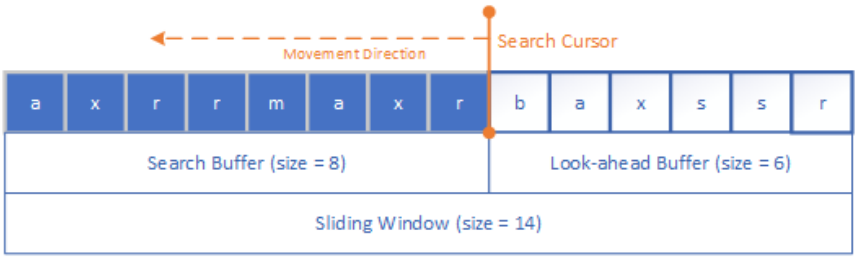


Fig. 2: Sliding window

LZ4 uses a dictionary-matching scheme like the LZ4 byte-oriented compression algorithm [8]. LZ4 has been found to be very fast both in compression and decompression and is often used in applications such as data backup, data archiving, and in-memory data storage.

Planning

The project planning consists of the following steps.

1. Getting test data
2. Testing each algorithm
3. Documenting the results
4. Comparing each algorithm
5. Using data visualization to represent the comparison
6. Complete writing the report
7. Presenting our results

The project schedule is represented by the Gantt chart below.

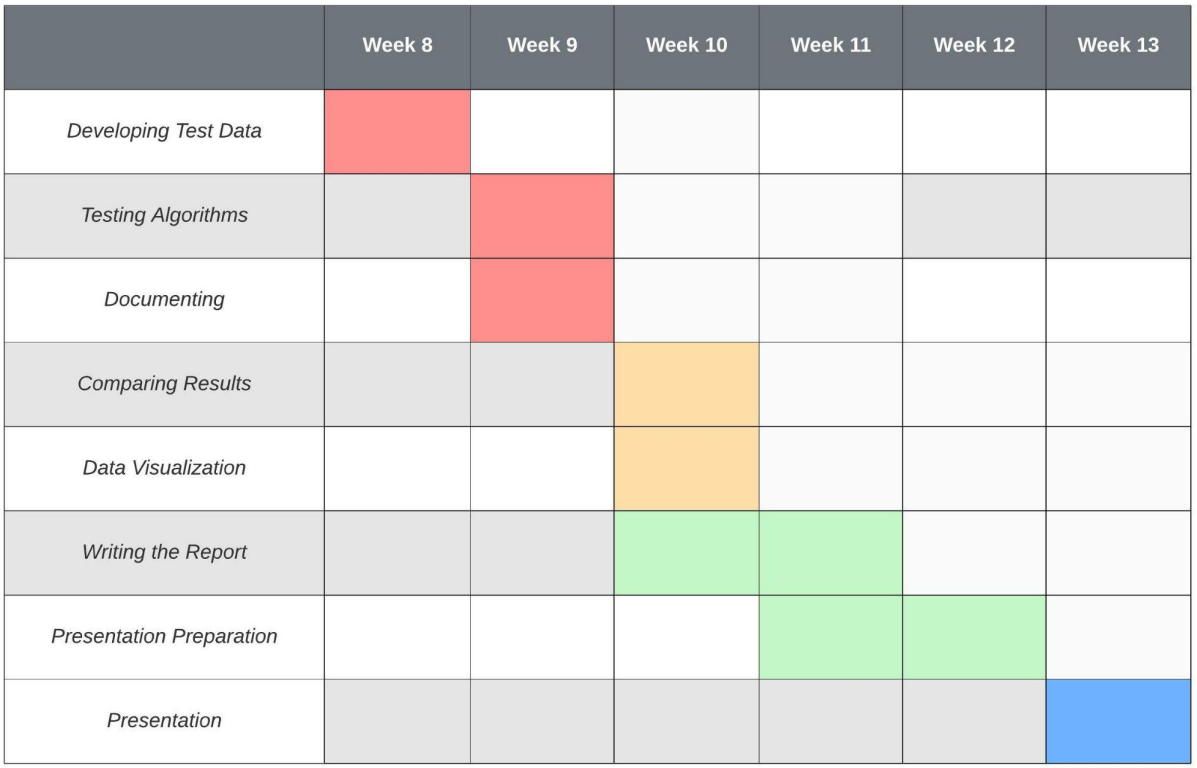


Fig. 3: Gantt chart of the project plan

Findings

The tables below show the average figures of the different corpus tests on the three compression algorithms.

Table 1.1: Results of the test on the text file corpus

Text			
Measurement	Zstandard	zlib	LZ4
Compression Ratio	2.60	2.62	1.67
Size Savings (%)	61.60	61.78	40.20
Compression Speed	12.47	6.89	38.20
Decompression Speed	12.97	13.35	10.05
Space Complexity (c)	768.00	770.00	768.00
Space Complexity (d)	736.00	740.00	736.00

Table 1.2: Results of the test on the small image file corpus

Image (small)			
	Zstandard	zlib	LZ4
Compression Ratio	1.00	1.00	1.00
Size Savings (%)	-0.01	1.00	-0.01
Compression Speed	153.78	34.18	205.81
Decompression Speed	41.31	43.16	35.69
Space Complexity (c)	1,020.00	686.00	1,020.00
Space Complexity (d)	866.00	734.00	730.00

Table 1.3: Results of the test on the large image file corpus

Image (large)			
	Zstandard	zlib	LZ4
Compression Ratio	1.00	1.00	1.00
Size Savings (%)	0.00	0.01	0.00
Compression Speed	398.42	39.19	527.90
Decompression Speed	692.38	247.69	585.48
Space Complexity	1,052.00	718.00	1,052.00
Space Complexity (d)	746.00	750.00	746.00

Table 1.4: Results of the test on the PDF file corpus

PDF			
	Zstandard	zlib	LZ4
Compression Ratio	1.09	1.10	1.08
Size Savings (%)	8.26	9.48	7.11
Compression Speed	244.950082	33.75751841	512.77
Decompression Speed	239.8868444	159.8692066	229.91
Space Complexity	772.00	774.00	772.00
Space Complexity (d)	738.00	742.00	738.00

Table 1.5: Results of the test on the CSV file corpus

CSV			
	Zstandard	zlib	LZ4
Compression Ratio	2.52	2.43	1.54
Size Savings (%)	60.29	58.87	35.02
Compression Speed	50.84780732	7.87	176.57
Decompression Speed	222.6286971	141.78	271.71
Space Complexity	780.00	782.00	780.00
Space Complexity (d)	742.00	746.00	742.00

Table 1.6: Results of the test on the JSON file corpus

JSON			
	Zstandard	zlib	LZ4
Compression Ratio	3.38	3.71	2.54
Size Savings (%)	70.43	73.07	60.61
Compression Speed	0.20	0.24	0.36
Decompression Speed	0.41	0.75	0.60
Space Complexity	774.00	776.00	774.00
Space Complexity (d)	740.00	744.00	740.00

After compiling the average results of the compression algorithm testings for all the corpus data types, we created bar graphs to further analyze and compare the differences between each compression algorithm in each measurement.

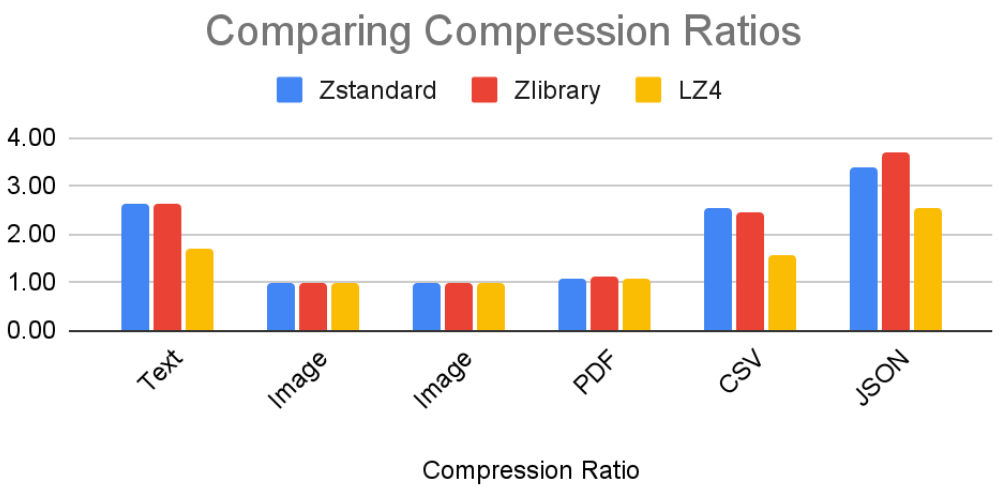


Fig. 4: Comparing Compression Ratios

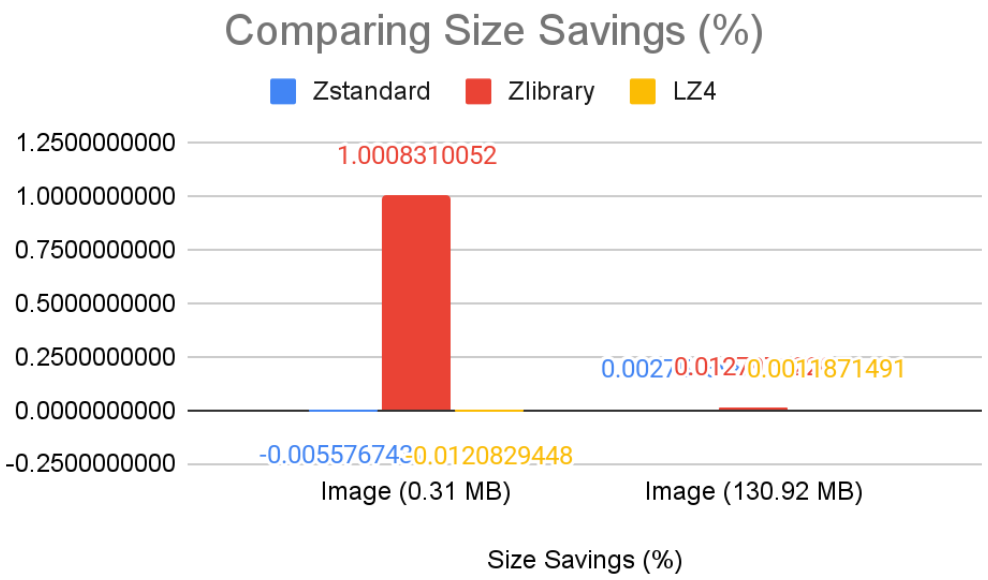


Fig. 5: Comparing Size Savings for Images

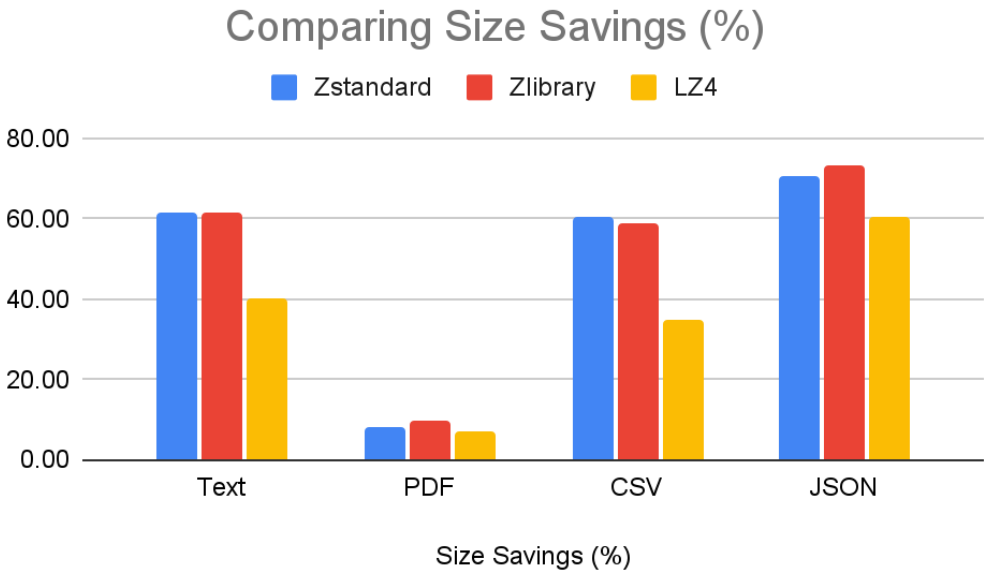


Fig. 6: Size Savings for Text, PDF, CSV, JSON files

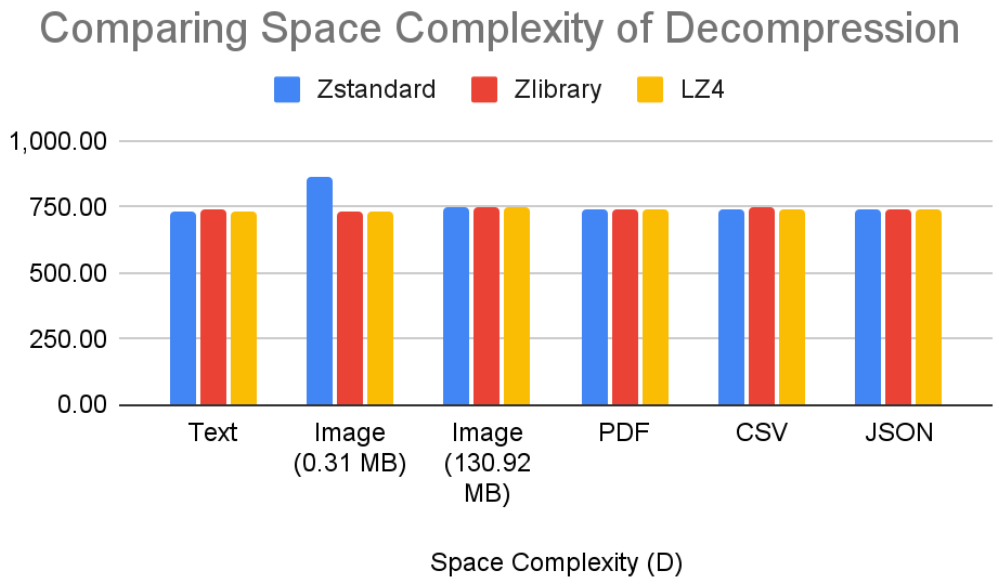


Fig. 7: Space Complexity for Decompression

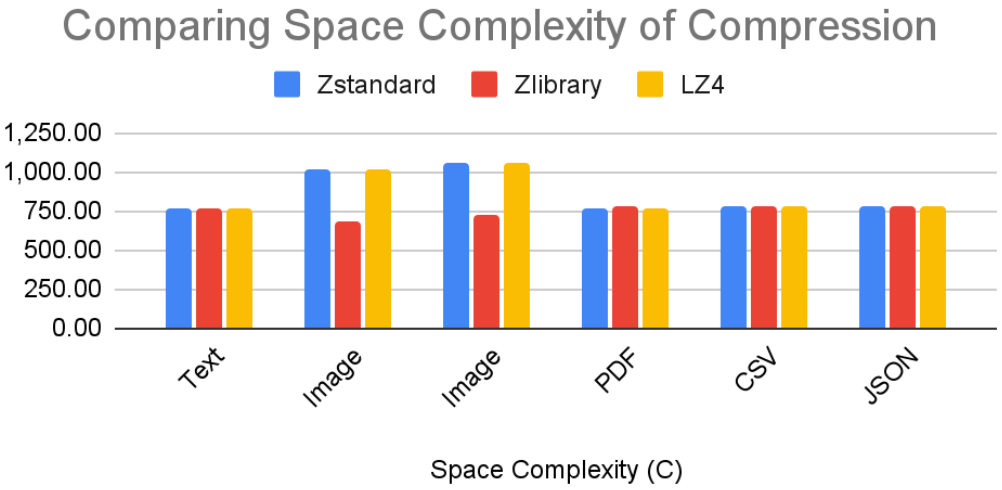


Fig. 8: Space Complexity for Compression

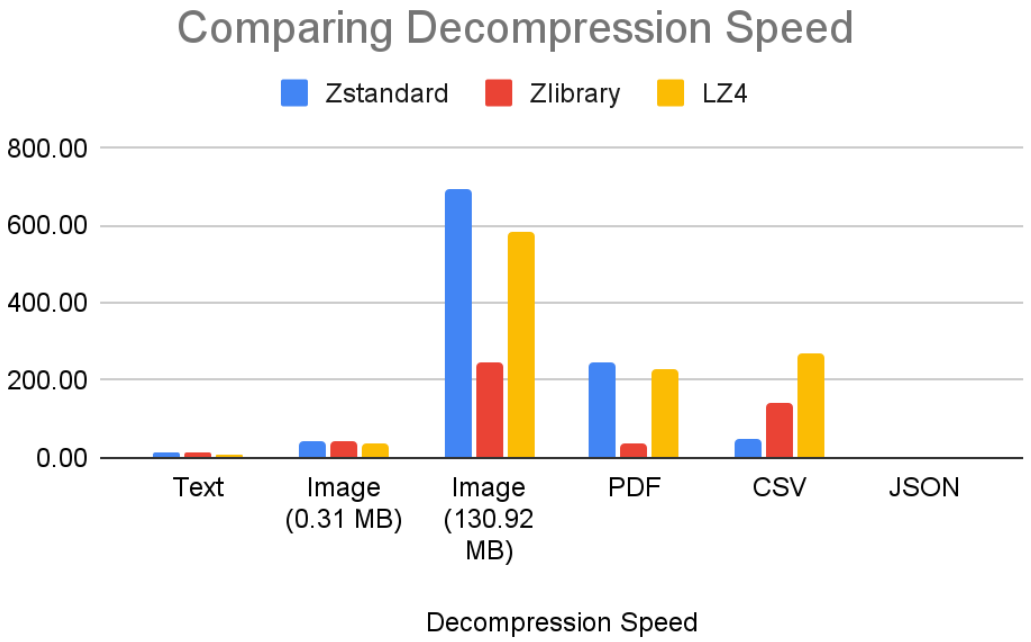


Fig. 9: Decompression Speed

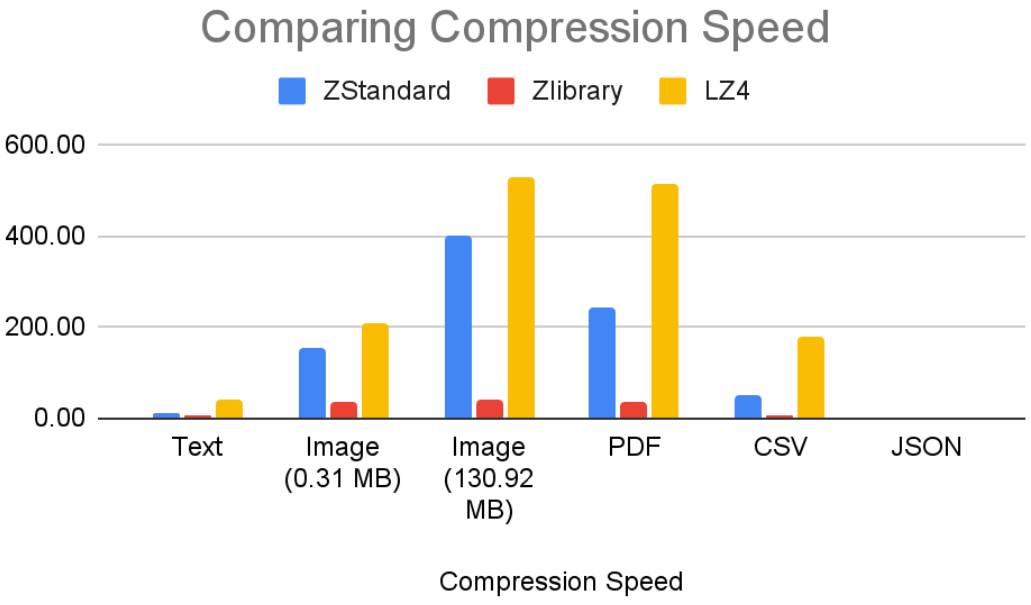


Fig. 10: Compression Speed

Discussion and Analysis

According to the findings, the file types that were compressed best were text-based and JSON files. As shown in Figure 4, the highest compression ratios were from JSON files, with text-based files falling second. This is further supported by Figure 6, which demonstrates the high values of size savings for both JSON and text-based files. However, the compression and decompression speeds of both file types are the lowest, as shown in Figure 9 and Figure 10. This may be the result of the relatively low size threshold of JSON and text-based files. The speeds are measured in MBps (megabytes per second), so the recorded speed values are low as there are not many megabytes to compress in the first place.

On the other hand, the worst compressed file types were PDFs and images, specifically JPEGs. It has the lowest compression ratios and size savings. They have higher compression and decompression speeds, but this may be attributed to their larger file sizes. Interestingly, the smaller-sized images have negative values of size savings. JPEGs are already a compressed file type. As a result, attempting to compress the size again did not lower the original size of the file. Instead, the size became larger due to the compression algorithms' overhead adding onto the original file size. This is why JPEGs were compressed poorly throughout the trials. The file sizes are already compressed to the point that no compression can decrease the file size further unless it is lossy.

Overall, zlib had the best performance for most data types. It had the best compression ratios and size savings except for CSV files, where they fell short of Zstandard. The LZ4 algorithm had the worst compression ratios and size savings out of the three, however, it had the best compression speeds. This is because LZ4 has a small overhead and relatively simple decoding. It provides high compression speeds at the cost of low compression ratios. As for decompression speeds, zlib had the best performance except for larger images and PDFs, where Zstandard took the lead. For space complexities, LZ4 and Zstandard have identical values across all data types. Interestingly, both algorithms share the same creator. zlib only had the advantage of the space complexity of compressing images and decompressing JSON files.

Conclusion and recommendation

This report presents a comparison between three lossless data compression algorithms, Zstandard, zlib, and LZ4; carrying out analyses of five different data types: text, images, PDF, CSV, and JSON. This research provided deeper insight into the field of compression algorithms. Based on our findings, we can conclude that while among the three compression algorithms, Zstandard has a higher encoding speed with consistent compression ratios, it being a recently released algorithm (stable release in 2022) with a lack of open sources can make it less reliable in general use cases compared to zlib, which uses the universally adopted DEFLATE algorithm and has been developed and improved upon since 1995. On the other hand, we can also conclude that LZ4 had the worst compression ratio in comparison to zlib and Zstandard, but had the fastest decoding speeds by a significant margin due to its low overhead and simple decoder. Thus, the LZ4 algorithm is best suited and often one of the only viable options for processes that involve small microprocessors or microcontrollers. In future research, a larger variety of data types and sizes can be used to improve generalizability. More benchmarks can also be used to further determine the usability for different cases and observe whether the current findings are still valid. There are still many different options of compression algorithms and techniques that are not covered within this study, so it would be beneficial to conduct further research to compare their performances with the current algorithms.

References

A. Anup, R. Ashok, and P. Raundale, “Comparative study of data compression techniques,” *International Journal of Computer Applications*, vol. 178, no. 28, pp. 15–19, 2019.

C. T. Yann Collet, “Smaller and faster data compression with Zstandard,” *Engineering at Meta*, 28-Jun-2018. [Online]. Available: https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/. [Accessed: 15-Jan-2023].

“Compression techniques,” *Isaac Computer Science*, 2020. [Online]. Available: https://isaaccomputerscience.org/concepts/data_compr_loss?examBoard=all&stage=all. [Accessed: 15-Jan-2023].

“Data compression,” *Barracuda Networks*, 21-Oct-2022. [Online]. Available: https://www.barracuda.com/glossary/data-compression. [Accessed: 15-Jan-2023].

E. Chen, *Understanding zlib*, 2019. [Online]. Available: https://www.euccas.me/zlib/. [Accessed: 15-Jan-2023].

lz4, “LZ4/lz4_block_format.MD at dev · LZ4/LZ4,” *GitHub*. [Online]. Available: https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md. [Accessed: 15-Jan-2023].

R. Lasch, I. Oukid, R. Dementiev, N. May, S. S. Demirsoy, and K.-U. Sattler, “Faster & Strong: String Dictionary compression using sampling and fast vectorized decompression,” *SpringerLink*, 20-Jul-2020. [Online]. Available: https://link.springer.com/article/10.1007/s00778-020-00620-x. [Accessed: 15-Jan-2023].

Z.-M. Lu and S.-Z. Guo, *Lossless information: Hiding in images*. Amsterdam, Netherlands: Zhejiang University Press, 2017.

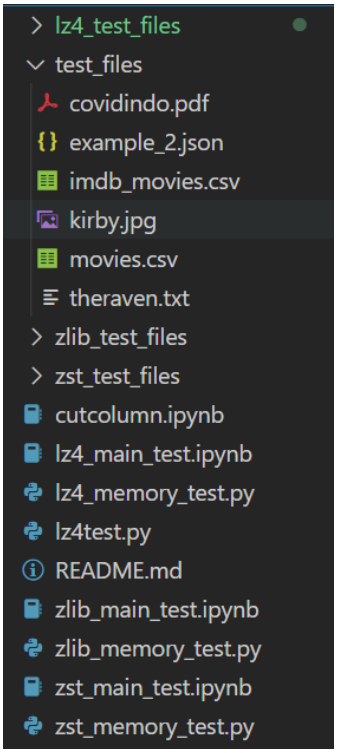
Annex

Annex 1 - Link to GitHub

1. GitHub repository: https://github.com/gloxiinia/ADA_FinalProject
2. Google Sheets for full results: ADaA FP Results
3. PPT file: ADA FP Presentation.pdf
4. Drive of all documents for this project: ADA Final Project Documents

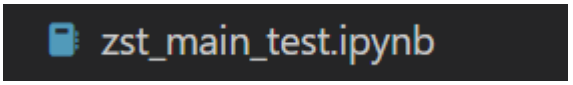
Annex 2: User Manual (Zstandard Example)

1. Download the code from the GitHub repository. Check that all the needed files are present.



   Note that for the large image file, you will need to download it from our Google Drive link, since GitHub does not support uploading files larger than 100MB.

2. To run the main tests (compression speed and size savings), open the Jupyter Notebook file containing the compression algorithm you'd like to test. In this example we'll be using the Zstandard algorithm.



3. In the Jupyter Notebook file, import the needed libraries, which are the compression algorithm's (in this case zstandard), time, and os.

```
#importing the libraries used

import zstandard as zst
import time #for runtime comparison
import os
```

4. While there are 6 different data types to run the test on, the process for doing the tests are similar across each data type. For this example, we will use the small image data type. To start the test, run the code to compress the file. This code will calculate the time taken for the function to open and compress the file.

```
1. IMAGE FILES

Compression

#getting the start time
com_st = time.time()

with open('test_files/kirby.jpg', 'rb') as infile:
    with open('zst_test_files/kirby.zst', 'wb') as outfile:
        outfile.write(zst.compress(infile.read()))


#getting the end time and elapsed time
com_et = time.time()
com_elapsed_time = com_et - com_st
[90]
```

5. Next, run the code to decompress the file. This code will calculate the time taken for the function to open the compressed file and decompress it.

```
Decompression

#getting the start time
decom_st = time.time()

with open('zst_test_files/theraven.zst', 'rb') as infile:
    with open('zst_test_files/theravenAfterzst.txt', 'wb') as outfile:
        outfile.write(zst.decompress(infile.read()))

#getting the end time and elapsed time
decom_et = time.time()
decom_elapsed_time = decom_et - decom_st
```

6. After that, we run the code block to print the compression and decompression times.

a. Runtime comparison

```
print('Compression execution time:', "{:.10f}".format(com_elapsed_time), 'seconds')
print('Decompression execution time:', "{:.10f}".format(decom_elapsed_time), 'seconds')
```

```
Compression execution time: 0.0020008087 seconds
Decompression execution time: 0.0049998760 seconds
```

7. Finally, we print the comparisons for the original file size, compressed file size, and also the decompressed file size, to check for any losses.

b. File size comparison:

```
#file size comparison
#original
file_stats = os.stat("test_files/theraven.txt")
print("Original file:")
print(f'File Size in Bytes is {file_stats.st_size}')
print(f'File Size in MegaBytes is {file_stats.st_size / (1024 * 1024)}\n')

#compressed
file_stats = os.stat("zst_test_files/theraven.zst")
print("Compressed file:")
print(f'file size in Bytes is {file_stats.st_size}')
print(f'file size in MegaBytes is {file_stats.st_size / (1024 * 1024)}\n')

#decompressed
file_stats = os.stat("zst_test_files/theravenAfterzst.txt")
print("Decompressed file:")
print(f'file size in Bytes is {file_stats.st_size}')
print(f'file size in MegaBytes is {file_stats.st_size / (1024 * 1024)}\n')
```

```
Original file:
File Size in Bytes is 66394
File Size in MegaBytes is 0.06331825256347656

Compressed file:
file size in Bytes is 25493
file size in MegaBytes is 0.02431201934814453

Decompressed file:
file size in Bytes is 66394
file size in MegaBytes is 0.06331825256347656
```

8. Now we can move on to the memory test. To do it, we open the corresponding memory test python file for the algorithm.

zst_memory_test.py

9. First we import the needed libraries, which are tracemalloc and the compression algorithm being tested.

```
import tracemalloc
import zstandard as zstd
```

10. Next, we run the tests for all 6 data types. We call the tracemalloc.start() function, then run the code for compression or decompression, and then take a snapshot by calling the tracemalloc.take_snapshot() function. Then after each test we call the tracemalloc.cleartraces() to clean the traces from the previous test before the next.

```
tracemalloc.start()

with open('test_files/kirby.jpg', 'rb') as infile:
    with open('zst_test_files/kirby.zst', 'wb') as outfile:
        outfile.write(zstd.compress(infile.read()))

img_com_snapshot = tracemalloc.take_snapshot()

print("================ IMG COMPRESSION SNAPSHOT ================")
for stat in img_com_snapshot.statistics("lineno"):
    print(stat)

tracemalloc.clear_traces()

with open('zst_test_files/kirby.zst', 'rb') as infile:
    with open('zst_test_files/kirbyAfterzst.jpg', 'wb') as outfile:
        outfile.write(zstd.decompress(infile.read()))

img_decom_snapshot = tracemalloc.take_snapshot()

print("\n================ IMG DECOMPRESSION SNAPSHOT ================")
for stat in img_decom_snapshot.statistics("lineno"):
    print(stat)

tracemalloc.clear_traces()
```

11. Results of the memory test for the small image file.

```
PS C:\Users\10\Documents\GitHub\ADA_FinalProject> python -u "c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py"
================ IMG COMPRESSION SNAPSHOT ================
c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py:7: size=514 B, count=5, average=103 B
c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py:6: size=506 B, count=5, average=101 B
C:\Users\10\AppData\Local\Programs\Python\Python310\lib\site-packages\zstandard\__init__.py:190: size=168 B, count=1, average=168 B

================ IMG DECOMPRESSION SNAPSHOT ================
c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py:19: size=434 B, count=5, average=87 B
c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py:20: size=432 B, count=1, average=432 B
c:\Users\10\Documents\GitHub\ADA_FinalProject\zst_memory_test.py:18: size=296 B, count=3, average=99 B
```