Report on

Data Compression

By Prem Kumar Yadav

Project Link: https://www.chameleoncloud.org/experiment/share/d1abff43-6953-4e30-a5f6-cd68c832cc2d

INTRODUCTION

The exponential rise of data in the digital age has shown how crucial effective data transport and storage are. Data compression techniques play a critical role in addressing this issue by lowering data size while maintaining its integrity. This paper explores about two widely used algorithms that are Huffman Coding and DEFLATE.
A key approach in data compression, Huffman coding uses variable-length codes to reduce the average number of bits needed to represent each symbol in a collection. It is a standard for comprehending the fundamentals of entropy coding due to its ease of use and efficiency. However, DEFLATE is a more advanced technique, (Combination of Lempel-Ziv-Storer-Szymanski (LZ77) algorithm with Huffman coding.) which achieves greater compression ratios, especially in datasets with repetitive patterns.
This study examines how well these algorithms function in various scenarios, paying particular attention to how they behave while dealing with both structured and randomized text material. We evaluate the strengths and limitations of each method, highlighting DEFLATE's efficiency in compressing repetitive data and Huffman coding's adaptability to randomized content. We aim to provide insights into the practical applications of these algorithms and their role in optimizing data handling and storage.

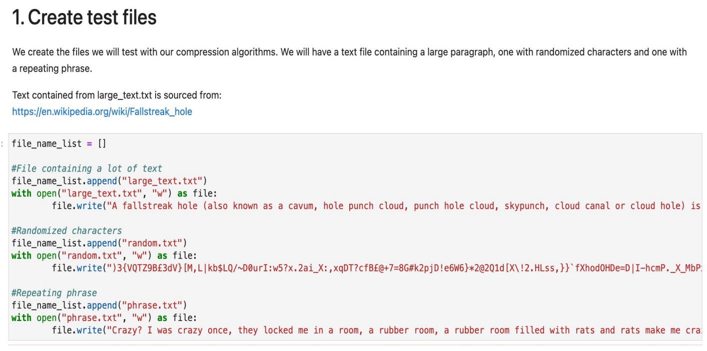
Basics of Data Compression

Data Compression is the technique to reduce the size of data to save storage space, faster transmission and optimize resource usage. Data Compression are of two types:
The technique of reducing a file's size (into Fourier or wavelet transformations) by eliminating some irrelevant or less significant data or components is known as **lossy compression.**
Lossless compression refers to reducing the file size without losing any data. Example: Run-Length Encoding, Huffman Coding, DEFLATE etc.

Compression ratio $=\frac{Uncompressed\ Size}{Compressed\ Size}$ (Higher ratio means better compression)

Understanding and running the Project:

1. First we create three files with our sample text



1. Create test files

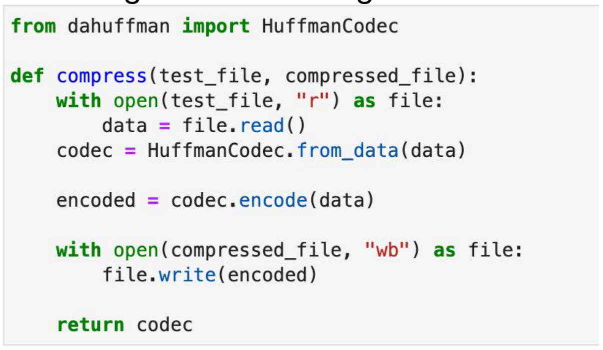
We create the files we will test with our compression algorithms. We will have a text file containing a large paragraph, one with randomized characters and one with a repeating phrase.

Text contained from large_text.txt is sourced from:
https://en.wikipedia.org/wiki/Fallstreak_hole

```
file_name_list = []

#File containing a lot of text
file_name_list.append("large_text.txt")
with open("large_text.txt", "w") as file:
        file.write("A fallstreak hole (also known as a cavum, hole punch cloud, punch hole cloud, skypunch, cloud canal or cloud hole) is

#Randomized characters
file_name_list.append("random.txt")
with open("random.txt", "w") as file:
        file.write(")3{VQTZ9B£3dV}[M,L|kb$LQ/~D0urI:w5?x.2ai_X:,xqDT?cfB£@+7=8G#k2pjD!e6W6}*2@2Q1d[X\!2.HLss,}}`fXhodOHDe=D|I-hcmP._X_MbP

#Repeating phrase
file_name_list.append("phrase.txt")
with open("phrase.txt", "w") as file:
        file.write("Crazy? I was crazy once, they locked me in a room, a rubber room, a rubber room filled with rats and rats make me cra
```

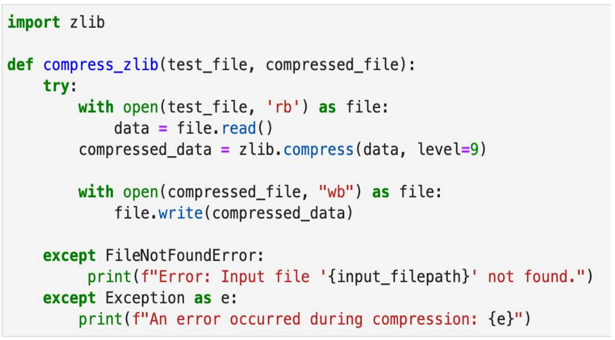
Here, three test files are created. the three types of sample document are large umber of text, repeated word text and random characters.

2. Testing huffman coding and DEFLATE



```python
from dahuffman import HuffmanCodec

def compress(test_file, compressed_file):
    with open(test_file, "r") as file:
        data = file.read()
    codec = HuffmanCodec.from_data(data)

    encoded = codec.encode(data)

    with open(compressed_file, "wb") as file:
        file.write(encoded)

    return codec
```

We use the HuffmanCodec from the imported library dahuffman. We will be using HuffmanCodec.from_data(data) to encode our files from plaintext to compressed data. The function defined below will be used to compress all of our test files. The returned codec will be used to show the table on how the characters were assigned bits later on.

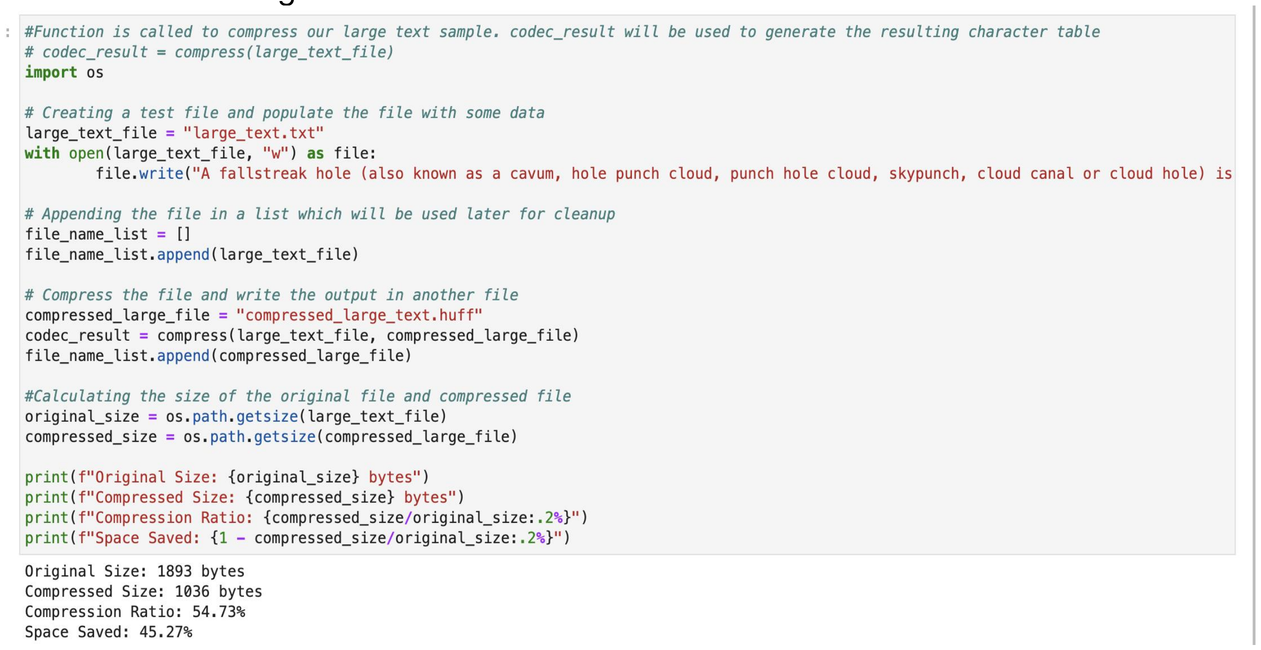


```python
import zlib

def compress_zlib(test_file, compressed_file):
    try:
        with open(test_file, 'rb') as file:
            data = file.read()
        compressed_data = zlib.compress(data, level=9)

        with open(compressed_file, "wb") as file:
            file.write(compressed_data)

    except FileNotFoundError:
         print(f"Error: Input file '{input_filepath}' not found.")
    except Exception as e:
        print(f"An error occurred during compression: {e}")
```

We will be using DEFLATE to compress the same text file that was used with Huffman coding. We should observe that the combined use of 2 compression techniques is often more efficient than on their own. We will be using zlib's compress function to perform this demo which will perform the same way DEFLATE would. We will be defining a new function named compress_zlib. This function will be similar to our compress function, but this time zlib will be used to compress our file.

3. Compressing file using Huffman coding

Huffman coding aims to reduce the amount of total bits used to store characters in text, so we should be able to see a very clear difference between the storage size of the original file and compressed file where the compressed file is significantly smaller than the original.

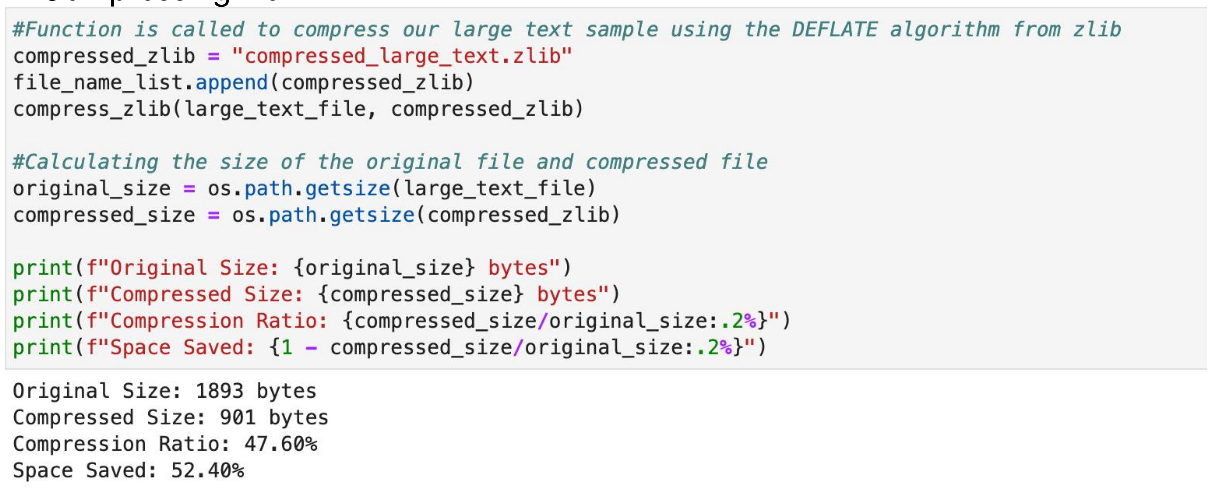


```
#Function is called to compress our large text sample. codec_result will be used to generate the resulting character table
# codec_result = compress(large_text_file)
import os

# Creating a test file and populate the file with some data
large_text_file = "large_text.txt"
with open(large_text_file, "w") as file:
        file.write("A fallstreak hole (also known as a cavum, hole punch cloud, punch hole cloud, skypunch, cloud canal or cloud hole) is

# Appending the file in a list which will be used later for cleanup
file_name_list = []
file_name_list.append(large_text_file)

# Compress the file and write the output in another file
compressed_large_file = "compressed_large_text.huff"
codec_result = compress(large_text_file, compressed_large_file)
file_name_list.append(compressed_large_file)

#Calculating the size of the original file and compressed file
original_size = os.path.getsize(large_text_file)
compressed_size = os.path.getsize(compressed_large_file)

print(f"Original Size: {original_size} bytes")
print(f"Compressed Size: {compressed_size} bytes")
print(f"Compression Ratio: {compressed_size/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size/original_size:.2%}")
```

```
Original Size: 1893 bytes
Compressed Size: 1036 bytes
Compression Ratio: 54.73%
Space Saved: 45.27%
```

The amount of space saved after compressing the file is 45.27% with a compression ratio of 54.73%.

4. Compressing file with DEFLATE



```
#Function is called to compress our large text sample using the DEFLATE algorithm from zlib
compressed_zlib = "compressed_large_text.zlib"
file_name_list.append(compressed_zlib)
compress_zlib(large_text_file, compressed_zlib)

#Calculating the size of the original file and compressed file
original_size = os.path.getsize(large_text_file)
compressed_size = os.path.getsize(compressed_zlib)

print(f"Original Size: {original_size} bytes")
print(f"Compressed Size: {compressed_size} bytes")
print(f"Compression Ratio: {compressed_size/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size/original_size:.2%}")
```

```
Original Size: 1893 bytes
Compressed Size: 901 bytes
Compression Ratio: 47.60%
Space Saved: 52.40%
```

Here we are compressing the same text file as we did in section 2 with only Huffman coding. This time the file will be compressed with the DEFLATE algorithm which will employ LZ77 and Huffman coding together.

We should see a better compression result than just Huffman coding since this text contains some repeated character sequences which will be efficiently compressed from LZ77 and then everything is compressed by Huffman coding when LZ77 is completed. This time the saved space was 52.40%, more than what we got with just Huffman coding.

5. Comparing Compression with repeated phrase

```python
#Repeating phrase
phrase_text_file = "phrase.txt"
file_name_list.append(phrase_text_file)
with open(phrase_text_file, "w") as file:
        file.write("Crazy? I was crazy once, they locked me in a room, a rubber roon

compressed_phrase_huff = "compressed_phrase.huff"
compressed_phrase_zlib = "compressed_phrase.zlib"
file_name_list.append(compressed_phrase_huff)
file_name_list.append(compressed_phrase_zlib)

compress(phrase_text_file, compressed_phrase_huff)
compress_zlib(phrase_text_file, compressed_phrase_zlib)

original_size = os.path.getsize(phrase_text_file)
compressed_size_huffman = os.path.getsize(compressed_phrase_huff)
compressed_size_deflate = os.path.getsize(compressed_phrase_zlib)

print(f"Original Size: {original_size} bytes\n")
print(f"Compressed Size Huffman: {compressed_size_huffman} bytes")
print(f"Compression Ratio: {compressed_size_huffman/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size_huffman/original_size:.2%}\n")
print(f"Compressed Size DEFLATE: {compressed_size_deflate} bytes")
print(f"Compression Ratio: {compressed_size_deflate/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size_deflate/original_size:.2%}")
```

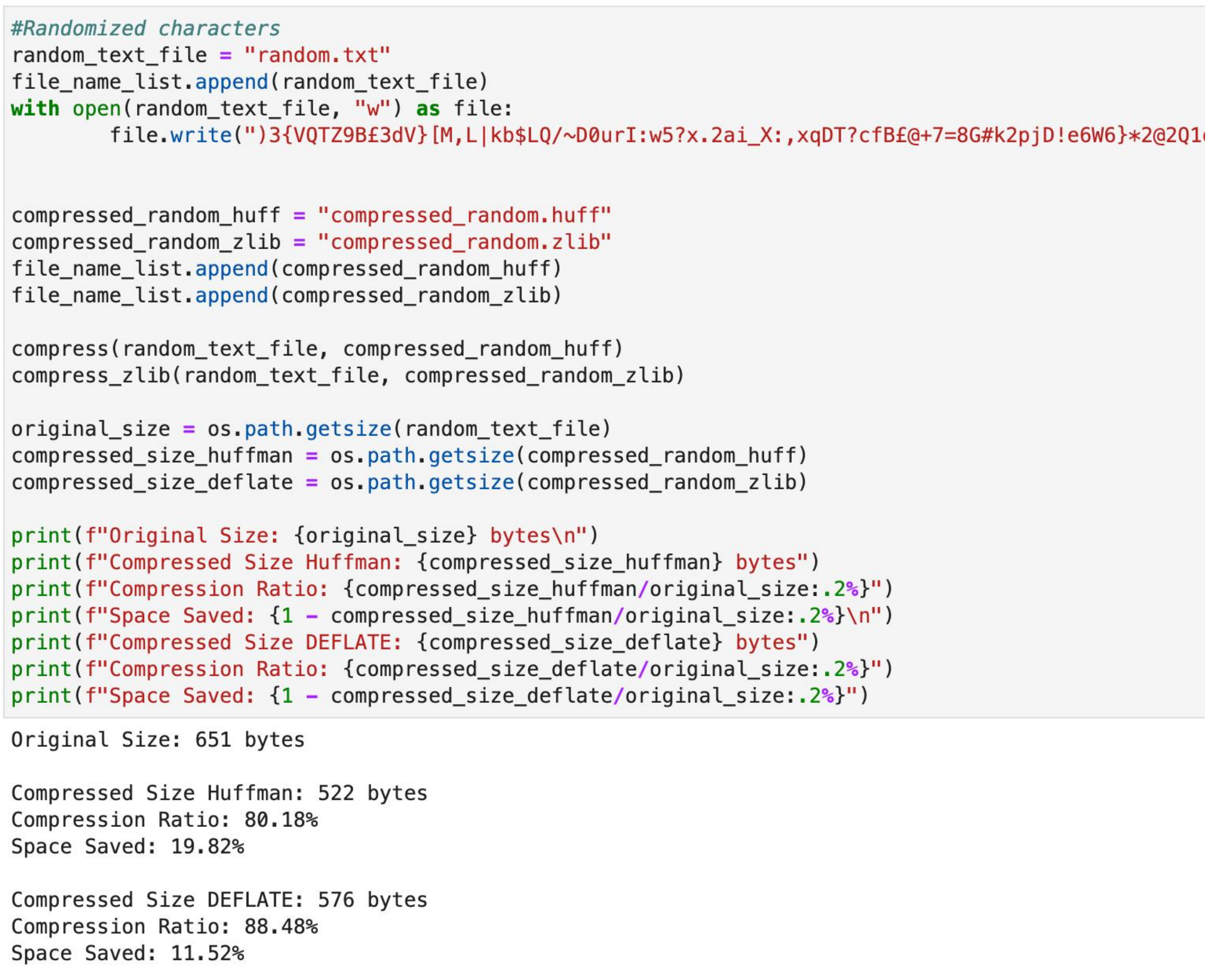
```
Original Size: 4800 bytes

Compressed Size Huffman: 2515 bytes
Compression Ratio: 52.40%
Space Saved: 47.60%

Compressed Size DEFLATE: 124 bytes
Compression Ratio: 2.58%
Space Saved: 97.42%
```

Since DEFLATE contains LZ77, the algorithm's ability to compress repeat phrases will be very efficient. We can compare how Huffman coding on its own performs comparably to DEFLATE when trying to compress a file containing repeated phrases. We should see that DEFLATE is greatly more efficient than Huffman coding with the compression results of phrase.txt.

6. Comparing Compression with random Characters

In this comparison, a text file filled with random characters will be compressed. Since there is a lack of repeating data sequences, using the combination of LZ77 and Huffman coding should be less efficient in this circumstance. The effectiveness of Huffman coding itself will also decrease since there are a larger pool of characters in this text file.

In this circumstance, Huffman coding by itself should be more efficient than attempting to use both techniques together since LZ77 will have a tough time finding any repeating character sequences resulting in a less efficient compression performance for DEFLATE. We can see this after running the following test.



```
#Randomized characters
random_text_file = "random.txt"
file_name_list.append(random_text_file)
with open(random_text_file, "w") as file:
        file.write(")3{VQTZ9B£3dV}[M,L|kb$LQ/~D0urI:w5?x.2ai_X:,xqDT?cfB£@+7=8G#k2pjD!e6W6}*2@2Q1


compressed_random_huff = "compressed_random.huff"
compressed_random_zlib = "compressed_random.zlib"
file_name_list.append(compressed_random_huff)
file_name_list.append(compressed_random_zlib)

compress(random_text_file, compressed_random_huff)
compress_zlib(random_text_file, compressed_random_zlib)

original_size = os.path.getsize(random_text_file)
compressed_size_huffman = os.path.getsize(compressed_random_huff)
compressed_size_deflate = os.path.getsize(compressed_random_zlib)

print(f"Original Size: {original_size} bytes\n")
print(f"Compressed Size Huffman: {compressed_size_huffman} bytes")
print(f"Compression Ratio: {compressed_size_huffman/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size_huffman/original_size:.2%}\n")
print(f"Compressed Size DEFLATE: {compressed_size_deflate} bytes")
print(f"Compression Ratio: {compressed_size_deflate/original_size:.2%}")
print(f"Space Saved: {1 - compressed_size_deflate/original_size:.2%}")
```

```
Original Size: 651 bytes

Compressed Size Huffman: 522 bytes
Compression Ratio: 80.18%
Space Saved: 19.82%

Compressed Size DEFLATE: 576 bytes
Compression Ratio: 88.48%
Space Saved: 11.52%
```

RESULTS

Types of documents	Original Size	Compressed with Huffman Coding	Compressed with DEFLATE
Large number of text	1893 bytes	1036 bytes	901 bytes
Repeated Words	4800 bytes	2515 bytes	124 bytes
Random Characters	651 bytes	522 bytes	576 bytes

CONCLUSION

We compressed text files with different patterns in order to compare DEFLATE and Huffman coding. Huffman coding had trouble unless the data was randomized, whereas DEFLATE did well with repeating data. The outcomes demonstrate how effective compression methods are in lowering storage requirements and improving data processing. Huffman Coding is efficient with random characters as compared to DEFLATE. DEFLATE is extremely efficient with repeated words document which saved space about 97.4%.

FUTURE SCOPE

Future research will examine sophisticated algorithms such as Brotli and Arithmetic Coding, assess their effectiveness, and if possible also applying compression methods to audio files for better transmission and storage. This could improve data handling much further for a variety of applications.