Computer Science Extended Essay

Topic:

Assessing the compression efficiencies of LZ77 and Zstandard algorithms.

Research Question:

How does the compression algorithm of the LZ77 technique compare to the Zstandard technique in relation to the ratio of uncompressed to compressed text blocks?

Word Count
3984

Contents

Introduction

Compression is a concept which is increasingly becoming more relevant and so are compression algorithms. Compression is a concept used to reduce the size of files to improve storage, transmission, and security. Algorithms are a series of instructions that are required to occur so that a task or function can be completed. In the sense of compression algorithms, those are a series of instructions that can diminish the size of data with no or very little loss of data. A key feature of compression algorithms is that it encodes and decodes information, through the process of decreasing the size of data into parts known as metadata, which is a type of data which provides information on the original collection of data encoded. Some applications of compression include the improvement of computer communication, and sharing and usage of audio and image files.

Compression techniques will be fundamental in the improvement of storage capacity, as this can affect the largest mediums of communications in the world, in the sense of improving streaming, maintaining fast and equal access to all websites (net neutrality), et cetera. This investigation will study one of two types of compression algorithms: lossless algorithms. These are compression methods which decrease storage sizes of different files whilst maintaining all information in them, unlike their counterparts which decrease the file sizes whilst losing some content and information in them (lossy compression). Lossless is more commonly used when compressing images and audios, where a slight decrease in content and quality won't be necessarily perceivable or matter. In the case of text files and documents for instance, lossless compression is used as to not lose important human information.

Compression algorithms still possess technical challenges, hence the attempt at investigating ***"How does the compression algorithm of the LZ77 technique compare to the Zstandard technique in relation to the ratio of uncompressed to compressed text blocks?"***

Background Information

Lempel-Ziv Algorithm (LZ77)

The Lempel-Ziv algorithm is more commonly known as the LZ77 algorithm, one of two versions of the Lempel-Ziv compression techniques. This is a compression algorithm which functions on the principle of cross-matching sequences of characters when compressing string text blocks. The algorithm is fast because it encodes on the run. It consists of a search buffer, containing the already encoded code, and a look-ahead buffer, containing the code that is going to be encoded [1]. Another key concept of the LZ77 is the use of triples to represent matching sequences of characters [2]. Those are tuples containing key arguments which contain:

- Offset <o>: The number of positions needed to move backwards in the search buffer to find the first character of the sequence of characters (represented as an integer).
- Length <l>: The number representing the size of the sequence of characters (represented as an integer).
- Codeword <c>: Represents the character that follows from the new-found match which is also encoded with the sequence.

Notation: (o, l, c)

The utilization of triples decreases the size of text sequences due to its matching scheme with other future equal sequences. The metadata in the triples to represent the sequences is also smaller in size and ends up compensating for all repeated sequences in strings, as one triple can represent many reoccurring combinations. The LZ77 is a linear algorithm, meaning that it searches the text blocks for the combinations of characters in the order they're arranged. This also means that the first time a sequence in a string appears, it will be registered as single characters, where the first two places in the triple are zero, and the codeword is the letter itself. Codewords which represent data sequences are also stored by the algorithm's triples.

Example of LZ77 [3]

The encoding of the string "ABRACADABRAD" using LZ77 goes as follows:

1. Separate the string into characters:

   ['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a', 'd']

2. Place all characters in the look-ahead buffer.

3. Commence process of using triples to encode:

   Pointer = ('x')

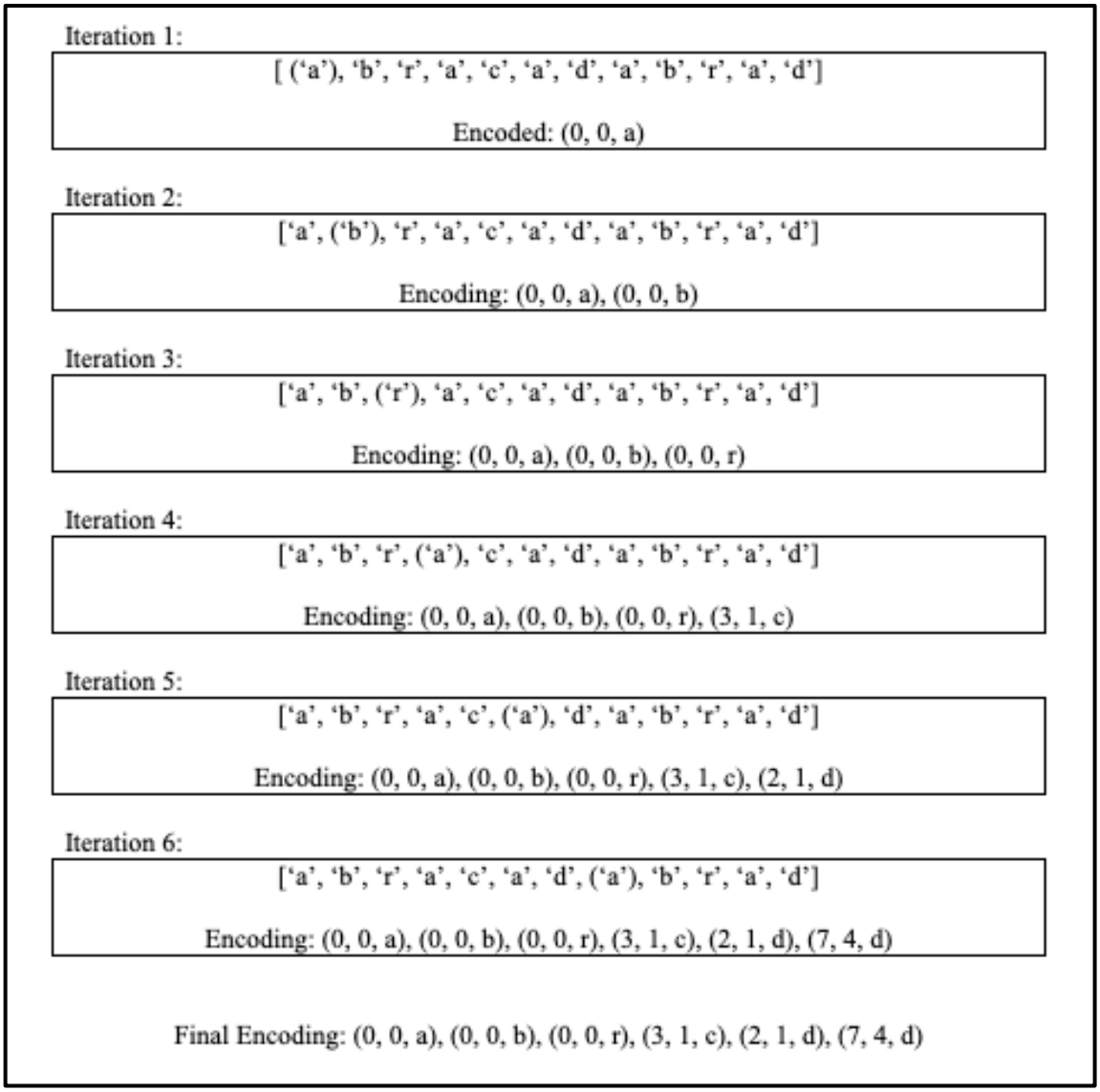

Figure 1: Image of iterations of LZ77 triples encoding (Made by author).

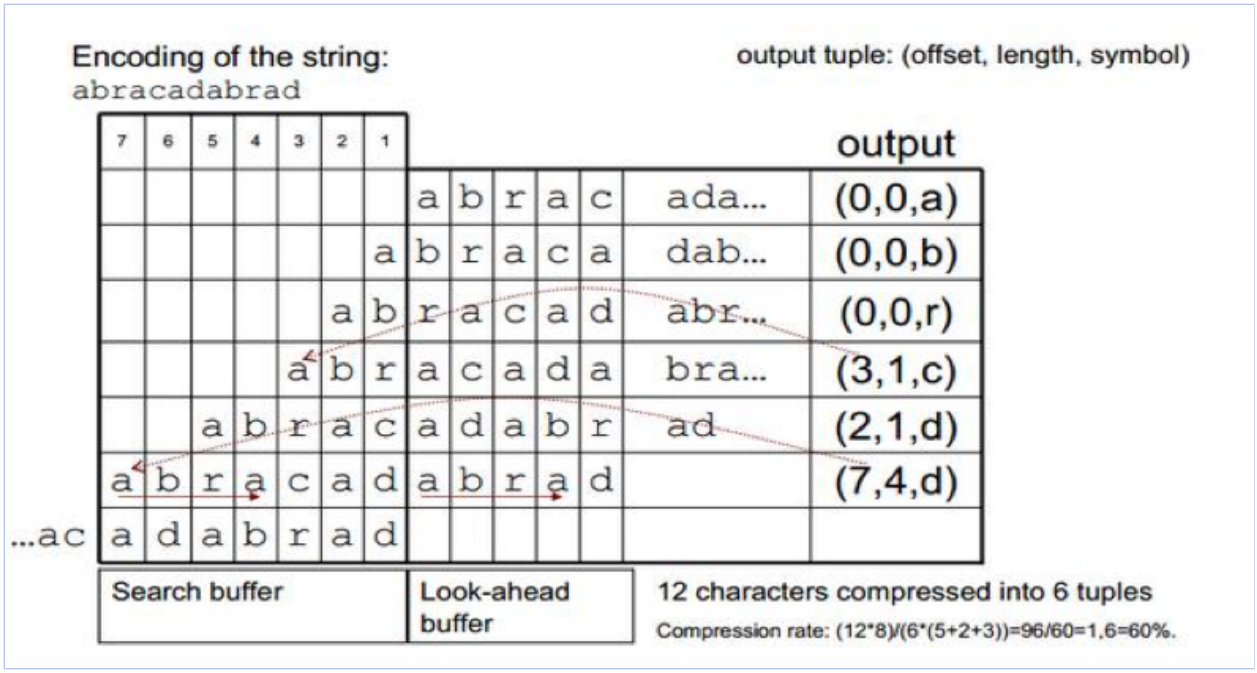

Figure 2: Illustration of LZ77 encoding. (International Journal of Engineering Science and Innovative Technology)

The decoding process of LZ77 works by adding together the size of the search and look-ahead buffer to store the triples, and then solving the triples by order of string characters compressed, a first in first out technique seen in queue abstract data types [4]. That means that a new combined buffer with the size of both encoding buffer solves the triples in strings and moves onwards after solving a triple to make way for others, as the buffers are typically smaller than the size of the uncompressed data.

Zstandard Algorithm (ZSTD)

Zstandard is a recently created algorithm developed on top of LZ77 [5]. The core algorithm of cross-matching common series of characters is also true for ZSTD. What extends itself and makes it different from LZ77 is the utilization of modern computing techniques and modern hardware aspects.

Modern computers and its Central Processing Units (CPUs) can run and decode instructions parallel to others if the instruction tasks are not intertwined. If the instructions set to decode are connected in some sense, the CPU utilizes a branchless design of decoding:

Independent instructions:

b = a2 + a3

c = a4 + a5

Intertwined instructions:

b = a2 + a3

c = a4 + b

The branchless design means that when the CPU encounters a problem where instructions relate to one another in parallel decoding, it runs the intertwined instructions by predicting what is the value of the key unsolved instruction recurring in another instruction, like in the example of 'b = a2 + a3' and 'c = a4 + b', where 'c' will be run based on a predicted outcome of 'b' due to statistical analysis of what is the most probable outcome based on commonality.

Alongside that, ZSTD also makes use of entropy coding. Entropy in this context is defined by the smallest number of bits needed to encode a character. Entropy coding is also based on the 'randomness' of certain characters. That is, the frequency to which a certain symbol appears in an instruction needed to be encoded.

The formula to calculate the entropy of a set of elements (H) is:

$$H = \sum_{i=1}^{n} -p(D_i)\log_2\big(p(D_i)\big)$$

[6]

Where:

D_i = Discrete distribution of character.

p = Probability of occurrence.

$-\log_2\big(p(D_i)\big)$ = Information associated to D_i

The reason for working with logarithms in base 2 is to calculate the probabilistic models specifically of binary values, which is a base two system of number representations that computers can easily read.

The following simplified example will demonstrate how the 'randomness' of a symbol (N) plays a role when encoding itself:

$$N = -\log_2 p$$

In a proposed scenario, the probability of a letter 'e' reoccurring is $\frac{1}{4}$, as it's a common letter in the English lexicon, whilst for 'z' is $\frac{1}{64}$, as it is a less frequent character. Hence, the number of bits needed to encode the characters will be:

Character 'e':

$$N = -\log_2 \frac{1}{4}$$

$$= -\log_2 2^{-2}$$

$$= 2$$

Character 'z':

$$N = -\log_2 \frac{1}{64}$$

$$= -\log_2 2^{-8}$$

$$= 8$$

This encoding schematics allow for a variable-length code, where different symbols can be represented using fewer binary digits. This means that the higher a symbol's frequency in a set, the smaller its length-code will be, consequently optimizing storage occupancy.
Such allocated bit estimates ignore the theoretical limit of lossless compression in computation [7], hence the large allocation sizes.

ZSTD utilizes the ANS (Asymmetric Numeric System) technique for entropy encoding, which is used to encode and combine the different symbol frequencies in a concise way, using the fast search time complexity of Huffman Trees and Huffman Encoding and the coding ratio of arithmetic coding [8].

Huffman encoding and its tree representation are organized based on the frequency of data symbols in a line of string. The example below demonstrates how a Huffman tree is constructed:

Example of Huffman coding [9]

The Huffman encoding and representation of the string of symbols "BCAADDDCCACACAC" is as follows:

1. Separate the string into characters:

   ['b', 'c', 'a', 'a', 'd', 'd', 'd', 'c', 'c', 'a', 'c', 'a', 'c', 'a', 'c']

2. Calculate frequencies of each character:

   'b' = 1

‘c’ = 6

‘a’ = 5

‘d’ = 3

3. Sort characters in order of their frequencies:

   ‘b’ = 1

   ‘d’ = 3

   ‘a’ = 5

   ‘c’ = 6

4. Initialize Huffman tree by creating an empty node.
5. Collect two least occurring characters in string and add the smallest frequency to the left of the new empty node, and the second smallest to the right of an empty node. Assign the sum of the two smallest occurring frequencies to the initially empty root node:

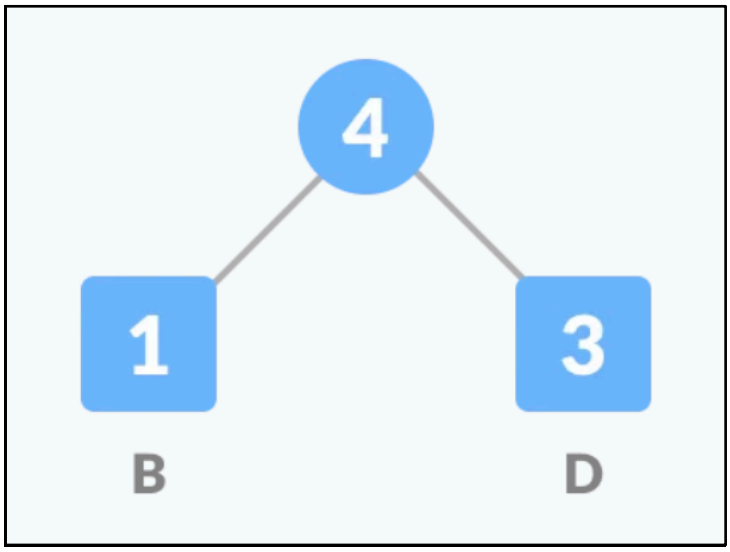

Figure 3: Representation of current Huffman tree with two least occurring frequencies.

6. Append ‘a’ to the tree by creating a new root node, where its value is the sum of its child nodes, those being 4 (left child) and A: 5 (right child), due to the 5 being greater than 4:

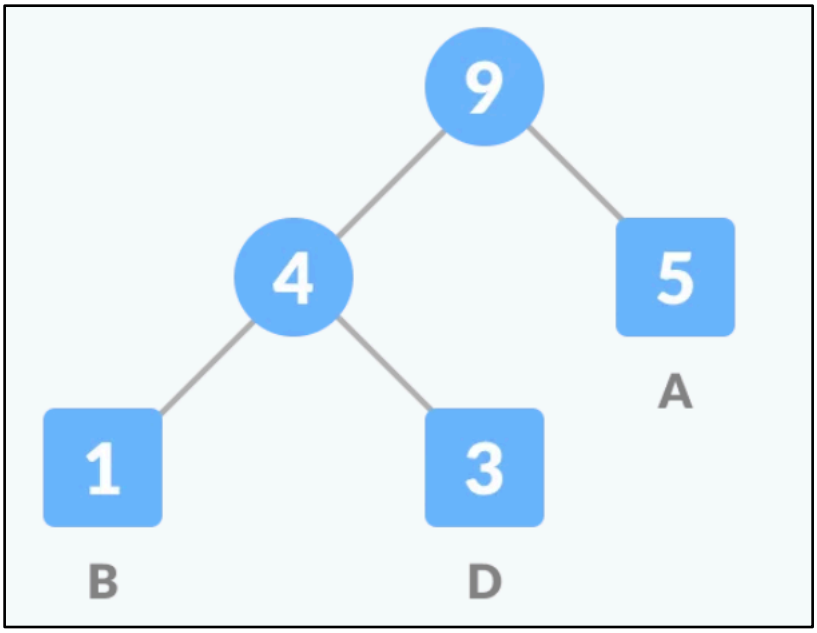

Figure 4: Representation of current Huffman tree with the A: 5 node appended.

7. Add C: 6 to the tree by repeating step 6, where a new root node is created which is the sum of C: 6 and 9 (15) and append C: 6 to the tree as a left child to the root node 15, as the value of 6 is smaller than 9.

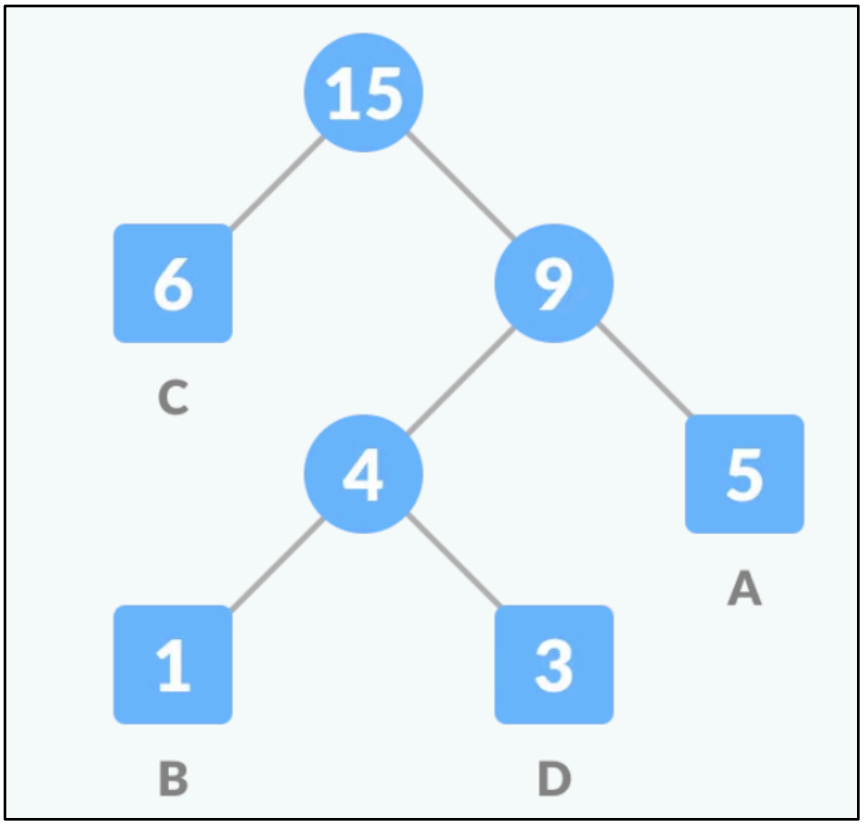

Figure 5: Representation of final Huffman tree with node C: 6 appended.

8. Finally, for each node in the tree, assign the value 0 for values to the left of a root, and 1 to the right.

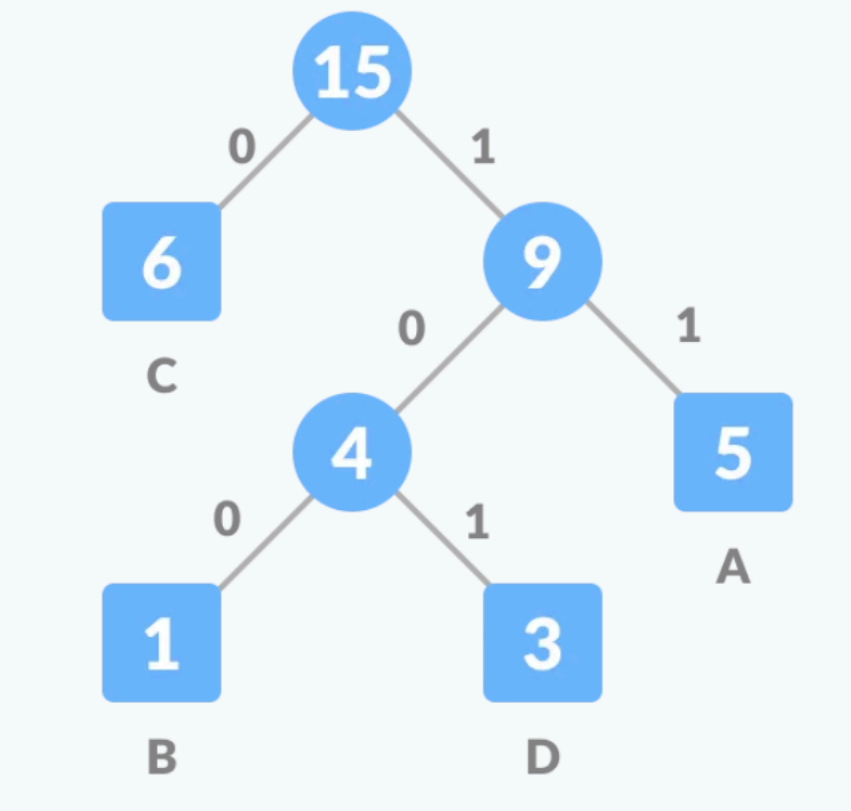

Figure 6: Representation of Huffman tree with binary values for branches.

Such technique of encoding values using Huffman trees allows more frequently occurring symbols to have a smaller storage amount, as is the case of C, which is represented by 0. On the other hand, less frequent occurring values such as B, will be represented by 100, as it is the path from the root node to the B node.

The time complexity to encode Huffman trees is $O(n\ log\ n)$, which is consistent with the equation for the sum of the entropy of set of symbols [10].

Methodology

The following experiment is run on Python, an interpreted and procedural programming language, meaning all lines of code are read by the computer in order and one by one. The experiment will analyze the compression ratio of a growing data set of text values and will have 100 mean data points. The details of the experiment follow:

Independent Variable

Each iteration of the program will generate a dictionary with specified keys. In this experiment, the dictionary is a data type which has a collection of values that possess an associated string or number to a key word or value.

Example of dictionary:

```
{'keyword': 99}
```

The keys in this experiment are fixed strings which cannot be altered.

Furthermore, during each iteration of the program, a text block will be generated which will vary in length and have a collection of 25 out of the 26 keys that will have associated numbers to them. The text block will be a dictionary with different symbols according to its size. Furthermore, the associated string to each value in the dictionary will be composed of a series of the 25 random keys with 25 randomly associate numbers, varying from 0 to 99.
An example of the text block (dictionary) created after each iteration of the program is demonstrated:

```
{'0': {'Unity': 64, 'Delta': 7, 'Yanks': 68, 'X-ray': 52, 'Kilos': 79, 'Oscar': 57,
'India': 45, 'Bravo': 54, 'Count': 75, 'Lemon': 43, 'Fetch': 33, 'Vitor': 73, 'Sound':
71, 'Zebra': 59, 'Night': 58, 'Enter': 63, 'Tango': 60, 'Hotel': 69, 'Wooly': 66,
'Quack': 67, 'Grade': 70, 'Julie': 74, 'Panda': 72, 'Romeo': 77, 'Menlo': 78}
```

As you can see from the example, only one entry in the dictionary is shown, and the following entries will vary the string indicator, commencing from 1.

Each iteration of the program will have the selected sizes of the text blocks in terms of built-in keys, which will be chosen at random, to satisfy the demanded text block size, utilizing Python's random number module [11]. Note that the such sizes are based on the number of keys in the text blocks, and not the actual number of characters in the text string, all keys have the same five-character length and will be stored on a dictionary, pointing to a number. The increments of text block size will be in 200's, ranging from 200 to 4,000, and each point being repeated five times to mitigate error.

Dependent Variable

The compression ratio is the initial size of data divided by the compressed size of data. The experiment will run simultaneously for both LZ77 and ZSTD with the embedded modules emulating the algorithms in Python being '`lz4`' [12] and '`zstd`' [13]. The original data size and compressed data size will also be observed independently from the ratio.

Controlled Variables

Table for the list of control variables can be seen in the Appendix.

Procedure

1. 200 Dictionaries are generated with 25 items and random number pointers ranging from 0 to 99.
2. The Dictionary is encoded using the Ultra JavaScript Object Notation module [15], a module that allows the transfer of text between files.
3. Uncompressed dataset is saved.
4. Length of text block is saved.
5. LZ4 and ZSTD algorithms are applied adapted text blocks which are then compressed.
6. Compressed length of text block is saved.
7. Compression ratio is found by dividing the uncompressed dataset by the compressed one.
8. Compression ratio, uncompressed length, and compressed length are inputted in Microsoft Excel alongside the number of Dictionaries.
9. Experiment is repeated five times.
10. The nine steps above are followed with an increased number of dictionaries by 200 until there are 2,000 dictionaries.

(Python script and raw table of trials can be seen in Appendix).

Data Presentation

Number of Dictionaries	Mean Size
200	48096
400	94536
600	143713
800	190484
1000	237327
1200	286110
1400	334656
1600	380587
1800	429822
2000	476640

Table 1: Average size of text blocks of number of dictionaries.

LZ77	Compressed Length
Number of Dictionaries	Average
200	16918
400	31958
600	47820
800	62923
1000	78044
1200	93695
1400	109317
1600	124074
1800	139803
2000	154814

Table 2: Average compressed length text blocks with LZ77 algorithm.

ZSTD	Compressed Length
Number of Dictionaries	Average
200	11205
400	21628
600	32424
800	42838
1000	53272
1200	64091
1400	74814
1600	85056
1800	96012
2000	106405

Table 3: Average compressed length of text blocks with ZSTD algorithm (Made by author with Microsoft Excel)

LZ77	Ratio
Number of Dictionaries	Average
200	2.83602665
400	2.94707824
600	3.00530124
800	3.02723647
1000	3.04108093
1200	3.05362745
1400	3.06132631
1600	3.18820326
1800	3.07438708
2000	3.07879815

Table 4: Average Compression ratio of LZ77 algorithm (Made by author with Microsoft Excel)

ZSTD	Ratio
Number of Dictionaries	Average
200	4.29231626
400	4.37470150
600	4.43235494
800	4.40659725
1000	4.45527618
1200	4.46408436
1400	4.47314379
1600	4.47456838
1800	4.47674232
2000	4.47949721

Table 5: Average Compression ratio of ZSTD algorithm (Made by author with Microsoft Excel)

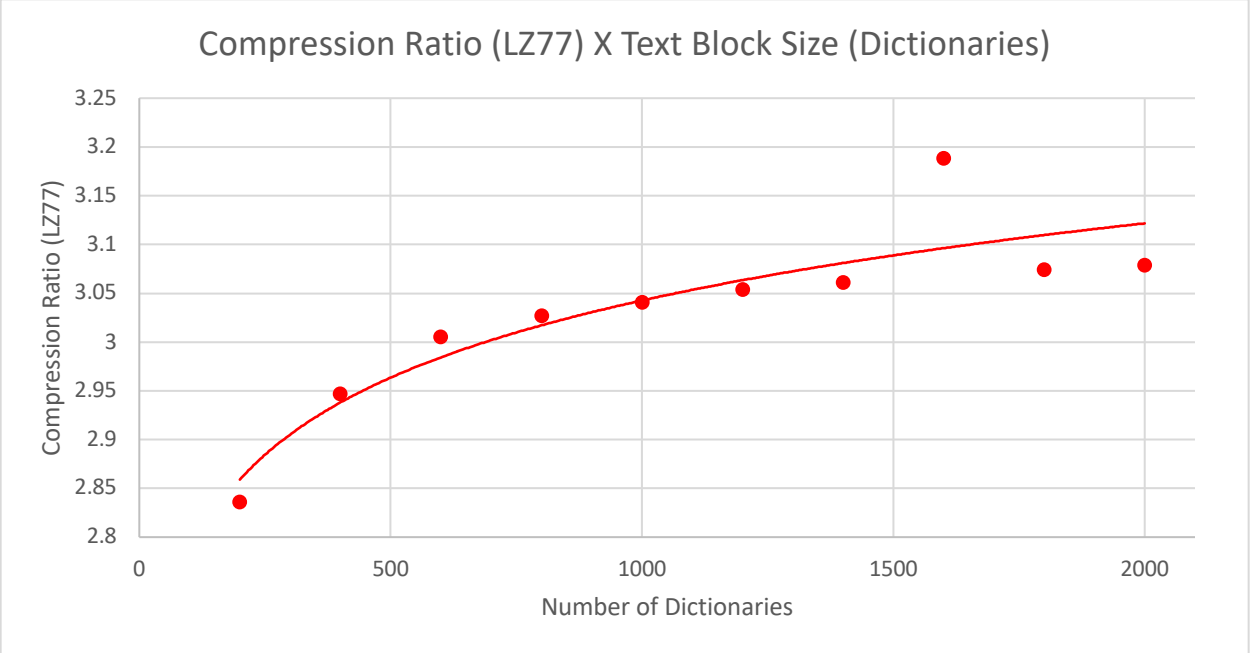

Figure 7: Graphical representation of compression ratio of LZ77 (Made by author with Microsoft Excel).

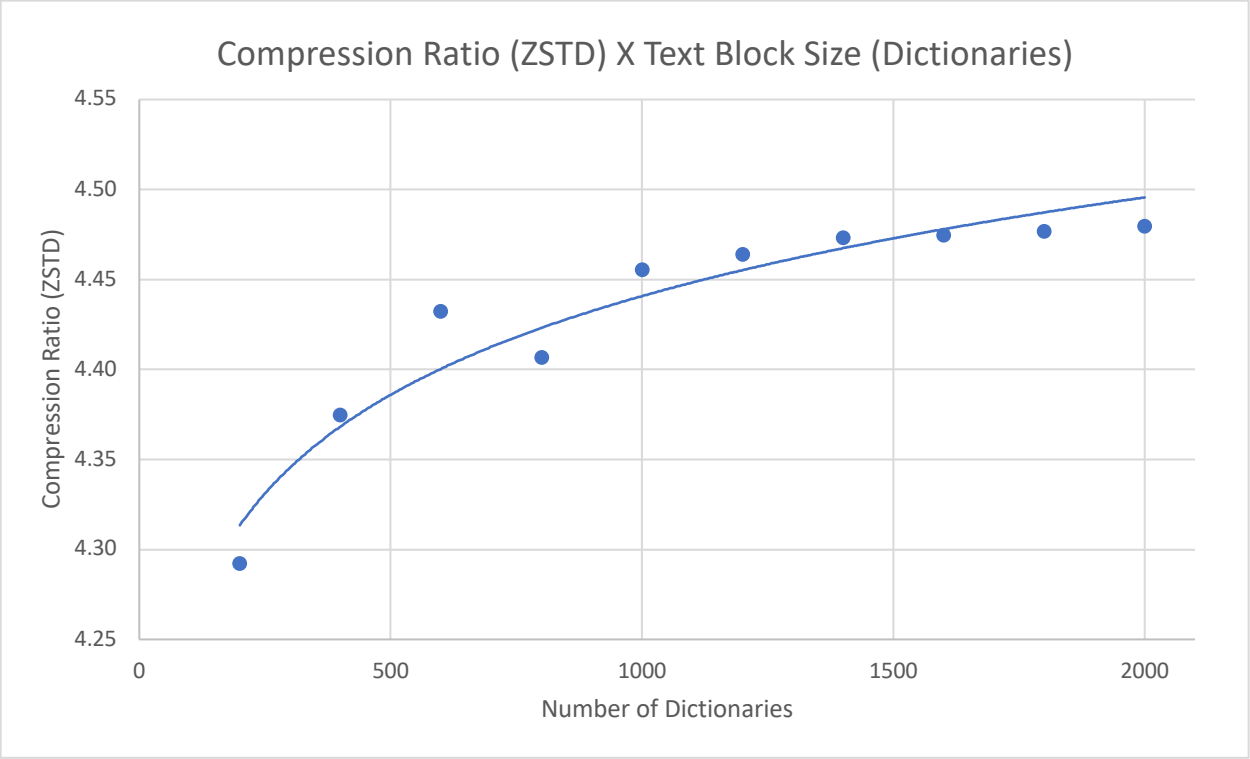

Figure 8: Graphical representation of compression ratio of ZSTD (Made by author with Microsoft Excel).

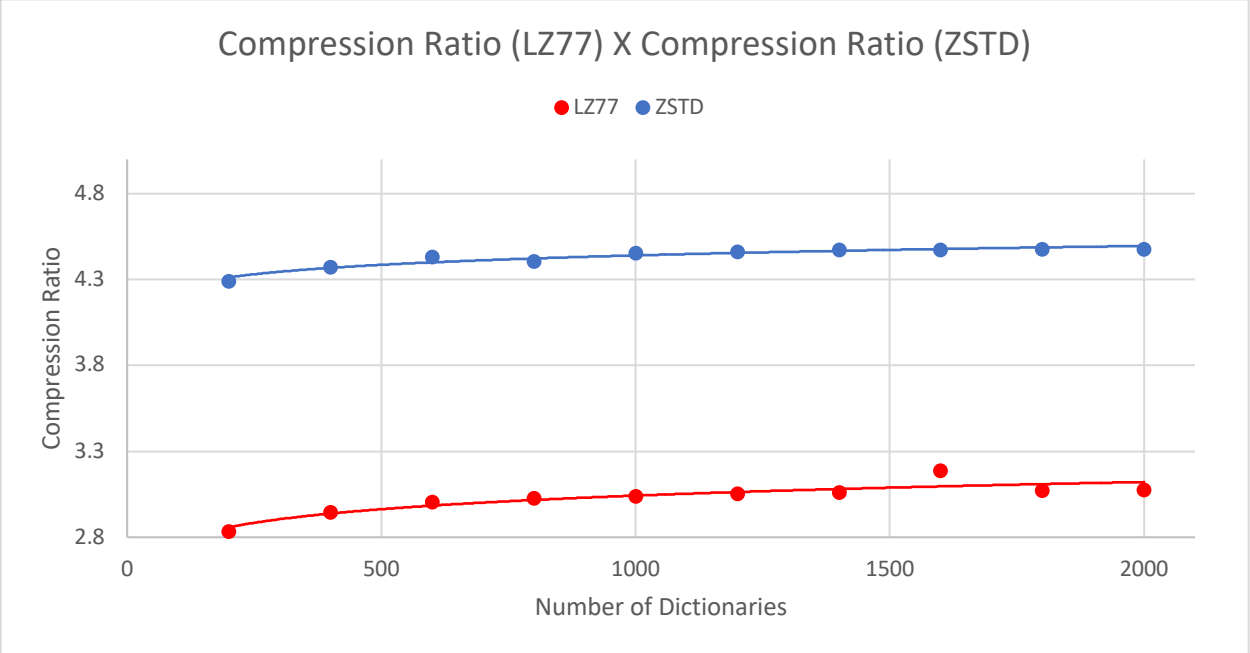

Figure 9: Comparison of compression ratio of the LZ77 and the ZSTD algorithms (Made by author with Microsoft Excel).

Data Analysis

The number of dictionaries and uncompressed length when comparing to the compression ratio and compressed lengths are very similar and show no dramatical changes in visual representation. The compressed ratio for LZ77 and ZSTD algorithms and the dictionary sizes contain a logarithmic relationship. The LZ77 compression ratio when applied a curve of best fit, has a regression rate of 0.91, compared to the ZSTD which has a regression rate of 0.96. It appears the ZSTD graph has more anomalies, since the regression line is not passing near various points. Yet, it is possible to see, by neglecting the anomaly point (800, 4.40), that the points align to make an almost perfect logarithmic relationship. Still, besides point (800, 4.40) in the ZSTD line, other outliers are: (600, 4.43) and (1000, 4.46), which slack off the trendline.

The LZ77 graph becomes flatter as the dataset sizes increases, providing evidence for the lower regression rate of 0.91. Besides LZ77 having a lower gradient, point (1600, 3.10) outperformed on the ratio metrics compared to other points and was a greater outlier than the (800, 4.40) point in the ZSTD graph. This is a result of the uncompressed length of the string blocks, not the compressed length which was closer to its subsequent point's compressed length than its previous point compressed length, showing the string wasn't compressed to a good size. Overall, LZ77 had a lower average compression ratio of 3.031306578 than the ZSTD algorithm, which possessed a 4.43292822 ratio.

Evaluation

Some experimental techniques justify the results gathered such as the random but limited number of keys. The fact that only 26 keys were scrambled with different appearance frequencies, means that both algorithms overperformed, as they are both based on the cross-matching technique of searching for common sequences of characters. That signifies that more common combinations of characters appeared more frequently, as well as single character combinations, as each key began with each letter of the Latin alphabet. Regular compression ratios for both algorithms are much smaller and range at 2.10 and 3.1 for both LZ77 and ZSTD, respectively [16].

As it is possible to observe from the graph and the data tables, both LZ77 and ZSTD algorithms follow a logarithmic complexity. This means that for very small data sizes, both algorithms excel at encoding as their gradient is seen to be greater for larger data sets. Many aspects on both algorithms suggest why they are not being able to improve their compression ratios.

First off, both algorithms have issues with a theorical threshold which cannot be surpassed. Additionally, PhD Eduardo da Silva, professor at Universidade Federal Rio de Janeiro (UFRJ) claims that the environment in which the algorithms run, can have a drawback in the true potential of these algorithms. Da Silva further states that there can be limitations in the hardware of the machine that don't permit high-performing versions of the algorithm to be executed [17], and hence can have a tradeoff in the speed ratio of the algorithms. In the experiment, the modules used for the LZ77 and ZSTD algorithms can have tradeoffs in the compression ratios and speeds of the algorithms, which have impacted the results. Such algorithms can have

changing speeds and ratio depending on the parameters and arguments passed to their module's methods [18].

Secondly, with the LZ77, all issues link to the schematics of the algorithm itself, and not any outside factors such as hardware or theoretical holes. Firstly, the look-ahead buffer and the search buffer are extremely small, meaning that they commonly can't link past sequences with recently found sequences as the search buffer is much smaller than the size of the text block, even after encoding with triples. Also, many long length matches cannot be combined as the maximum size of a sequence match can only be the size of the look-ahead buffer, discarding much larger encoding possibilities [19]. LZ77 algorithm is also suited for decoding other alphabets and other data types, one of the reasons why it struggles with increasing data lengths as it also searches and saves memory space for other alternative encoding data types, and such logarithmic relationship can be proved with discrete mathematics [20].

The ZSTD algorithm has all the issues mentioned above, as the algorithm is derived from LZ77. However, the problems on the algorithm can be countered by the design of finite state entropy of the algorithm, parallel encoding with the use of hardware, and its branchless design. But even such aspects that differentiate ZSTD from its counterparts, can have its problems. Firstly, the Shannon limit of the entropy of encoding doesn't allow the algorithm to improve its compression ratio without suffering some loss in data, which is unacceptable and in most compression algorithms, inexistent in text files. As the data set is also larger, the encoding entropy is less effective than compared to smaller datasets, as more uncommon letters in the alphabet occur more frequently, thus signifying that more bits are needed to decode more frequent letters, as

entropy is related to probable outcomes and distribution and the total probability of all systems equals one.

On the other hand, the ZSTD algorithm utilizes compression techniques such as Huffman encoding which are known to be greedy algorithms, these are a series of algorithms which always looks for the more optimal solution. Due to this property of finite state entropy, it has allowed the ZSTD to outperform the LZ77 compression ratios and improve its compression ratios after each iteration of the experiment with larger data sets.

This logarithmic relation of the ZSTD and LZ77 algorithms can also be seen as a positive, as its compression ratios improve for increasing text blocks. In this case, it is shown that there is an improvement in the ratios, suggesting that the algorithms truly benefited from the experiment surrounding itself, through the repetition of text blocks and text combinations, and the limitations in speed the environment had in the runtime of the algorithms, which resulted in an opportunity for the algorithms to explore their compression ratios in this case.

Implication of Findings

The findings of this investigation indicate that the ZSTD is a more effective way of compression strings of data, in terms of compression ratio. More data is being created and uploaded on the internet, and storage systems cannot cope with the increasing flow of data. Such new developed compression algorithms join parts of other algorithms and concepts of machine hardware and information theory to improve the rate at which data sizes are decreased on the internet. Both businesses and customers can take advantage of these breakthroughs. Businesses can implement

such compression techniques into their systems for communication and storage purposes, effectively making their products more attractive, whilst customers save money on uploading more data to cloud storage systems, for instance, for the same price. Communication between institutions and regions can improve as data is compressed and encrypted before being transferred to another place. A smaller data size means data transmission becomes faster and more secure.

Organizations such as the IB can also benefit from such innovations. The IB is an international program which supports thousands of students worldwide and needs to be constantly communicating with schools to clarify doubts, provide new updates on the curriculum or handle special student cases. More effective compression would mean the IB, and schools can communicate faster, effectively, as most times messages can contain sensitive or important information that is too vital to lose. Besides that, the IB also keeps a comprehensive record of coursework from students, exam papers and other information for many years. With more efficient ways of compressing data, organizations can store more information on servers without necessarily having to update their file storage systems frequently.

Further Scope of Investigation

Text based compression is one of the many types of compression existent, but it's not the only one. Besides text files, image, video and audio files are amongst the most present types on the internet. Such investigation could be repurposed to handle video, audio, and image compression with the same or different algorithms, as LZ77 is used on the DEFLATE [21] compression

algorithm to rearrange the pixels and hexadecimal values of images into bits ranging from denary of 0 to 255 [22].

Looking at other evaluating techniques of compression, Da Silva suggests that a key aspect of compression algorithms are also its speeds, as it could bring in light other evaluation techniques such as time complexities of the algorithms [23], and hardware limits with CPU clock speeds, for example. For such experimentation to occur, another programming language should be used. That is because Python (the language used in the experiment) is an interpreted programming language, meaning that all lines of code are read one by one, to check for errors. That may cause some delays in between lines due to hardware aspects for example. Compiled languages would be better suited for the job, as the code is joined into an executable file and run altogether. Languages suited for such type of task and fulfill the criteria as compiled are Java and C++.

Some of the limitations of the experiment include the usage of a small range of keys used in the text block, resulting in more frequent matches, especially of single characters, as each key initially composed a letter of the alphabet. Furthermore, the small size of text blocks isn't realistic, as real-world compression handles much larger text blocks composed of millions of characters, and multiple text blocks. This was limited by the environment in which the experiment occurred, as common laptops can't emulate large-scale compression processes.

Conclusion

The aim of this investigation was to compare different type of compression algorithms and its effectiveness, more specifically: ***"How does the compression algorithm of the LZ77 technique compare to the Zstandard technique in relation to the ratio of uncompressed to compressed text blocks?"***. Due to experimental-driven analysis, it is possible to conclude that the LZ77 compression algorithm is inferior to the Zstandard algorithm in terms of compression ratio. More so, it is also slightly less scalable than Zstandard, as its logarithmic relationship to increasing data sizes, if excluding anomalies, showing that the LZ77 compression ratio stays almost put after a certain dataset size. Both algorithms also outperformed on the ratio metrics as the experimental procedure was better suited for the sequence-matching characteristics of the algorithms. Yet, it should be noted that such logarithmic relationship isn't a result of a complication of the algorithms, but rather a result of the effective use of the binary base two system to utilize bits to encode the data.

Zstandard proved to be a superior algorithm due to its modern algorithm and use of modern aspects of computing and theoretical Computer Science, such as parallel encoding and finite state entropy, and should be used more frequently in more aspects of Computer Science as it has shown to be a major development on the Lempel-Ziv work and other LZ algorithm variations.

References

1. https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/concept.htm - Concept of the LZ77 algorithm, Stanford University. Accessed 12/03/2022.
2. https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097 - How data compression works: exploring LZ77, Towards Data Science. Accessed 12/03/2022.
3. https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097 - Adapted LZ77 example. Accessed 04/08/2022.
4. https://www.cs.princeton.edu/courses/archive/fall18/cos126/lectures/CS.12.StacksQueues-2x2.pdf - Stacks and Queues. 'First In, First Out' concept of Queues, Princeton Computer Science Department. Sedgewick, R. Wayne, K. Accessed: 12/03/2022.
5. http://www.stringology.org/DataCompression/lz77/index_en.html - LZ77 Description and Implementations. Accessed: 12/03/2022.
6. http://www.math.tau.ac.il/~dcor/Graphics/adv-slides/entropy.pdf - Entropy Encoding equation. Tel Aviv University. Accessed 03/09/2022.
7. http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html - Finite State Entropy in Information Theory, Shannon Limit. Published: 16/12/2022. Accessed: 13/03/2022
8. https://arxiv.org/abs/1311.2540 - Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. Duda, S. Published: 11/11/2013. Last revised: 06/01/2014. Accessed: 03/09/2022.

9. https://www.programiz.com/dsa/huffman-coding - Huffman coding and Huffman trees example. Programiz. Accessed 04/09/2022.
10. https://www.programiz.com/dsa/huffman-coding - Huffman coding time complexities. Programiz. Accessed 04/09/2022.
11. https://docs.python.org/3/library/random.html - Generate Pseudo-random numbers, Python Documentation at python.org. Accessed 13/03/2022.
12. https://lz4.github.io/lz4/ - lz4 Module Descriptions, Comparisons and Use. Accessed 13/03/2022.
13. https://pypi.org/project/zstd/ - zstd Module, Python Package Index. Dryabzhinski, S. Stuk, A. Accessed: 13/03/2022
14. https://www.wise-geek.com/what-is-a-computer-terminal.htm - What is a Terminal, Wise-Geek. Modified: 03/02/2022. Accessed: 13/03/2022.
15. https://pypi.org/project/ujson/ - UltraJSON Module, Python Package Index. Tarnstrom, J. Accessed: 13/03/2022.
16. https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/ - Comparing Compressions, Meta Engineering. Published: 31/08/2016. Accessed: 13/03/2022.
17. Interview with PhD Eduardo A. B. da Silva, professor at UFRJ (eduardo@smt.ufrj.br) - http://www02.smt.ufrj.br/~eduardo/ - Accessed 04/09/2022.
18. https://indico.fnal.gov/event/16264/contributions/36466/attachments/22610/28037/Zstd_LZ4.pdf - zstd and lz4 compression ratios and speeds. University of Lincoln-Nebraska. Bockelman, B. Shadura, O. Accessed – 13/03/2022.

19. https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/shortcoming.htm - Shortcomings of the LZ77 algorithm, Stanford University. Accessed 13/03/2022.
20. https://arxiv.org/pdf/1707.09789v2.pdf - Relations Between Greedy and Bit-Optimal LZ77 encodings, University of Helsinki. Kosolobov, D. Accessed: 13/03/2022.
21. https://www.keycdn.com/support/what-is-image-compression - What is image compression, KeyCDN. Accessed: 13/03/2022
22. https://www.geeksforgeeks.org/what-is-image-compression/ - Image Compression Techniques, Geeks for Geeks. Updated: 12/01/2022. Accessed: 13/03/2022
23. https://towardsdatascience.com/logarithms-exponents-in-complexity-analysis-b8071979e847#e39a – Logarithms and Exponents in Complexity Analysis, Towards Data Science. Alraja, H. Published: 16/08/2021. Accessed: 13/03/2022.

Transcript of interview with PhD Eduardo A. B. da Silva, professor at UFRJ – eduardo@smt.ufrj.br:

Q: What are the applications of the compression algorithms studied in this investigation, and others?

A: Firstly, there are the primordial applications to compress computer archives: programming scripts, text files, audio files. How ZIP and RAR files do such things. Other applications include multimedia signals. Those being voice and audio files. When using streaming service nowadays, you're essentially seeing video and audio which is already compressed, and that use algorithms for compression in various levels of the process. Of course, such algorithms can compress things

beyond what you've analyzed (text blocks). But of course, the basis of the area of compression comes from all the concepts you've studied. In addition, when compressing data before transmission, it is much more beneficial as you end up using less bandwidth. Nowadays, compression is used in almost all areas of telecommunications, transmissions, audio and video streaming, etc.

Q: Do you think that compression algorithms has increased its presence in these areas?
A: I believe so, one primordial example is video. Nowadays, there is more bandwidth to transmit video. But on the other hand, there is a greater demand for video streaming, with 4K and High-definition qualities, for instance. 10 years ago, no one in the industry dreamed of streaming the number of videos today with such a high quality. Furthermore, this trend will tend to grow in future years.

Q: What do you think are the new trends in the field of compression? Will there be any new algorithms being developed, for example?
A: In the lossless compression section, I don't think there will be any brand-new compression algorithms any time soon. What has been done in recent years is the improvement of existing compression algorithms. New and efficient algorithms are developed over time, which use the concepts of already existing algorithms. In the terms of video compression, there has been a recently new algorithm launched in 2021 which is the VVC algorithm, which allows for almost a double compression ratio than its predecessor, HEVC. The MPEG people are already investigating the successor of the VVC. The reason for this is that companies can set standards on these algorithms and gain royalties on the used algorithms.

Q: How do you think the experiment carried out in this investigation reflect the actual performances of the algorithms? With the ZTSD algorithms having a higher compression ratio than the LZ77?

A: The performances analyzed in the investigation is on par with what is expected on these algorithms indeed.

Q: What about their compression speeds versus their compression ratios tradeoff? Could this have affected the results?

A: Indeed, there is a tradeoff. The longer you take for an algorithm to run, the better its compression ratio will be, and vice versa. There can be issues in hardware, for example, there may be a bottleneck on the local machine and consuming power the algorithm has on the local machine. There can be limitations on battery life and processing power, causing algorithms to be less complex, allowing them to run on regular computers which don't explore the true potential of these algorithms. Such technologies and scenarios can affect the tradeoffs that may occur, both in the speed of the algorithm, and the quality needed of the algorithms.

Q: Beyond physical limitations, would you say there are other limitations which affect the performance of these algorithms?

A: Of course, Shannon has theorized many limits to the compression algorithms which cannot be surpassed, such as the Shannon limit.

Figure 2: https://www.ijesit.com/Volume%204/Issue%203/IJESIT201503_06.pdf - Study of LZ77 and LZ78 Data Compression Techniques. Choudhary, S. Patel, A. Parmar, S. Published: 05/2015. Accessed: 12/03/2022

Figure 3: https://www.programiz.com/dsa/huffman-coding - Huffman coding and Huffman trees example. Programiz. Accessed 04/09/2022.

Figure 4: https://www.programiz.com/dsa/huffman-coding - Huffman coding and Huffman trees example. Programiz. Accessed 04/09/2022.

Figure 5: https://www.programiz.com/dsa/huffman-coding - Huffman coding and Huffman trees example. Programiz. Accessed 04/09/2022.

Figure 6: https://www.programiz.com/dsa/huffman-coding - Huffman coding and Huffman trees example. Programiz. Accessed 04/09/2022.

Figure 10: Python Script – Written by author using 2019 MacBook Pro with 2.4 GHz Intel Core i5 Processor and 8GB DDR4 Random Access Memory. Text Editor: Visual Studio Code.

Appendix

Controlled Variables

Variable	Detailed Description
Hardware and Software Utilized	2019 MacBook Pro 2.4 GHz Intel Core i5 Processor 8GB DDR4 Random Access Memory
Keys: Size and Specifics	Size: 5-character length Specifics: "Alter", "Bravo", "Count", "Delta", "Enter", "Fetch", "Grade", "Hotel", "India", "Julie", "Kilos", "Lemon", "Menlo", "Night", "Oscar", "Panda", "Quack", "Romeo", "Sound", "Tango", "Unity", “Vitor", "Wooly", "X-ray", “Yanks", "Zebra"
Environment Used to Execute Experiment	2019 MacBook Pro Kernel: Terminal Application [14].
Data type for compression	Strings
Algorithm used for compression	LZ77 and ZSTD

Number of Dictionaries	Uncompressed Length					
	Trial 1	Trial 2	Tiral 3	Trial 4	Trial 5	Mean
200	49788	47038	47804	47685	48164	48096
400	94801	94659	95386	93099	94736	94536
600	144295	145159	142014	143132	143965	143713
800	189694	193788	189439	190790	188707	190484
1000	234811	237185	238781	236859	238998	237327
1200	285586	291097	284844	287592	281432	286110
1400	337754	335251	333291	333488	333494	334656
1600	379704	380515	384600	380969	377146	380587
1800	427190	430802	432451	430746	427920	429822
2000	475332	480238	479159	474928	473545	476640

Raw Table 1: Trials of varying size of text blocks with number of dictionaries. (Made by author with Microsoft Excel)

LZ77	Compressed Length					
Number of Dictionaries	Trial 1	Trial 2	Tiral 3	Trial 4	Trial 5	Mean
200	17497	16596	16684	16840	16975	16918
400	32032	32011	32219	31515	32015	31958
600	47954	48303	47347	47641	47853	47820
800	62758	63956	62620	62957	62324	62923
1000	77398	77958	78356	77980	78530	78044
1200	93561	95095	93395	94318	92105	93695
1400	110070	109618	108951	108951	108994	109317
1600	123579	124050	125438	124219	123083	124074
1800	139239	139974	140535	139993	139273	139803
2000	154591	155932	155498	154305	153742	154814

Raw Table 2: Trials of compressed length for LZ77 algorithm (Made by author with Microsoft Excel).

LZ77	Compression Ratio					
Number of Dictionaries	Trial 1	Trial 2	Tiral 3	Trial 4	Trial 5	Mean
200	2.84551637	2.83429742	2.83131959	2.83165083	2.83734904	2.83602665
400	2.95957168	2.95707725	2.90551228	2.95411709	2.95911292	2.94707824
600	3.00902949	3.00517566	2.99942974	3.00438698	3.00848432	3.00530124
800	3.0226266	3.03002064	3.02521559	3.03048112	3.02783839	3.02723647
1000	3.03471666	3.04247159	3.04738629	3.03743268	3.04339743	3.04108093
1200	3.05240431	3.06111783	3.0498849	3.04917407	3.05555616	3.05362745
1400	3.0685382	3.0583572	3.05909078	3.06089894	3.05974641	3.06132631
1600	3.07256087	3.67432487	3.06605654	3.06391408	3.06415996	3.18820326
1800	3.0680341	3.07728721	3.0771765	3.07691099	3.07252662	3.07438708
2000	3.07477149	3.07979119	3.08144799	3.07785231	3.08012775	3.07879815

Raw Table 3: Trials of compression ratio of LZ77 algorithm (Made by author with Microsoft Excel).

ZSTD	Compressed Length					
Number of Dictionaries	Trial 1	Trial 2	Tiral 3	Trial 4	Trial 5	Mean
200	11583	10966	11104	11130	11242	11205
400	21620	21690	21794	21351	21685	21628
600	32490	32771	32058	32302	32497	32424
800	42643	43437	42645	42974	42489	42838
1000	52727	53219	53603	53196	53614	53272
1200	64010	65121	63882	64441	63003	64091
1400	75492	74957	74490	74621	74512	74814
1600	84752	85134	86011	85120	84262	85056
1800	95457	96142	96578	96169	95714	96012
2000	106223	107163	107030	105950	105659	106405

Raw Table 4: Trials of compressed length for ZSTD algorithm (Made by author with Microsoft Excel).

ZSTD	Compression Ratio					
Number of Dictionaries	Trial 1	Trial 2	Tiral 3	Trial 4	Trial 5	Mean
200	4.2983683	4.28944009	4.30511527	4.28436658	4.28429105	4.29231626
400	4.38487512	4.36417704	4.37670919	4.36040466	4.38734148	4.37470150
600	4.44121268	4.42949559	4.42990829	4.4310569	4.43010124	4.43235494
800	4.44842061	4.46135783	4.44223238	4.23966119	4.44131422	4.40659725
1000	4.4546627	4.45677296	4.45462008	4.45257162	4.45775357	4.45527618
1200	4.46158413	4.47009413	4.45890861	4.46287302	4.46696189	4.46408436
1400	4.47403698	4.47257761	4.47430528	4.46909047	4.47570861	4.47314379
1600	4.48017746	4.46960086	4.47152108	4.47566964	4.47587287	4.47456838
1800	4.47520873	4.48089285	4.4777382	4.4790525	4.47081932	4.47674232
2000	4.47485008	4.48137883	4.4768663	4.48256725	4.4818236	4.47949721

Raw Table 5: Trials of compression ratio of ZSTD algorithm (Made by author with Microsoft Excel).

```python
import ujson
import zstd
import lz4.block
from random import choice
from random import randint

keys = (
    "Alter", "Bravo", "Count", "Delta", "Enter", "Fetch", "Grade", "Hotel", "India",
    "Julie", "Kilos", "Lemon", "Menlo", "Night", "Oscar", "Panda", "Quack", "Romeo",
    "Sound", "Tango", "Unity", "Vitor", "Wooly", "X-ray", "Yanks", "Zebra"
)

data = {}

for i in range(2000):
    d = {}
    for k in range(randint(10, 100)):
        d[choice(keys)] = k
    data[str(i)] = d


dataset = ujson.dumps(data).encode()


data = zstd.compress(dataset, 6)
zstd.compress(dataset, 6)
print("Test data has length", len(dataset))
print("Compressed data length for Zstandard is", len(data), ", ratio is", len(dataset) / len(data))

data = lz4.block.compress(dataset)
lz4.block.compress(dataset)
print("Compressed data length for LZ77 is", len(data), ", ratio is", len(dataset) / len(data))
```

Figure 10: Python program developed for experimentation (Written by author).