International Journal of Engineering &Technology, 8 (3) (2019) 169-174

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Analysis and Comparison of Data Compression Algorithms

Shrenik Nambiar [1] *, H Srikanth Kamath [1]

[1] Manipal Academy of Higher Education
**Corresponding author E-mail: srikanth.kamath@manipal.edu*

Abstract

The amount of data being shared over the internet is increasing exponentially. In this digital age, where even devices like refrigerators are connected, data needs to be stored in compressed form. The compressed data should be retrieved without loss of information else the data will be deemed corrupt. As we are approaching 5G communication, the data need to be transferred over the internet at a higher rate. This cannot be achieved by older compression algorithms which has lesser compression ratio and even lesser compression and decompression speed. In this paper, an analysis of modern compression algorithms along with some older compression algorithms have been conducted. Also the implementation and comparison is conducted. The comparison was done with the help of graphs plotted using MATLAB software. The compression algorithms compared were Deflate, bzip2, Zstandard, Brotli, LZ4 and LZO. The files used for compression were taken from Canterbury and Silesia Corpus.

Keywords: *Compression; Compression Ratio; MATLAB; Compression Speed; Decompression Speed; Canterbury Corpus; Silesia Corpus*.

1. Introduction

Data Compression is becoming relevant over the most recent 20 years. Both the amount and the nature of research in this field gives sufficient evidence. The requirement for compacting information has been felt before. There are numerous known strategies for data compression. They depend on various thoughts, are appropriate for various kinds of information, and deliver diverse outcomes, however they are all in view of a similar rule, pack information by removing excess from the source record. Any nonrandom gathering information has a few structure, and this structure can be misused to accomplish a smaller portrayal of the information, a portrayal where no structure is detectable. The terms excess and structure are utilized as a part of expert writing, and additionally smoothness, cognizance, and connection; they all allude to a similar thing. In this way, excess is a vital idea in an data compression. [6].
The possibility of compression by reducing redundancy, proposes the general law of data compression, which is to ”appoint short codes to basic information (images or expressions), long codes to uncommon occasions.” There are numerous approaches to execute this law, and an investigation of any compression technique demonstrates that, it works by complying with the general law. Packing data is finished by changing its portrayal from wasteful (i.e., long) to productive (short). The reason that wasteful (long) data portrayals are utilized is that they make it less demanding to process the data, and data preparing is more typical and more essential than data compression. [6]
The ASCII code for characters is a decent case of data representation, that is longer. It utilizes 7-bit codes on the grounds that settled size codes are simple to work with. A variable-measure code, be that as it may, would be more effective, since certain characters are utilized more than others thus could be relegated shorter codes.

2. Previous work

Data compression goes way back compared to digital technology. Morse code, the primary objective of this algorithm was to use short codes for characters occurring at higher frequencies so that the redundant data can be reduced was developed in 1838. This procedure came to be known as Data Compression. Although in 1949 Shannon- Fano coding came into existence it was during the late 1970s that data compression started to play an important role in the field of computing as the Internet came to the fore. With information required to be stored digitally increasing at a rapid rate, compression is of paramount importance.
Shannon-Fano coding is a way of assigning codes for character occurring with respect to the frequency of occurrence. Later Huffman came up with another algorithm which was more efficient as compared to Shannon-Fano Coding. The primary difference being that in Huffman code the tree was built top-down instead of bottom up. In 1977 Abraham Lempel and Jacob Ziv came up with the LZ77 algorithm which even now is used in modern compression algorithm. It was the first algorithm which made use of a dictionary for data compression and uses sliding window for dynamic dictionary compression. In the following year they came up with LZ78 which used a static dictionary.
Almost all the algorithms were derived from LZ77 as LZ78 was patented by Sperry and started suing people who used it without the license. Due to patent issues the UNIX community started looking towards open source algorithms like Deflate and Burrows Wheeler

Transform based on bzip2 format. Although the patent expired in 2003, the community stuck with these two algorithms as they were found having better compression ratios. Therefore LZW ended up being used only for GIF compression.
Bandwidth were expensive and limited and to get rid of this bottleneck, compression was very important. Also to meet the demand of World Wide Web, new compression file formats were developed like ZIP, GIF and PNG. ARC was the first successful archive format released in terms of profit in 1985. It uses a slight modification of LZW algorithm. Due to its popularity, Phil Katz developed a program in 1987 but was found a copy of ARC and hence had to pay the price. In 1989 he successfully tweaked and ended up with ZIP format and switched to IMPLODE algorithm due to LZW's patent issues. Later Katz went to further make some updates by implementing the DEFLATE algorithm and split volume was added as a new feature. This version is what being used as the .zip files.
For images, Graphic Interchange Format (GIF) was developed using LZW algorithm. Although it had patent issues, CompuServe were able to escape as Unisys were not able to stop the format.
Still CompuServe, after the success of GIF decided to develop a new format named Portable Network Graphics (PNG). It used DEFLATE algorithm despite Katz patenting it.
A new archiver WinRAR was released by Eugene Rochal using the RAR format in 1993. It was implemented using PPM and LZSS algorithms. It quickly become the standard for file sharing over the internet. Bzip2 also rapidly took over from DEFLATE based gzip format when it was introduced in 1996. 7-Zip was the first format to challenge the predominance of ZIP and RAR be- cause of its high compression ratio and the format's measured modularity and receptiveness. This format isn't restricted to utilizing one compression algorithm, yet can rather pick between bzip2, LZMA, LZMA2, and PPMd algorithms among others. At last, on the cutting edge of authentic programming are the PAQ* formats. The main PAQ format was developed by Matt Mahoney in 2002, called PAQ1. PAQ significantly enhances the PPM algorithm by utilizing a procedure known as context mixing which consolidates at least two models to create a superior prediction of the following symbols than both of the models. [9].

3. Methodology

A brief discussion of the algorithms is done in this section. The modern algorithms are basically a combination of the older ones like Huffman coding and LZ77.

3.1. Data compression algorithms

3.1.1. Deflate

LZ77 and Huffman coding is combined to come up with this compression. Basically with the help of this algorithm, compression is done block-wise. The duplicate substrings can be compared along a distance of up to 32K bytes whereas the lengths are limited to 258 bytes. For compression, tokens having three fields are formed namely, offset (which is the distance of substring repetition), length of the repeated substring and the next symbol. Usually the last field is not required when there is a match but is used when there is no match found.
In LZ77, the text to be compressed is divided into look ahead buffer and search buffer. The search buffer is the portion of text which is already compressed and the look ahead contains the text to be compressed. Search buffers are limited to 32K bytes in length. When the token is formed it uses the Huffman table to write that particular binary representation into the compression stream. Token cannot be written into compression stream. There are two Huffman tables, one for the length/literal and the other one is for the distance. Looking at this two fields we can determine the length and the distance while decompression. [1]
Copied strings are discovered utilizing a hash table. All info strings of length 3 are embedded in the hash table. A hash record is registered for the following 3 bytes. In the event that the hash chain for this file isn't vacant, all strings in the chain are contrasted and the present information string, and the longest match is chosen. The hash chains are sought beginning with the latest strings, to support little separations and in this way exploit the Huffman encoding. The hash chains are separately connected. There are no cancellations from the hash chains, the calculation basically disposes of matches that are excessively old. To stay away from a more pessimistic scenario, long hash chains are discretionarily truncated at a specific length, controlled by a runtime alternative (level parameter of deflateInit). So empty() does not generally locate the longest conceivable match but rather by and large finds a match which is long enough. Flatten() likewise concedes the choice of matches with a sluggish assessment component. After a match of length N has been found, empty() scans for a more drawn out match at the following information byte. On the off chance that a more extended match is discovered, the past match is truncated to a length of one (therefore delivering a solitary exacting byte) and the procedure of sluggish assessment starts once more. Some- thing else, the first match is kept, and the following coordinate hunt is endeavored just N steps later.

3.1.2. Bzip2

The critical algorithm of BZIP2 is the Burrows Wheeler change (BWT) that changes over the first information into an appropriate frame for following coding. The present adaptation applies a Huffman code. In prior periods of the advancement the more productive number juggling coding was utilized. This is limited by licenses and isn't appropriate for open source ventures. Moreover BZIP2 contains alternatives for a run length encoding (RLE), which is obsolete mean-while. The achievable rate of BZIP2 is significantly better in contrast with traditional configurations e.g. Flatten or Deflate64 (GZIP, ZIP) and marginally underneath the best cutting-edge strategies accessible (PPM, LZMA). Looking at the memory limits these days accessible, the use for the pressure of under 8 MB is generally little, however it is obviously bigger than Deflate. The inward algorithm forms the information in pieces being absolutely autonomous from each other. The square sizes can be set in a range from 100,000 - 900,000 byte (1-9); standard are pieces of 900,000 images. Littler piece sizes diminish the memory necessities. [11]
The inside calculation forms the information in squares being absolutely autonomous from each other. The square sizes can be set in a range from 100,000 - 900,000 byte (1-9); standard are pieces of 900,000 images. Littler piece sizes decrease the memory necessities. The handling speed is relative to the achievable compression and to practically identical methods (slower than Deflate, faster than PPM).

3.1.3. Zstandard

Zstandard is a real-time compression algorithm, providing high compression ratios. There is a requirement for a "look-into table, for example, the one utilized by the "quick mode". The present default size of this table is right now chose at 128 KB, yet this too will be

configurable, from as low as a couple of KB to a couple of MB. More grounded seek algorithms will require more tables, subsequently more memory. [12] While such speed qualify Zstandard as a ”quick compressor/decompressor”, regardless it doesn’t achieve LZ4 region. In this manner, choosing which algorithm best suits your need exceedingly relies upon the speed objective. Another property Zstandard is produced for is configurable memory prerequisite, with the target to fit into low-memory designs, or servers dealing with numerous associations in parallel. On the translating side, Zstandard memory prerequisite is isolated into 2 principle parts. FSE needs a few change tables, which as of now cost 10 KB. The expectation is to make this size configurable, from at least 2.5 KB to a most extreme of 20 KB. This is moderately mellow necessity, generally fascinating for frameworks with extremely constrained memory asset. The match window measure, which is fundamentally the span of "think back cradle" decompression side must keep up with a specific end goal to finish "coordinate duplicate" tasks.
Fundamental thought is: the bigger the window estimate, the better the pressure proportion.
Nonetheless, it likewise expands memory necessity on the interpreting side, so an exchange off must be found. Current default window measure is 512 KB, yet this esteem will be configurable, from little (KB) to expansive (GB), in the desire to fit various situations needs.
The compression necessities to deal with a couple of more memory fragments, the number and size of which is exceptionally subject to the chose seek calculation. At the very least, there is a requirement for a "look-into table, for example, the one utilized by the "quick mode". The present default size of this table is right now chose at 128 KB, yet this too will be configurable, from as low as a couple of KB to a couple of MB.

3.1.4. Brotli

Brotli likewise utilizes the LZ77 and Huffman calculations for pressure. The two calculations utilize a sliding window for backreferences. Gzip utilizes a settled size, 32KB window, and Brotli can utilize any window estimate from 1KB to 16MB, in forces of 2 (short 16 bytes). This implies the Brotli window can be up to 512 times bigger window than the empty window. This distinction is relatively insignificant in web-server context, as content records bigger than 32KB are the minority. [7]
Brotli additionally includes a static dictionary. The”dictionary” bolstered by collapse can enormously enhance pressure, yet must be provided freely and must be tended to as a major aspect of the sliding window. The Brotli dictionary is a piece of the execution and can be referenced from anyplace in the stream, to some degree expanding its productivity for bigger documents. Besides extraordinary changes can be connected to expressions of the dictionary viably expanding its size. Brotli additionally bolsters something many refer to as context displaying. Context demonstrating is a component that permits different Huffman trees for a similar letter set in a similar square. [7]
Different contrasts incorporate littler insignificant match length (2 bytes least in Brotli, contrasted with 3 bytes least in empty) and bigger maximal match length (16779333 bytes in Brotli, contrasted with 258 bytes in flatten). Brotli likewise includes a static dictionary. The dictionary upheld by empty can significantly enhance compression, however must be provided autonomously and must be tended to as a major aspect of the sliding window. The Brotli word reference is a piece of the execution and can be referenced from anyplace in the stream, to some degree expanding its effectiveness for bigger records. Besides extraordinary changes can be connected to expressions of the word reference adequately expanding its size.

3.1.5. LZ4

The”quick” rendition of LZ4 facilitated on Google Code utilizes a quick output procedure, which is a solitary cell wide hash table. Each situation in the information piece gets”hashed”, utilizing the initial 4 bytes (min match). At that point the position is put away at the hashed position. The measure of the hash table can be adjusted while regarding full configuration similarity. For confined memory frameworks, this is a vital element, since the hash size can be diminished to 12 bits, or even 10 bits (1024 positions, requiring just 4K). Clearly, the smaller the table, the more crashes (false positive) we get, diminishing pressure adequacy. Yet, it in any case still works, and remain completely perfect with more perplexing and eager for memory forms. The decoder couldn’t care less of the strategy used to discover coordinates, and requires no extra memory. [13]
An estimation of 15 in both of the bit fields demonstrates that the length is bigger and there is an additional byte of information that will be added to the length. An estimation of 255 in these additional bytes demonstrates that yet another byte to be included. Consequently subjective lengths are spoken to by a progression of additional bytes containing the esteem 255. The series of literals comes after the token and any additional bytes expected to show string length. This is trailed by a counterbalanced that shows how far back in the yield sup- port to start replicating. The additional bytes (assuming any) of the match-length come toward the finish of the succession

3.1.6. LZO

Decompression is simple and very fast. It requires very less memory for decompression. Compression speed is a little less as compared to LZ4. Requires 64 kB of memory for compression and allows us to increase compression ratio in exchange of speed. The speed of the decompressor is not reduced. Includes compression levels for generating pre-compressed data which achieve a quite competitive compression ratio. Supports overlapping compression and in-place decompression. Algorithm is thread safe and lossless. [3]
This has a fascinating result: In a lot of cases it pays off to LZO-pack your information, even briefly. With current circle and CPU speeds, it for the most part pays off to LZO-pack in- formation before composing it and decompress it subsequent to perusing: The season of perusing the compacted information from the circle and uncompressing it is by and large not as much as an opportunity to peruse the uncompressed in- formation from the plate. Something else LZO (and the gzip- like channel program 'lzop') is extraordinary for is moving vast bits of information over the system. Indeed, even with present day CPUs and gzip regardless you are near the point where gzip may wind up being a bottleneck on a 100 Mbit/s organize. Doing LZO pressure on the sending end and decom- pression on the less than desirable end, then again, is generally an unadulterated win, unless your information is totally un- compressible, in which case it barely has any kind of effect.

4. Results and analysis

All six algorithms were benchmarked on Intel(R) i5-4210U quad- core processor (clock speed of 1.70 GHz), 4GB of RAM and running Ubuntu 16.04 LTS.
For analysis purpose six files present in the Silesia Corpus have been used for comparison with respect to compression ratio, com- pression and decompression speed.

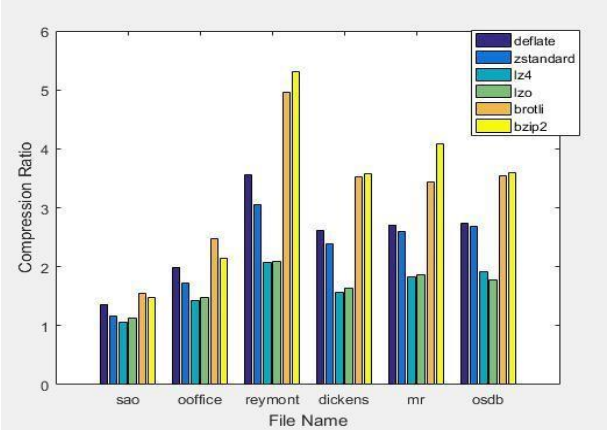

Fig. 1: Comparison of Compression Ratio.

From the graph in figure 1 we can observe that bzip2 along with brotli provides us with the highest compression ratio. The pdf filetype, reymont is compressed more compared to other files in the corpus.

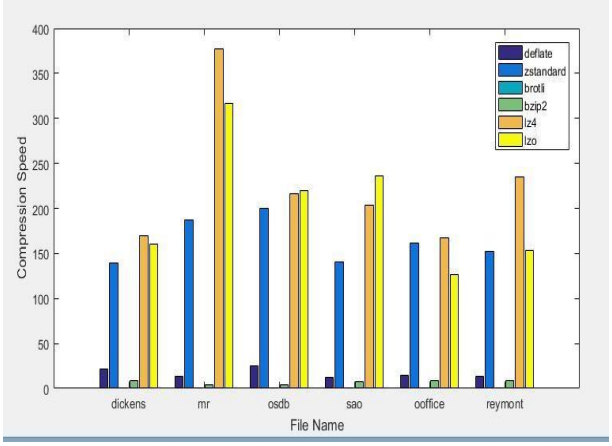

Fig. 2: Comparison of Compression Speed.

LZ4 has the highest compression speed, followed by LZO algorithm. The algorithm compensates it's low compression ratio with its high compression speed. The algorithm with the lowest com- pression speed is brotli. From the figure 2 we can also come to the conclusion that images were compressed faster compared to other files.
Again from figure 3 we can see LZ4 has highest decom- pression speed followed by Zstandard algorithm. Bzip2 has the lowest decompression speed.
We also used the Canterbury Corpus which had 10 files of varying sizes and filetypes. We divided the files into two groups of five each so that the graph is not clustered. In figure 4 and figure 5, the brotli has higher compression ratios with the only exception being that bzip2 has highest compression ratio for text files. Therefore we can conclude that bzip2 works better for text files whereas brotli works best for non .txt files.

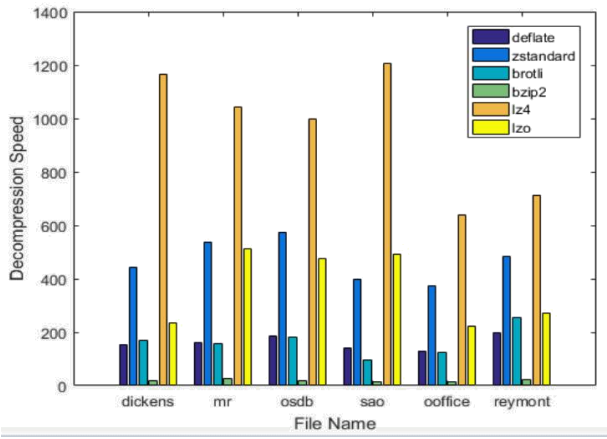

Fig. 4: Comparison of Decompression Speed.

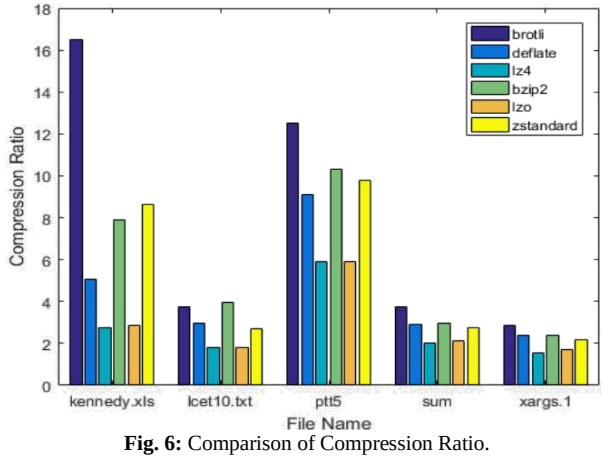

Fig. 6: Comparison of Compression Ratio.

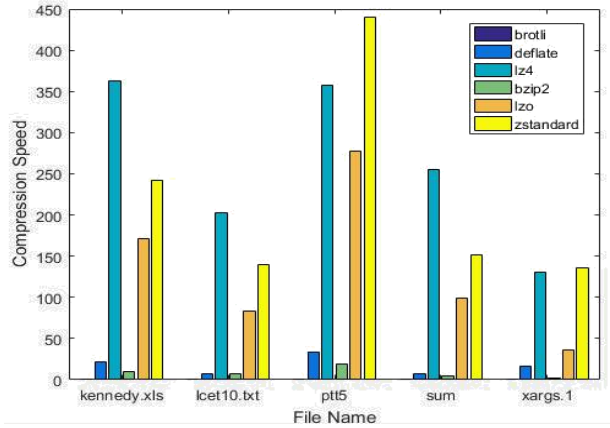

Fig. 3: Comparison of Compression Speed.

From figure 6 and figure 7, LZ4 and Zstandard algorithms have the highest compression speed followed by LZO. Taking both compression ratio and speed into consideration, zstandard is the best compression algorithm because not only does it have a very high compression speed but also high compression ratio.

4.1. Implementation

The source code for each algorithm except deflate (zlib format) were taken from their respective Github repositories. The deflate algorithm in zlib format was already available in Python 2.7.
The source code downloaded were build using the build command and scripts were developed for each algorithm. The python scripts were run from the terminal and the parameters value were noted.
In order to give a graphical view of the end results the values not- ed was used in MATLAB to obtain the graphs.

4.2. Compression parameters

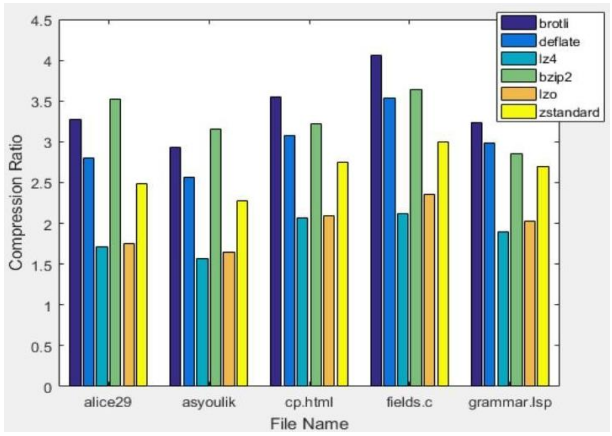

Fig. 5: Comparison of Compression Ratio.

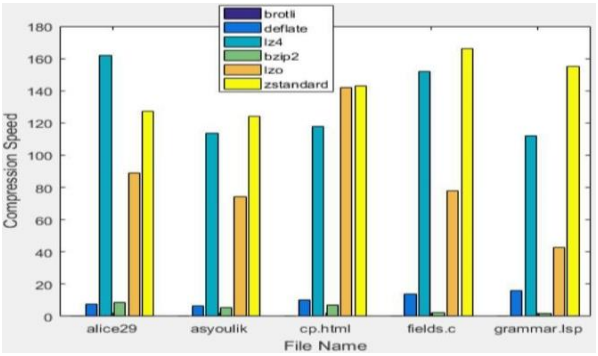

Fig. 7: Comparison of Compression Speed.

4.2.1. Compression Ratio

Data needs to be compressed be- fore it is transferred to its destination. Transferring the raw size would waste network bandwidth, which will result in more traffic. Therefore the compression algorithm should be able to reduce the data size as low as possible before transferring.
Compression ratio is basically the uncompressed data di- vided by compressed data. The value is always greater than 1 Among the six algorithms we tested, bzip2 has the highest compression ratio and LZ4 having the least.

4.2.2. Compression speed

The data needs to be compressed quickly. With the increasing data which needs to be transferred there should be as low latency as possible. If the algorithm takes more time to compress or decompress data, then it will greatly slow down the network. In order to cope up with the latest surge in data storage and consumption, new algorithms like LZ4, LZO and even Zstandard have very high compression speed.

4.2.3. Decompression speed

When the data is compressed, we need to decompress the data to get back the original content. This also needs to be done quickly. LZ4 and Zstandard have the highest decompression speed among the algorithms we benchmarked.
Decompression speed is the amount data that can be de- compressed per second.

5. Conclusion and future work

An analysis and comparison of six different compression algorithm namely deflate, brotli, Zstandard, bzip2, LZ4 and LZO were done. Among these, bzip2 was found to have the highest compression ratio according Silesa Corpus whereas Canterbury corpus [4] gives a clearer picture by showing bzip2 works better for text files and for other filetypes brotli has the higher compression ratio. LZ4 has the highest compression and decompression speed but has the lowest compression ratio. Zstandard was found to be having a good compression ratio along with high compression and decompression speed.
The test was conducted with files taken from the Silesia Corpus [2], which contains a wide variety of file types. We gave six different files namely, dickens (text), sao and osdb (database), reymont (polish text pdf), ooffice (executable) and mr (image) as input. [2] The file sizes ranges from 5MB to 10MB.
Calgary [5] can also be used to test the algorithms. Only Silesia Corpus is used in this paper. In this age of Big Data, better compression algorithms needs to be developed with higher compression ratio and compression and decompression speed.

Acknowledgement

We would like to thank our department lab coordinator who let us work in the lab.

References

[1] P. Deutsch (1996) DEFLATE Compressed Data Format Specification version 1.3 [Online]. Available: https://tools.ietf.org/html/rfc1951. https://doi.org/10.17487/rfc1951.
[2] The Silesia corpus, http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia.
[3] LZO Compression library, https://boutell.com/lsm/lsmbyid.cgi/001070.
[4] The Canterbury corpus, http://corpus.canterbury.ac.nz/.
[5] The Calgary corpus, http://www.data- compression.info/Corpora/.
[6] David Solomon, Data Compression, The Complete Reference, 3rd edition, Springers Publication, 2003.
[7] Vlad Krasnov, Results of experimenting with Brotli for dynamic web content, https://blog.cloudflare.com/results- experimenting-brotli/
[8] Fano, R.M."The transmission of information". Technical Report No. 65, USA: Research Laboratory of Electronics at MIT, 1949.
[9] Ziv, Jacob; Lempel, Abraham. "A Universal Algorithm for Sequential Data Compression". IEEE Transactions on Information Theory23 (3): pp. 337343, May 1977. https://doi.org/10.1109/TIT.1977.1055714.
[10] Huffman, D." A Method for the Construction of Minimum- Redundancy Codes". Proceedings of the IRE40 (9): pp. 10981101, 1952. https://doi.org/10.1109/JRPROC.1952.273898.
[11] Michael Burrows, David Wheeler, A Block-sorting Lossless Data Com-pression Algorithm, Digital Systems Research Center, Research Report 124, 1994.
[12] Zstandard A Stronger Compression Algorithm, http://fastcompression.blogspot.in/2015/01/zstd-stronger- compression-algorithm.html.
[13] LZ4 explained, http://fastcompression.blogspot.in/2011/05/lz4- explained.html.