Huffman Coding Implementation on Gzip Deflate Algorithm and its Effect on Website Performance

I Putu Gede Wirasuta - 13517015
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
wirasutat@gmail.com 13517015@std.stei.itb.ac.id

***Abstract*—Huffman coding is a lossless compression commonly used as backbone of modern compression standard. One of them being deflate algorithm commonly used in compression software such as zip and gzip. The need for high performance and fast compression rises as the demand for higher performance website rises. In this paper, the author will discuss the implementation of Huffman coding in modern compression algorithm and the effect on compressed website sources relying on it.**

***Keywords*—Huffman coding, GZIP, Deflate Algorithm, Website Compression**

I. INTRODUCTION

Computers have seen it's fair growth in recent years, and along with it the average size of digital data and the need to transfer those data. The current median web page size is 1.6MB [1] and increasing each day. Yet, more users are demanding faster loading time for a website. This becomes a problem when file size gets too big beyond the network capabilities to transfer the data in comfortable time reliably. One solution to this problem is by reducing the source code size by compressing it.

Data compression is an act of reducing the size of one or more file so that it requires less disk space or Internet bandwidth to transfer. To compress file, multiple techniques is available based on different mechanism to attain different results. In general, data compression algorithm is divided in two categories, *lossless* and *lossy* compression.

One of the simplest *lossless* compression algorithm is Huffman coding, introduced by David A. Huffman in his 1952 paper. The Huffman coding make use of statistical information of the data to be compressed and construct a compact version of its symbols based on a corresponding tree scheme, therefore reducing the overall data size.

Due to its simplicity and efficiency, Huffman coding is often used with other compression algorithm to produce an even better compression ratio and speed. Deflate is a file format and compression algorithm that combines Huffman coding with LZ77 compression algorithm to achieve a better compression ratio. Deflate was designed by Phil Katz and specified in RFC 1951 [2]. Other than its own file type, Deflate algorithm is also used in many other file types such as PNG and GZIP file format. Most modern browser [3] support compressed source in gzip format.

This paper will discuss Huffman coding implementation on deflate algorithm, which is used on gzip file format and observe its effect on website performance.

II. DATA COMPRESSION ALGORITHM

Data compression is the process of altering a structure of data in such a way so that it occupies less disk space or Internet bandwidth. In a paper titled "A Mathematical Theory of Communication" published in 1948, Shannon describes efficient way of representing data and its limit, which later become the foundation of data compression theoretical background. Data compression works on reducing redundancies by replacing it with shorter and specific data or removing it [4]. By this principle, data compression can be categorized into *lossless* and *lossy* compression.

Lossless compression is data altering process that reduce the amount of needed space without losing information. A lossless compressed file can be reconstructed perfectly into its original data. The ability to reconstruct the original data perfectly makes lossless compression suitable for text-based data compression (including software) and media archival. However, lossless compression has a very strict compression rate defined by Shannon's source coding theorem [5].

Lossy compression on the other hand, is data altering process that reduce the amount of needed space with assumption that the data doesn't need to be reconstructed as perfectly after compression. Lossy compression is commonly used on media files, because on certain level of compression, the compressed data is undistinguishable to human eyes. Lossy compression works by removing parts of unneeded data while maintaining the "visible" part. Thus, lossy compression can attain far greater compression ratio.

The argument on which compression type is better is pointless, as it is not an apple to apple comparison. Each type has its own strength and weaknesses, also its appropriate usage. In fact, some lossy compression algorithm use lossless compression technique to achieve greater compression ratio.

There are two essential figure to a compression, compression ratio and compression time. For example, an executable binary file is 100KB large. Using certain type of compression algorithm it can be reduced to only 20KB. In this example, the compression ratio is 5:1 or 20% [6]. Compression ratio vary because different algorithm performs differently on the same file and the same algorithm performs differently on different file. Each data compression algorithm has its own proper use. Compression time is the time needed to compress a file. The longer an analysis is done to compress a file, the more it is possible to reach optimum compression. Therefore, generally the relation between compression time and compression is inversely proportional.

III. Huffman Code

Huffman code is an optimal prefix code commonly used for lossless data compression. The process of encoding plaintext to Huffman code is called Huffman coding. Huffman coding was designed by David A. Huffman in 1952 as his term paper on information theory problem to find the most efficient method of representing numbers, letters or other symbols using a binary code [7]. As he was getting desperate on the paper for not solving the problem, the solution came to him and eventually published in his paper titled “A Method for the Construction of Minimum-Redundancy Codes” [8]. In it he described a method to construct a variable-length code for each input symbol by using multi-step weight/frequency table. Character with highest frequency will have a shorter length code and vice versa. In this paper, the multi-step weight/frequency table will be represented as binary tree with each node containing merged symbols and summed weight.

A. Binary Tree

In graph theory, tree is an undirected and acyclic graph. For any graph T with n vertices that satisfies any of these conditions:

- T has no simple cycles and has n − 1 edges.
- T is connected and has n − 1 edges.
- T is connected, and every subgraph of T includes at least one vertex with zero or one incident edges.

Then T is a tree. A rooted tree is a tree with one of the vertices set as root and all the other vertices is given directions away from the root. Degree is the amount of connected edges, as rooted tree is directed, there exist inward and outward degree. A root is a vertices with zero inward degree. Every vertices in a tree can be reached by a unique path from the root [11].

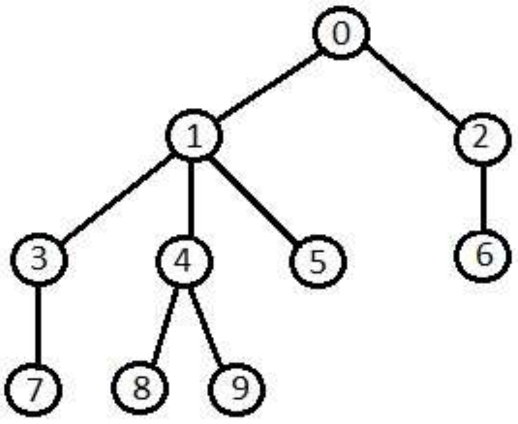


Figure 1

For this paper, there are a few terminologies that needs to be understood:

1. Child and parent. X is the child of Y (Y is the parent of X), iff there is an incident edge going outward from Y and inward to X. In figure 1, 0 is the parent of 1 and 2, while 6 is the child of 2.
2. Path. Path from vertice X to vertice Y is a sequential edges from X to Y thru zero or more vertices. In figure 1, there is a path from 0 to 6 thru 2 edges, or thru vertice 2.
3. Leaf. A Leaf is a vertices with zero outward degree. In figure 1, vertice 7,8,9,5, and 6 are leaves.

Tree is considered one of the most important part of graph because of its uses in graph theory related problems. One specific tree commonly used is binary tree. Binary tree is a tree where every node has at most two outward degree. A binary tree can only have two child, commonly called as left and right child respectively.

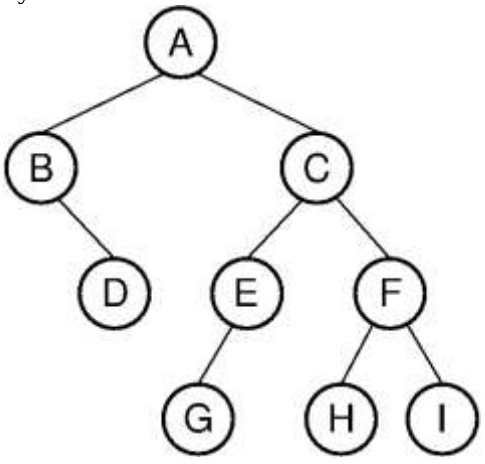


Figure 2

B. Prefix Code

Prefix code is a type of code which requires none of its code word is a prefix of another code word. For example, a code with these words {0,10,11} in binary representation is a prefix code. But {0,1,10,11} in binary representation is not a prefix code, because “1” is prefix of “10” and “11”. Therefore, words on a complete prefix code can be identified without any separator and any incomplete prefix code (terminated before end of code word) is invalid and unidentifiable [9].

C. Huffman Coding Algorithm

Based on [10], Huffman coding process on a plaintext can be done by building a variation of binary tree (Huffman tree) from the symbols frequency table. The algorithm is as following

1. Count the frequency of each input symbol in input text
2. Choose two symbol with the lowest frequency, then create a parent node with symbol equals to the two symbol concatenated and frequency equals to the two symbol's frequency added. This parent node replaces the two building symbols in the frequency table
3. Repeat step 2 until only one symbol remaining which contains all input symbol
4. Label every left branch as 0 and right branch as 1
5. The path from root to a leaf is the Huffman code for the corresponding leaf's symbol

For example, figure 1 is the Huffman tree to code “INFORMATIKA ITB” into Huffman code.

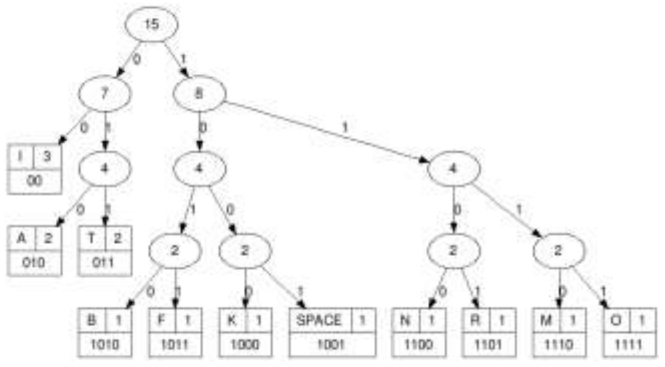


Figure 3

Tracing each path to the leaf of the Huffman tree above yields the following Huffman code

Input	Frequency	Huffman Code
I	3/15	00
A	2/15	010
T	2/15	011
B	1/15	1010
F	1/15	1011
K	1/15	1000
(SPACE)	1/15	1001
N	1/15	1100
R	1/15	1101
M	1/15	1110
O	1/15	1111

Table 1

Notice the shortest Huffman code in table above correspond to input with the highest frequency. The resulting Huffman code also shows no code word is a prefix of another, conforming the prefix code definition.

However, a Huffman tree is not unique. A different placement of node with the same weight is possible, creating a different Huffman tree and code. Thus, before a decompression could be done, the Huffman tree must be reconstructed. For a specific type of input, where input symbol frequencies are more or less the same, the tree can be constructed once and reused. Otherwise, a coded information of the corresponding tree, often called as codebook in decompression process is needed to be sent when transferring a Huffman coded data. There are two main ideas to do this

First, prepending the frequency. This is the easiest way to include information needed to reconstruct the Huffman tree. Unfortunately, tree reconstruction is costly rendering this idea unusable.

Another ide is by prepending the tree. This might sound better after learning that tree reconstruction is memory and time consuming. But, this idea is space consuming because of the need to store every node value, child references, and value assigned to every edges. Moreover, tree traversing is proven to be memory consuming making this idea also unusable for practical use.

The problem of Huffman coding is now apparent. It is not the compression or the decompression technique, rather the means to transfer the codebook for decompression use. One way to overcome this is to standardize the production of Huffman code. One of the most used standard of Huffman code generation is the canonical Huffman code [12].

D. Canonical Huffman Code

The canonical Huffman code addresses the problems of space- and time-consuming decoding process of Huffman code by generating the code in a more standardized format and in linear time. Every code words value is assigned sequentially based on the lengths. This can be done by creating a Huffman tree with respect to the order of the input alphabet. This means, a sorted frequency table of an input is the easiest to be made into canonical Huffman code. Fortunately, any Huffman code can be transformed into canonical Huffman code by this algorithm

1. Sort the input and corresponding code word first by code word length and secondly by input value
2. The first input symbol is assigned code word with the same length as normal Huffman code but all zeros
3. The next input symbol is assigned the next binary number
4. If the next input symbol code word is longer then append zeros until the length match
5. Repeat step 3 and 4 for all input symbol

Using the algorithm above on Table 1, the canonical Huffman code for “INFORMATIKA ITB” is the following

Input	Frequency	Huffman Code
I	3/15	00
A	2/15	010
T	2/15	011
B	1/15	1000
F	1/15	1001
K	1/15	1010
M	1/15	1011
N	1/15	1100
O	1/15	1101
R	1/15	1110
(SPACE)	1/15	1111

Table 2

The advantage of canonical Huffman code is that for the decoding process, it is possible to encode the codebook in fewer bits while still containing the same amount of information. This paper will discuss two ways to encode the codebook of a canonical Huffman code [12].

The first way is to write each number of bits and code word based on the order of the input alphabet. The codebook on Table 2 would be encoded as

$(A, 3, '010'), (B, 4, '1000'),$
$(F, 4, '1001'), (I, 2, '00'), \ldots\ (SPACE, 4, '1111')$

Since the encoding is ordered by the input alphabet, it can be removed from the codebook encoding. Furthermore, because the canonical Huffman code generation algorithm is standardized and can be done in linear time then the code word can be removed from the codebook encoding. Leaving only the code word length need to be encoded to reconstruct the codebook. Then, codebook on Table 2 would be encoded as

$$3,4,4,2,4,4,4,4,4,3,4$$

The second way of encoding the codebook is by writing the input symbols in increasing order based on the code word length along with number of symbols for each code word length. Using this encoding, codebook based on Table 2 would be encoded as

$$(0,1,2,8)\ (I, A, T, B, F, K, M, N, O, R, SPACE)$$

This means, no symbol has code word length of 0, the code word of I has length of 2, the code word of A and T has length of 3, and the rest of the input alphabet has code word length of 4.

IV. Brief Explanation of LZ77

LZ77 is a lossless data compression algorithm designed by Jacob Ziv and Abraham Lempel in their paper "A Universal Algorithm for Sequential Data Compression" published in 1977 [13]. LZ77 is a dictionary coder, meaning it would search the data to be compressed for specific word available on the dictionary then replace the occurrence on the text with reference to the word on the dictionary. In LZ77 algorithm, the dictionary is prefix of the current word up to several kilobytes. Because the dictionary is moving as the word search continues, it is termed as "sliding window".

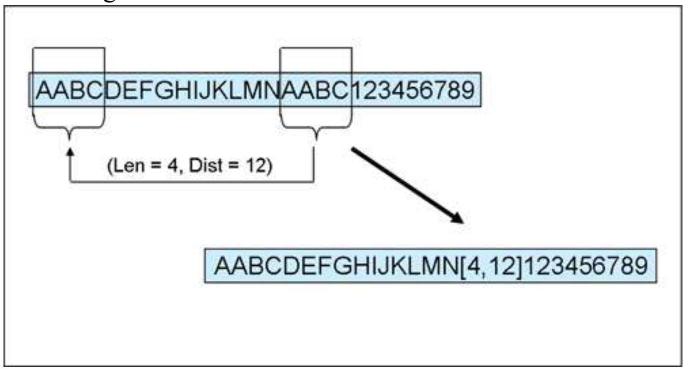


Figure 4

V. Deflate Algorithm

Deflate is a lossless data compression algorithm initially designed by Phil Katz in 1990, and developed further in 1996 and specified by L Peter Deutsch in RFC 1951 [2]. Deflate combine LZ77 and Huffman coding (specifically the canonical Huffman coding) algorithm to achieve compression ratio of 2.5:1 to 3:1 on English text and even more on rasterized images.

A compressed data by deflate algorithm consists of series of blocks. Each block arbitrarily sized and is produced as a result of compression using LZ77 and Huffman coding on the input block. There are four types of block in a deflate compressed data: uncompressed, compressed with predefined Huffman code, compressed with dynamic Huffman code, and reserved/error block. While arbitrary in size, there exist maximum size of an uncompressed block of 65535 bytes.

A. LZ77 Recurring Word Replacement

The LZ77 in deflate algorithm in implemented in such a way that is not patented. It is implemented with sliding windows size of 32KB. The replacement in this implementation is independent from the type of the block compression. The recurring word from dictionary is replaced by 256 bits of distance to reference, 1 bit of end block sign, and 9 bits + extra bits of word length in that order. The duplicated word reference may refer to previous block, but not before the beginning of the input.

B. Huffman Coding

There are two possible Huffman coding implementation in Deflate algorithm with integer ranging from 0 to 285 as the alphabet. The first implementation being fixed Huffman code with specification as the table below

Lit Value	Bits	Codes
0 – 143	8	00110000 through 10111111
144 – 255	9	110010000 through 111111111
256 – 279	7	0000000 through 0010111
280 – 287	8	11000000 through 11000111

Table 3

Block with fixed Huffman code compression are marked with "01" as the first two bits. This fixed Huffman code is an example of specific input type with predictable input symbol frequency.

The second type of Huffman coding implementation is dynamic Huffman coding. This type generates canonical Huffman code for a specific input of 19 symbols with the following specification.

0 – 15	: Represent code lengths of 0 - 15
16	: Copy the previous code length 3 - 6 times. The next 2 bits indicate repeat length (0 = 3, ... , 3 = 6) Example: Codes 8, 16 (+2 bits 11), 16 (+2 bits 10) will expand to 12 code lengths of 8 (1 + 6 + 5)
17	: Repeat a code length of 0 for 3 - 10 times. (3 bits of length)
18	: Repeat a code length of 0 for 11 - 138 times (7 bits of length)

Unlike block compressed with fixed Huffman code, block compressed with dynamic Huffman code must encode its codebook for decompression process. It is encoded as the code word length of each symbols in this order

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

The code word length are interpreted as 3 bit integers (0-7), with 0 as unused symbols. The code word length is encoded not in a sequential order because the less likely used symbols are placed in the end, and removable when unneeded. The encoded codebook is then prepended to the Huffman coded block. Huffman tree of a block can be different than the previous or next block and doesn't take consideration of other symbols outside its own block.

V. Gzip Implementation

Gzip is a lossless compression software and also a file format. It was developed by Jean-loup Gailly and Mark Adler in 1992 to replace UNIX *compress* utility which was patented at that time and later used by GNU project. The compression software is based on deflate algorithm [14]. Other compressed file type that also use deflate algorithm includes zip and 7zip. Gzip produces a compressed file with .gz file extension with file format of the same name.

Gzip file format is comprised of 10 byte header, optional flags, a body containing compressed data using DEFLATE algorithm, and a 8 byte footer of checksum and original file size mod 2^{37} [14].

Gzip being a single file compression software means it is commonly used with archival software. Gzip is used with tar to produce compressed archive format with .tar.gz file extension. The compression is usually better than zip files as it takes advantages of redundancy reduction across multiple files. A typical gzip compression ratio range around 2:1 to 5:1, in accordance with the deflate algorithm it is based on. Table 4 highlight gzip compression performance compared to other type of compression on different level of compression. The performance is calculated as percentage of the compressed file relative to the uncompressed file. The uncompressed file is 445MB large.

lv	gzip	bzip2	lzma	xz	lz4
1	26.8%	20.2%	18.4%	18.4%	35.6%
2	25.5%	18.8%	17.5%	17.5%	35.6%
3	24.7%	18.2%	17.1%	17.1%	35.6%
5	22.0%	17.6%	14.9%	14.9%	-
7	21.5%	17.2%	14.4%	14.4%	-
9	21.4%	16.9%	14.1%	14.1%	-

Table 4 [15]

Performance can also be measured from compression and decompression time. Table 5 shows the same file being compressed by multiple type of compression. While table 6 shows decompression comparison

lv	gzip	bzip2	lzma	xz	lz4
1	8.1s	58.3s	31.7s	32.2s	1.3s
2	8.5s	58.4s	40.7s	41.9s	1.4s
3	9.6s	59.1s	1m2s	1m1s	1.3s
5	14s	1m1s	3m5s	3m6s	-
7	21s	1m2s	4m14s	4m13s	-
9	33s	1m3s	4m48s	4m51s	-

Table 5

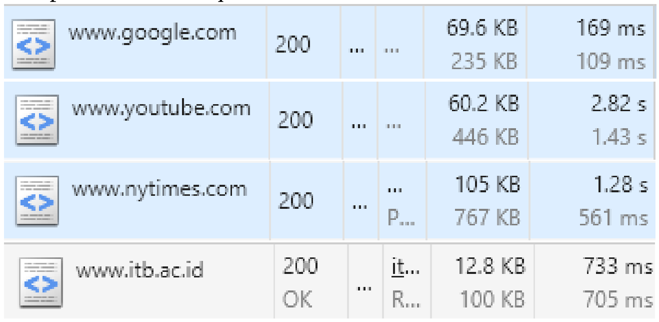
	gzip	bzip2	lzma	xz	lz4
1	3.5s	3.4s	6.7s	7.2s	0.4s
2	3s	15.7	6.3s	6.8s	0.3s
3	3.2s	15.9s	6s	6.7s	0.4s
5	3.2s	16s	5.5s	6.2s	-
7	3s	15s	5.3s	5.9s	-
9	3s	15s	5s	5.6s	-

Table 6

From tables above, it's clear that gzip is not the best in compression ratio, nor it's the fastest to compress and decompress. But, compression ratio is "good" enough and compression and decompression time is only behind the lz4 compressed file type. This balance of compression ratio and time is what makes gzip widely adopted standard of HTTP compression [3].

VI. Effect on Website Performance

Most modern browser has long accept compressed file format, with gzip as the most supported. With the steadily rising trend of website size, HTTP compression has become a necessity. The adoption rate has reach 77% in December 2018, growing 5% from 2017 [16]. In figure 2 below is data transfer statistic from several websites. On the second rightmost column are the data downloaded (numbers on the top) and data size in disk (numbers on the bottom). All of them are using gzip as compression technique.



www.google.com	200	...	...	69.6 KB 235 KB	169 ms 109 ms
www.youtube.com	200	...	...	60.2 KB 446 KB	2.82 s 1.43 s
www.nytimes.com	200	...	... P...	105 KB 767 KB	1.28 s 561 ms
www.itb.ac.id	200 OK	...	it... R...	12.8 KB 100 KB	733 ms 705 ms

Figure 5

Google manage to attain compression ratio of 4:1. While youtube, new york times, and itb's site manage to achieve nearly 8:1 compression ratio. Gzip is able to achieve such high compression ratio because web sources are mostly redundant. Web sources are wrapped in tags with matching opening and closing tag, with only difference being a slash ('/'). Menu in websites are also subject to compression, usually a menu bar is made from multiple unordered list element. HTML and CSS tags are also redundant. This is why gzip with deflate algorithm performs incredibly on compressing web sources.

VII. Conclusion

Gzip uses deflate algorithm which combines LZ77 and Huffman coding to achieve good enough compression ratio while maintaining comfortable compression and decompression time. Web sources are documents with high redundancy and high demand. To improve website performance, lower load times is needed which means lower file size. Compressed gzip file works well on web sources because of high redundancies, achieving up to 8:1 compression ratio.

VIII. Acknowledgment

The author would like to thank God the Almighty for grace and blessings. The author would also thank Dr. Judhi Santoso M.Sc. as the lecturer of discrete mathematics (IF2120). And also gratitude to family and friends for their support in the making of this paper.

References

[1] https://httparchive.org/reports/page-weight (Accesed December 8, 2018)
[2] P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*. Menlo Park, CA. 1996.
[3] http://schroepl.net/projekte/mod_gzip/browser.htm (Accesed December 8, 2018)
[4] Lohit and Jagadish, *A NEW LOSSLESS METHOD OF IMAGE COMPRESSION AND DECOMPRESSION USING HUFFMAN CODING TECHNIQUES*. Hubli, India. 2005.
[5] C.E. Shannon, *A Mathematical Theory of Communication*. 1948.
[6] S. Khalid, *Introduction to Data Compression, Third Edition*. San Fransisco, CA:Elsevier. 2006.
[7] http://www.huffmancoding.com/my-uncle/scientific-american (Accesed December 8, 2018)
[8] Huffman. D, *A Method for the Construction of Minimum-Redundancy Codes*. 1952.
[9] http://www.atis.org/glossary/definition.aspx?id=6416 (Accesed December 8, 2018)
[10] R. Munir, *Matematika Diskrit*. Bandung: Departemen Teknik Informatika Institut Teknologi Bandung. 2003.
[11] T.K. Shmuel, *Space- and Time-Efficient Decoding with Canonical Huffman Trees*. London, UK: Springer. 1997.
[12] Y. Nekritch, *Byte-oriented decoding of canonical Huffman codes*. London, UK: Springer. 2000.
[13] Z. Jacob, *A Universal Algorithm for Sequential Data Compression*. London, UK: Springer. 1997.
[14] https://www.gnu.org/software/gzip/manual/gzip.html#index-options-4 (Accesed December 8, 2018)
[15] https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO (Accesed December 8, 2018)
[16] https://w3techs.com/technologies/details/ce-compression/all/all (Accesed December 8, 2018)

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2018

[illegible]

I Putu Gede Wirasuta - 13517015