Comparative Analysis of CSV Database Backup Storage Optimization Using Huffman Algorithm and Run-Length Encoding

Bayu Palamarta Wirawan - 13525085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: bayupalamartawirawan@gmail.com , 13525085@std.stei.itb.ac.id

***Abstract*—This paper presents a comparative analysis of two optimization methods: the Huffman algorithm and Run-Length Encoding, specifically to compress CSV data storage. This research will compare the performance of such methods' effectiveness and efficiency. The evaluation focuses on time complexity and compression ratio on processing CSV datasets.**

Keywords—Huffman algorithm; Run-Length Encoding; data compression

I. Introduction

Data compression is a process of reducing the size of data by reducing the number of bits required to store the information contained within the data. This process is used to conserve storage space and optimize the use of available storage space. This will help increasing data transfer speed and reducing data redundancy.

One application of the data compression process is the compression of text data, such as CSV files. Two of the many methods available for text data compression are the Huffman algorithm and Run-Length Encoding. Both methods reduce data size without losing information, while maintaining the integrity of the original data. Both can also be applied to CSV files. However, the Huffman algorithm and Run-Length Encoding have different approaches.

In this paper, the author used tree theory and algorithmic complexity to compare the performance of these two algorithms to identify their strengths and weaknesses. The author also identified situations in which one algorithm is more suitable than the other for compressing files.

II. Theoretical Background

A. Data Compression

Data Compression is a process used to reduce the amount of storage needed to store data. For example, a photo with size around a megabyte can be compressed into only several hundreds of kilobytes. This process is quite useful especially for managing files inside a project such as an application. The smaller the data size, the faster the system can transfer them, optimizing the performance of said application.

The two main categories of data compression are lossless and lossy. Lossless data compression are algorithms that can reduce the data size without erasing any of the original data, making it suitable for compression important documents like a backup dataset. The other advantage of this type of compression data is that the compressed data can still be restored to the original size incase it's needed.

Meanwhile, lossy data compression causes a few or more losses of data during the process of compressing it. Even though there is definitely a clear disadvantage, lossy data compression still excels when it comes to reducing redundant data in situations where quality isn't the main priority. For example, compression of MP3, JPEG, and MPEG can be done quite well using lossy data compression.

B. CSV File

CSV File (Comma-Separated Values) is a type of text file that save data like a table. Each line represents the rows of said table, while the commas separate the columns. The first line usually shows the name of each column or variable. It appears to be easier and faster to store and transfer text files by CSV.

The most valuable aspect of CSV file is how it can adapt to multiple situations. The text file format proves to be very convenient since it's compatible with most software and programming languages in multiple OS. CSV file can also be converted into XLSX file, which can be opened via Google Spreadsheet or Microsoft Excel. CSV file is usually used to exchange data, analyze data, and manage the database.

Unfortunately, this type of CSV file can't save data such as font or color. It is also not recommended to use CSV file if there are too many data that need to be store or the data structure is too complex. Several sources can also use different format of CSV files, creating some inconsistencies.

C. Lossless Compression

Lossless compression is a type of data compression where the algorithm still keeps the original value of the data being compressed without erasing any information. This allows a decompression algorithm to restore the data exactly like how it was before. Therefore, lossless compression is beneficial in storing data where exact precision is mandatory, such as computer codes and important documents.

Lossless compression has perfect quality retention with little to no data degradation. However, in some cases, lossless compression can't reduce a file's size as much as lossy compression. On average, lossless compression usually reduces 20% of the original file's size, while a lossy compression can reduce up to 80% of the original file's size.

There are several algorithms to compress data without removing any of the data. Two of them are Huffman algorithm and Run-Length Encoding, which are the topic embraced in this study. Other than those two, there are more lossless compression algorithms, such as:

- Lempel-Ziv-Welch
- Shannon Fano Coding
- Bit Plane Coding
- Arithmetic Coding
- Dictionary Based Coding
- Lossless Predictive Coding

D. Huffman Algorithm

Huffman Algorithm is a lossless data compression algorithm by assigning a code for each character based on the frequency of identical characters inside a data that's going to be compressed. The main idea behind this algorithm is giving the character with the highest frequency compared to other characters with the shortest code. The more a character shows up inside a dataset, the shorter the code assigned to said character.

The codes are made in such a way that each distinct character in a dataset is assigned with different codes. For example, the codes for letter 'a', 'd', 'm' inside the word "adam" are 0, 10, and 11. This method should be able to mitigate any ambiguity when decompressing a file back to its original state. The code to save a character value usually consist of a bunch of '1's and '0's, similar to a binary code. Hence, the code strings can be stored in bits, increasing the compression ratio.

In order to apply Huffman Algorithm to compress a file, we must first create the Huffman Tree, a tree data structure designed to help assigning codes to each character inside a data and avoiding any ambiguity while decoding. The algorithm must read the entire dataset first while saving information on how many times a character shows inside the data. As an example, the word "adam" has two letters 'a', one letter 'd', and one letter 'm'. This information will be saved for the next step of creating the Huffman Tree.

1	1	2
d	m	a

Fig. 1. Arrangement of the Letters Inside "adam"

After collecting the frequency of each character, we take two nodes with the least frequency in the dataset. In this case, it was the letter 'd' and 'm', considering each of them only showed up once. Next, we create an internal node with the frequency equal to the sum of the previous two nodes' frequency. Then, both nodes become the children of the internal node. The graph below is the graph representation of the result of this step. The internal node consists of 'd' and 'm', making the total frequncy becomes two.

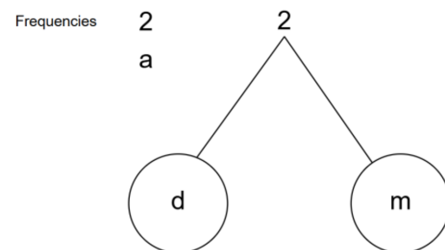

Fig. 2. Result of First Merging

Then, the previous steps of picking two nodes with the least frequencies and combining them into one internal node are repeated until there is only one node left. After that, the final step is assigning 0 and 1 values to each branches and leaves on the tree to complete the Huffman Tree.

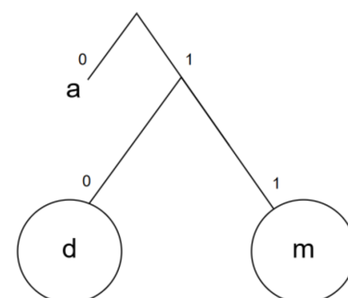

Fig. 3. Full Huffman Tree from the Word "adam"

Based on the tree we just created, we can determine the code of the letter 'a', 'd', and 'm' in the word 'adam':

- Code for letter 'a' : 0
- Code for letter 'd' : 10
- Code for letter 'm' : 11

Therefore the final compressed data is '010011'.

Huffman Algorithm directly manages a file based on the characters inside a data, so it is commonly used to manage and compress text files. However, Huffman Algorithm can also be used to compress other types of files like JPEG, PNG, and MP3.

E. *Run-Length Encoding*

Run-Length Encoding is another type of lossless data compression that utilizes the frequency of characters inside a dataset. In this algorithm, a text will be compressed based on the characters that appear consecutively. For example, in the word "aaabb", the letter "a" appears three times consecutively while the letter "b" appears twice. The main idea behind this algorithm is converting a long string of the same character into the character followed by the count of its consecutive occurrences.

The first step of converting a sentence into a string code of compressed data through Run-Length Encoding is reading the sequence of data which usually contains repeating elements. For example, the sentence "aaabbcccc" has repeated letter "a", letter "b", and letter "c". This is the information we need to begin compressing data.

The next step is to take the first character inside the sentence. In this case, it's the letter "a". Then, we count how many times the same character repeat itself. The sequence "aaa" inside the sentence has 3 letters "a". Therefore, the first result code of Run-Length Encoding is a3.

If we repeat the process until the end of the sentence, the final code string we'll get is "a3b2c4". The length of this sentence has reduced from 9 to 6. The result code string after Run-Length Encoding algorithm depends on how many consecutive characters does the sentence have.

III. METHOD

A. *Datasets*

In this paper, several CSV files will be used to examine the performance of Huffman Algorithm and Run-Length Encoding on compressing these files. This results of this evaluation will be used to determine the capabilities, strength, and weakness of these Data Compression algorithms. In addition, the algorithm will also attempt to decompress the result of encoding in order to see the performance of restoring the CSV files like before they are compressed.

There are two kinds of datasets used for this paper in order to begin this evaluation, such as:

1. Test Datasets

   Test datasets are small and simple datasets used to evaluate the performance of an algorithm's capabilities in doing the task intended in order to validate it. These datasets consist of several CSV files each represent a certain condition to identify the advantages and disadvantages of Huffman Algorithm and Run-Length Encoding. The following is a list of test datasets used by the author:

   - data.csv: A normal test case, consists of three columns and four rows.
   - data(all_letters).csv: A modified version of data.csv in which all numerical values are written in words.

     (e.g, 9 is converted into "nine")
   - test1.csv: A test case consisting of 2 columns and 4 rows. In theory, this test benefits Run-Length Encoding more than Huffman Algorithm.
   - test2.csv: A test case consisting of 1 column and 2 rows. In theory, this test is bad for both Huffman Algorithm and Run-Length Encoding.
   - test3.csv: A test case consisting of 1 column and 1 row. In theory, this test benefits Run-Length Encoding more than Huffman Algorithm.
   - test4.csv: A test case consisting of 1 column and 1 row. In theory, this test benefits Huffman Algorithm more than Run-Length Encoding.
   - test5.csv: A test case consisting of 1 column and 1 row. In theory, this test benefits Run-Length Encoding more than Huffman Algorithm.

2. Kaggle Datasets

   Kaggle is an online platform, specialized in Data Science and Machine Learning. Kaggle is usually used to host competitions, where several users compete to build the best model to solve a specific problem. People can also share codes and datasets as well as collaborate on a project, building a large community.

   Kaggle also offers thousands of public datasets, ranging from multiple topics. These datasets can be used for exploratory data analysis (EDA), machine learning model building, deep learning projects, data visualization, and academic research. Several of public datasets available in Kaggle can be downloaded as CSV files. For this reason, the author may utilize various public datasets in Kaggle to assess the effectiveness of Huffman Algorithm and Run-Length Encoding in data compression.

   Out of these datasets within Kaggle, the author has picked three datasets to observe the performance of Huffman Algorithm and Run-Length Encoding when compressing relatively large files compared to the test datasets. The names of the public datasets from Kaggle used in this paper are:

   a. UCI Nursery Data Set

   b. Phone Prices

   c. Mushroom Classification

B. *Programming*

In this paper, the author utilizes Python for implementing the algorithm needed to conduct the experiment. Python is one of the most well-known programming languages out there. Furthermore, Python's flexibility upon managing data make this

programming language one of the most suitable out of most. Python also support multiple libraries convenient for data management, such as NumPy and Pandas.

The author makes three python files used to evaluate the Huffman Algorithm and Run-Length Encoding. The author used three libraries: heapq, collections, and csv, each with different functions. Despite only testing two kinds of algorithm, three files are needed for evaluating different cases. Then, the author uses a terminal for running the program and processing the files while keeping track of the results.

The three python files created for this study are:

- Huffman.py

  This file contains the algorithms in Python for creating a Huffman Tree, encoding, and decoding the file. Upon creating the functions necessary for such algorithms, the author utilize the "heapq" library and the "collections" library. The "heapq" library is used to create a heap for storing the data of the Huffman Tree. Meanwhile, from the "collections" library, the algorithm imports Counter function. This function's main capability is counting the frequency of every character inside a string, completing the first step for creating a Huffman Tree

  Each node in the tree contains four variables. The first two are the character itself and its frequency. Then, the other data are two links called left and right. The links are used to point to the child nodes, crucial for creating the Huffman Tree. The class also defines '_lt_', telling Python to compare the nodes by their frequencies.

  The build tree function begins by creating a heap containing all of the characters and their frequencies. Then, two nodes with the least frequencies are picked and merged into one. This process continues until only one node is left, completing the Huffman Tree.

  Once the Huffman Tree is finished, the "generate_code" function creates a list consisting of a character and its code. The algorithm will add a "0" when it goes left, and a "1" when it goes right. In a case where there is only one character in the text, the algorithm will give the code 0.

  The resulting code is the one used to encode and decoding the CSV file. The encoding process is done by replacing every character in the file with the code, while the decoding process is done by replacing the codes with the original characters.

- Run-Length.py

  The first version of the Run-Length Encoding algorithm in Python is by far the simplest algorithm out of the three. The encoding process begins by reading the first character in the file and set count to one. If the next character is the same, the count is added by one. If not, then the character and final count is added to resulting string. The count variable's value reverted to 1 to save the number of consecutive occurrences of the next character. This process is repeated until the whole CSV file's content has been read.

  However, there is an issue with the first algorithm when processing a file with a number value. For example, the encoding result of "111bb" is "312b". However, the decoding process will print three hundreds and one (312) letters instead. Therefore, the author modifies the code to add a special symbol to separate each character's specification. Unfortunately, this modification causes the compression ratio to be smaller, so both codes are tested.

- RunLengthv2.py

  The second version of the Run-Length Encoding algorithm in Python works a little differently from the previous version. The commas, despite being an integral part of a CSV file, has prevented long consecutive characters in a string, making the Run-Length Encoding less viable for compressing such files.

  So, rather than just managing the string from each row, this version of Run-Length Encoding Python algorithm processes each column instead, reading the first row as header. Values in a column also tends to repeat itself. For example, in a column named "Gender", the values tend to consist of "Male" and "Female". This algorithm still works under the same principle as a normal Run-Length Encoding, storing a value then the number of consecutive occurrences from said value.

  Although, this doesn't mean that this version of Run-Length Encoding is better than the previous one. It is still possible that this algorithm is less effective when dealing with datasets with a few rows or non-consecutive values. Therefore, it is still important to test both versions of the algorithm.

C. *Time Complexity Analysis*

An algorithm's efficiency is usually determined by the time and memory space needed to run it. An algorithm is deemed efficient if it requires as little time and space as possible. So, the less the needed time and space, the better. The time required to run an algorithm is called time complexity.

One method to measure the time complexity is observing the real-time runtime by executing the program. However, it's important to note that computer's specification and architecture can vary heavily. A computer with higher and more modern specification will run the same program faster. Furthermore, a compiler also effects how fast a program run because a compiler is responsible for generating the machine language. Therefore, if we want to be accurate with this method, we'll need to use the same device every time.

Fortunately, there is a more efficient way. Instead of determining the exact time, we can calculate the amounts of steps needed to process an input. The base of this calculation is the size of the input (e.g. n). A big O Notation (O(n)) can be used to represent the value of the time complexity of an algorithm. For this study, the author observes and calculates the time complexity for the Huffman Algorithm and Run-Length Encoding.

D. Compression Ratio

Compression ratio is the ratio between the original data's size with the compressed data's size. It is one of the simple ways to determine a data compression algorithm's effectiveness. A good data compression algorithm should be able to produce a compressed data smaller than the original file. The formula for determining the compression ratio is

Compression Ratio = Original Size/Compressed Data Size

To enhance operational efficiency, the author modified the program to calculate compression ratio right after running the program. Each time the program run to process a CSV file, the program will show the compression ratio. Different test datasets or Kaggle datasets may cause different outcomes, even after being run by the same algorithm. Therefore, each runtime will provide a useful information about the compression ratio for analyzing the Huffman Algorithm and Run-Length Encoding's performance.

IV. RESULTS AND ANALYSIS

A. Huffman Algorithm

Time complexity for Huffman Algorithm proves to be quite complicated than expected by the author. Huffman encoding, decoding, and code generation functions' time complexities are determined by the number of unique characters a file has. It is almost impossible to determine how many unique characters inside a file with only the information of the file's size.

The other problem is that the code's length for each character can't be easily determined without looking inside the contents of the CSV file. Therefore, the author proposes three variables for Huffman Algorithm:

- n: the size of the file
- u: the number of unique characters inside a file
- s: the total length of the Huffman-encoded bit stream.

The time complexities of each Huffman Algorithm's processes are described below:

- Building a Huffman Tree:

  Building a Huffman Tree begin by reading the characters inside a string one by one, adding the time complexity by the size of the original CSV file (O(n)). Next, the algorithm takes each distinct character's node (O(u)) and begin the process of merging them to create a Huffman Tree. Meanwhile, the heap operations' time complexity for each distinct character is O (log u). Therefore, the time complexity needed for generating the Huffman Tree is O (n + u log u).

- Generating codes

  The code generation is done dynamically by visiting the Huffman Tree while continuously adding 0 or 1 to the resulting code string. The tree's size depends on how many distinct characters inside a tree, so the time complexity for this process is O (u).

- Encoding

  The encoding process begins by building the Huffman Tree and generating the codes for each character. Then, the final compressed file string is appended by the code string for every single character of the original size. Therefore, the time complexity for encoding is O (n + u log u + s).

- Decoding

  The decoding process is done by reading every bit inside the Huffman-encoded bit stream. So, total complexity time for decoding is O (s).

Here is the compression ratio after running the Huffman Algorithm to process the CSV files from the test datasets and the Kaggle datasets:

TABLE I. HUFFMAN ALGORITHM

File Name	Result		
	Original Size (bits)	***Compressed Size (bits)***	***Compression Ratio***
data.csv	456	237	1.9
data(all_letter).csv	648	324	2.0
test1.csv	640	193	3.32
test2.csv	424	255	1.66
test3.csv	160	30	5.33
test4.csv	256	32	8.0
test5.csv	200	60	3.33
cleaned_all_phones.csv	1675752	995533	1.68
mushrooms.csv	2992024	1176027	2.54
nursery.csv	8475584	4401870	1.93

B. Run-Length Encoding

Due to the two possible implementations of Run-Length Encoding, the results are categorized into two approaches: row-wise (Read-by-Rows) and column-wise (Read-by-Columns). The row-wise Run-Length Encoding will also be adjusted on the appearance of any number value. So, the row-wise Run-Length Encoding's performance is judged by the best out of both versions.

Assume that N is the size of the file and L is the size of the compressed file. Despite having different methods, both the row-wise and column-wise Run Length Encoding have the same time complexity for both encoding and decoding, which are O(N) and O(N+L). The Run-Length Encoding algorithm only need to pass the file once to complete the final string code. Meanwhile, the decoding process is executed by reading the compressed file and rewriting the contents. The compressed file isn't always smaller than the original file if there are enough characters that only appear consecutively once. (e.g. "ABCDEFGH").

Here is the compression ratio after running row-wise Run Length Encoding to process the CSV files from the test datasets and the Kaggle datasets:

TABLE II. ROW-WISE RUN-LENGTH ENCODING

File Name	Result		
	Original Size (byte)	***Compressed Size (byte)***	***Compression Ratio***
data.csv	57	227	0.25
data(all_letter).csv	81	158	0.51
test1.csv	80	34	2.35
test2.csv	53	106	0.5
test3.csv	20	9	2.22
test4.csv	32	64	0.5
test5.csv	25	10	2.5
cleaned_all_phones.csv	209469	817270	0.26
mushrooms.csv	374003	1495983	0.25
nursery.csv	1059448	4192855	0.25

Here is the compression ratio after running column-wise Run Length Encoding to process the CSV files from the test datasets and the Kaggle datasets:

TABLE III. COLUMN-WISE RUN-LENGTH ENCODING

File Name	Result		
	Original Size (byte)	***Compressed Size (byte)***	***Compression Ratio***
data.csv	57	108	0.53
data(all_letter).csv	81	134	0.6
test1.csv	80	112	0.71
test2.csv	53	56	0.95
test3.csv	20	21	0.95
test4.csv	32	33	0.97
test5.csv	25	26	0.96
cleaned_all_phones.csv	209469	136154	1.54
mushrooms.csv	374003	285623	1.31
nursery.csv	1059448	364110	2.91

C. Analysis

Based on the program files used for this experiment, the Huffman Algorithm has higher time complexity than Run-Length Encoding when encoding a CSV file and decoding the compressed result. The time complexity for encoding and decoding a file using Huffman Algorithm is O (n + u log u + s) and O (s), assuming that "n" is the size of the file, "u" is the number of unique characters inside a file, and "s" is the total length of the Huffman-encoded bit stream. However, the Run-Length Encoding or Decoding only have the time complexity of O(N), where N is the size of the CSV file.

One of the main contributors for this result is the additional processes the Huffman Algorithm needs to do. Huffman Algorithm requires a tree data structure in order to create the Huffman Tree. Huffman Algorithm also involves a special algorithm to generate code for each distinct character, taking the character's information from the Huffman Tree itself. Meanwhile, the Run-Length Encoding only require to save consecutive values within a file and the value appearance's count. Therefore, Run-Length Encoding generally has less time complexity than Huffman Algorithm.

The first two datasets, data.csv and data(all_letters).csv, are random datasets created by the author. It consists of three columns and three rows, with each value is distinct from one another. It is the reason why the Run-Length Encoding wasn't very effective. Meanwhile, the Huffman Algorithm doesn't care about consecutive occurrence, only the frequencies of each character. Therefore, Huffman Algorithm is more effective than Run-Length Encoding in this case.

The next part of the experiment is creating and using more test datasets. The datasets consist of test1.csv, test2.csv, test3.csv, test4.csv, and test5.csv. These datasets are mostly only form by one or two columns and rows. They are made to see the performance of Huffman Algorithm and Run-Length Encoding on smaller scale files.

Unexpectedly, the Huffman Algorithm outperforms both row-wise Run-Length Encoding and column-wise Run-Length in every test case. In test 5, the string used is "AAAAABBBBBCCCCCDDDDDEEEEE". In theory, this string should have given more advantage for Run-Length Encoding due to the long consecutive runs of the same character. However, the end result indicates otherwise.

One plausible reason why this happen is based on how the data is stored. In Huffman Algorithm, the compressed data can be stored in bits due to the code's value only consisting of number 1's and 0's. Meanwhile, a compressed file by Run-Length Encoding is still stored in a string of characters, where every character requires 8 bits of data. Therefore, Huffman Algorithm tends to create a smaller compressed file than Run-Length Encoding.

Lastly, the Huffman Algorithm and Run-Length Encoding are tested with external datasets from Kaggle, specifically the UCI Nursery Data Set, Phone Prices, and Mushroom Classification Data Set. At first, Huffman Algorithm still proves to be more effective than Run-Length Encoding upon compressing the Phone Prices dataset and the Mushroom Classification Data Set. In spite of that, Run-Length Encoding managed to have more compression ratio than Huffman Algorithm when compressed the UCI Nursery Data Set.

After reassessing the UCI Nursery Data Set, the author notices that many columns have the same consecutive values. The values on most columns are packed together, creating a list of consecutive runs of the same value. For example, the first column contains only three unique values, with identical values grouped together consecutively. For this reason, the column-wise Run-Length Encoding managed to take advantage of them to create a compressed file with size smaller than a compressed file from Huffman Algorithm.

V. Conclusion

Generally, Huffman Algorithm is more flexible and effective than Run-Length Encoding for compressing CSV files, yet takes longer time due to the algorithm's complexity. However, in some cases where long consecutive values exist, Run-Length Encoding may compress a CSV file better than Huffman Algorithm. Nevertheless, this situation remains exceedingly conditional.

GITHUB LINK

https://github.com/BravewindPC/Run-Length-Encoding-and-Huffman-Algorithm

Acknowledgment

The author would like to thank first of all to God for all of the guidance during the process of learning and writing this paper. The author would also like to thank the lecturer of ITB Discrete Mathematics, Mr. Rinaldi Munir for sharing his knowledge and guidance for all of his students including the author throughout the semester. The author would also like to thank his family who have given their unconditional love and support throughout his study in ITB from the beginning.

References

[1] terapan-ti.vokasi.unesa.ac.id, 'Mengenal Kompresi Data: Pengertian, Metode, dan Penerapannya di Era Digital' https://terapan-ti.vokasi.unesa.ac.id/post/mengenal-kompresi-data-pengertian-metode-dan-penerapannya-di-era-digital [Accessed on 12 Juni 2026]

[2] barracuda.com, 'Data Compression' https://www.barracuda.com/support/glossary/data-compression [Accessed on 12 June 2026]

[3] primacom.com, '3 Manfaat Kompresi Data bagi Perusahaan' https://primacom.com/news/3-manfaat-kompresi-data-bagi-perusahaan/ [Accessed on 12 June 2026]

[4] seagate.com, 'Apa itu Kompresi Data dan Bagaimana Cara Kerjanya?' https://www.seagate.com/id/id/blog/what-is-data-compression/ [Accessed on 13 June 2026]

[5] trivusi.web.id, 'Kompresi Data: Pengertian, Jenis, Kelebihan, dan Kekurangannya' https://www.trivusi.web.id/2023/01/kompresi-data.html [Accessed on 13 June 2026]

[6] adobe.com, 'Apa itu file CSV?' https://www.adobe.com/acrobat/resources/document-files/what-is-a-csv-file.html [Accessed on 13 June 2026]

[7] sciencedirect.com, 'Lossless Compression' https://www.sciencedirect.com/topics/engineering/lossless-compression [Accessed on 15 June 2026]

[8] adobe.com, 'What is Lossless Compression? Everything you need to know.' https://www.adobe.com/uk/creativecloud/photography/discover/lossless-compression.html [Accessed on 15 June 2026]

[9] tinify.com/blog, 'Lossy vs Lossless Compression Explained for Image Optimization' http://tinify.com/blog/lossy-vs-lossless-compression-image-optimization [Accessed on 15 June 2026]

[10] scribd.com, 'Lossless Compression Algorithms Overview' https://www.scribd.com/presentation/883719266/Stu-Lossless-Compression-Algos [Accessed on 15 June 2026]

[11] geeksforgeeks.org, 'Huffman Coding Algorithm' https://www.geeksforgeeks.org/dsa/huffman-coding-greedy-algo-3/ [Accessed on 15 June 2026]

[12] jurnal.polgan.ac.id, 'Application of Huffman Algorithm and Unary Codes for Text File Compression' https://jurnal.polgan.ac.id/index.php/sinkron/article/view/11567 [Accessed on 15 June 2026]

[13] kaggle.com, 'UCI Nursery Data Set' https://www.kaggle.com/datasets/heitornunes/nursery [Accessed on 15 June 2026]

[14] kaggle.com, 'Phone Prices' https://www.kaggle.com/datasets/berkayeserr/phone-prices [Accessed on 15 June 2026]

[15] kaggle.com, 'Mushroom Classification' https://www.kaggle.com/datasets/uciml/mushroom-classification [Accessed on 15 June 2026]

[16] geeksforgeeks.org, 'Test Data in Software Testing' https://www.geeksforgeeks.org/software-testing/what-is-test-data-in-software-testing/ [Accessed on 17 June 2026]

[17] coursera.org, 'What Is Kaggle and What Is It Used For?' https://www.coursera.org/articles/kaggle [Accessed on 17 June 2026]

[18] youtube.com, 'How to Find and Use Kaggle Datasets in Your Project' https://www.youtube.com/watch?v=krkS9u140tM&t=479s [Accessed on 17 June 2026]

[19] R. Munir, 'Homepage Rinaldi Munir'. https://informatika.stei.itb.ac.id/~rinaldi.munir/. [Accessed om 17 June 2026]

[20] ioriver.io, 'Data Compression Ratio' https://www.ioriver.io/terms/data-compression-ratio [Accessed om 17 June 2026]

[21] geeksforgeeks.org, 'Huffman Coding in Python' https://www.geeksforgeeks.org/dsa/huffman-coding-in-python/ [Accessed om 17 June 2026]

[22] algomap.io, 'Heaps & Priority Queues' https://algomap.io/lessons/heaps [Accessed om 17 June 2026]

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, 17 Juni 2025

Bayu Palamarta Wirawan - 13525085