MJI
Media Jurnal Informatika
Vol. 17, No. 2, December 2025
p-issn : 2088-2114, e-issn : 2477-2542

Evaluation of Deflate Algorithm in Lossless Compression of Digital Document Formats

Muhammad Irwan Nawawi [a,1], Finsa Nurpandi [a,2,*]

[a] Informatics Engineering Program, Suryakancana University, Cianjur 43216, Indonesia
[1] muhammadirwannawawi@gmail.com; [2] finsa@unsur.ac.id*;
* corresponding author

ARTICLE INFO

Article history
Received 2025/08/06
Revised 2025/10/05
Accepted 2025/12/05

Keywords
Deflate
Data Compression
Document
Python
Zlib

ABSTRACT

As the volume of digital data continues to escalate across sectors such as education, business, and government, the demand for efficient data storage and transmission methods grows increasingly critical. Data compression algorithms offer a prevalent solution to this challenge. This study undertakes an evaluation of the Deflate algorithm's performance in compressing digital document files, specifically examining its efficacy in reducing file size and its efficiency in processing time. Employing a comparative analysis methodology, the research involves measuring file sizes before and after compression, recording compression and decompression durations on a machine with an Intel Core i5 CPU, 8 GB RAM, running Windows 10 64-bit, and calculating compression ratios. The implementation utilizes Python and the Zlib library, which directly supports the Deflate algorithm. Tests were conducted on diverse document types, including plain text files, mixed-content files, and files rich in visual elements like images. The findings indicate that the Deflate algorithm achieves a significant compression ratio, reducing file sizes by over 90% and reaching a maximum ratio of 99.60% for text files. Compression and decompression operations were most rapid for text files, averaging 0.01 seconds. However, for documents containing images, the compression ratio was considerably lower and less impactful. Notwithstanding this, the compression and decompression times remained relatively swift and consistent across all document types. These results underscore the importance of aligning compression algorithm selection with the specific content characteristics of a document to attain optimal efficiency.

1. Introduction

In the current digital landscape, data has emerged as a vital asset, underpinning the operations of various sectors, including education, government, and business [1], [2]. The proliferation of information systems, cloud platforms, and digital services has led to an exponential growth in the volume of digital data generated and exchanged daily [3]. As organizations increasingly rely on data-driven decision-making and paperless workflows, the need for efficient data storage and transmission becomes more pressing. Among the most frequently encountered types of digital data are documents in formats such as PDF, DOCX, PPTX, and TXT. These formats are commonly used to encapsulate structured text, images, tables, and graphics—elements that contribute significantly to information density and file size [4], [5]. As a result, large-scale document handling not only poses storage challenges but also affects transfer latency, server bandwidth, and resource management in digital infrastructures [6].

To address these challenges, data compression technology has become an indispensable solution. By reducing the physical size of files without altering or losing their content, compression techniques help organizations optimize system performance, reduce storage costs, and streamline the exchange of information. Two principal classifications of compression techniques exist: lossless and lossy [7]. While lossy compression is suitable for certain types of multimedia data where minor data degradation is tolerable, lossless compression is preferable for textual and document data where preserving the original content in its entirety is mandatory. Document formats typically used in academic, legal, or governmental contexts often contain critical information that must remain intact during the compression-decompression cycle.

Among lossless compression algorithms, the Deflate algorithm—a hybrid of LZ77 sliding-window compression and Huffman entropy encoding—has been widely adopted due to its balance of fast processing capabilities and high compression efficiency [8]. It has been integrated into several standard file formats and tools, such as ZIP, GZIP, and PNG, and is dominant in practical implementations due to its portability, reliability, and general-purpose applicability. However, despite its widespread usage and the emergence of newer lossless algorithms such as Brotli and Zstandard, Deflate remains deeply embedded in widely used formats and software ecosystems (e.g., ZIP, GZIP, PDF toolchains), making it highly relevant in practice. Consequently, critical evaluation of Deflate's performance when applied specifically to document file formats such as PDF, DOCX, PPTX, and TXT remains limited in academic literature [9]. Most existing research on compression algorithms tends to focus either on theoretical aspects of compression techniques or on multimedia formats—such as images, videos, or 3D models—where compression methods confront different computational and perceptual challenges. For instance, several studies have evaluated Deflate's role in compressing 3D data models by comparing it with other algorithms in terms of compression ratios and processing speed. However, these findings, while insightful, may not directly translate to scenarios involving highly structured textual data typically found in administrative, academic, or business documents. This leaves a gap in the empirical assessment of Deflate's practical performance when compressing files that prioritize data fidelity and structured content.

Given the centrality of textual digital documents in daily academic and professional activities, alongside the growing imperative for rapid data access and efficient storage, it is increasingly important to conduct systematic evaluations of the Deflate algorithm's performance in compressing these formats. Recent studies have highlighted that factors such as the initial size of the file, the structural complexity of document content—including the presence of tables, charts, images, and metadata—and the nature of the file format itself can considerably impact the attainable compression ratio and computational performance [10], [11]. For instance, Deflate demonstrates superior compression efficiency on text-based files due to the repetitive character patterns often present, achieving compression ratios exceeding 38%–90% depending on the test data [11]. However, the algorithm's effectiveness decreases in documents with embedded multimedia elements, where redundancy is lower and data structures are more complex [12], [13]. Additionally, research suggests that it is essential to balance the trade-offs between compression ratio, processing time, and resource consumption, since optimizing for one aspect may compromise others, such as throughput or energy use (Poranki, 2020). Comparative studies indicate that while the Deflate algorithm remains a benchmark standard, newer algorithms like Zstandard and Brotli can offer higher performance in some scenarios, further emphasizing the need to align compression methods with specific application requirements [12]. Therefore, ongoing research and evaluation are necessary to guide the selection and potential improvement of compression algorithms suited to the evolving landscape of digital document management.

Therefore, this study aims to address a significant gap in the literature by offering a thorough analysis of the Deflate algorithm's performance when applied to widely used digital document formats. The research intends to evaluate critical performance metrics including compression ratio, execution time encompassing both compression and decompression phases, and the integrity of the decompressed output file. These parameters are essential for the practical adoption of the Deflate algorithm in real-world applications such as document archiving, electronic transmission, and cloud-based synchronization systems, where efficiency and data fidelity are paramount. Prior studies have highlighted the importance of balancing compression efficiency and computational speed to optimize overall system performance in document management contexts [14], [15]. Moreover, the effective evaluation of compression algorithms plays a vital role in guiding the design of scalable and adaptive systems capable of handling diverse file types and evolving data formats [16]. By providing

empirical evidence of Deflate's operational strengths and limitations, this study not only enriches the academic discourse on data compression but also contributes to the advancement of more efficient and tailored document management solutions that meet the specific needs of institutions and industries reliant on digital document processing.

This study aims to apply the Deflate algorithm in the compression of digital document files, with a focus on evaluating outcomes based on post-compression file size, compression and decompression speed, and the achieved compression ratio. These parameters serve as key indicators in determining the extent to which the Deflate algorithm is effective for compressing digital documents. Additionally, the results of this study are expected to provide insights into the algorithm's capability to optimize the storage of document files. Through this research, it is anticipated that a practical implementation and assessment of the Deflate algorithm on various document file types and sizes can be undertaken. By conducting quantifiable testing on the resulting file sizes and compression ratios, the study will generate empirical evidence that supports an informed understanding of Deflate's effectiveness beyond traditional use cases, such as archival formats. In practical terms, the findings from this study may also serve as a reference for the development of more tailored, efficient, and adaptable compression systems suited to digital document management across diverse platforms and industries.

2. Method

2.1 Research Method

This study adopts a comparative analysis approach aimed at assessing the performance of the Deflate algorithm in compressing digital document files across a range of commonly used formats. The methodology involves testing multiple file types to evaluate the algorithm's effectiveness in terms of compression ratio, speed, and output file size, which are treated as the primary performance indicators in this study. Other metrics such as memory consumption and CPU utilization are recognized as important in compression algorithm evaluations but are left for future work to maintain a focused experimental scope. Guided by the general research framework as described in "Research Imagination: An Introduction to Qualitative and Quantitative Methods" [17], this study follows six essential stages: formulating the research problem, selecting relevant document types as research subjects, collecting empirical data through controlled compression and decompression procedures, analyzing the resulting data quantitatively, interpreting the findings in relation to document characteristics and algorithmic performance, and reporting the conclusions in a structured format. This systematic approach ensures that the analysis remains both rigorous and replicable. Further details regarding the methodology applied are visually summarized in Figure 1, providing a clearer view of the research process implemented in this study.

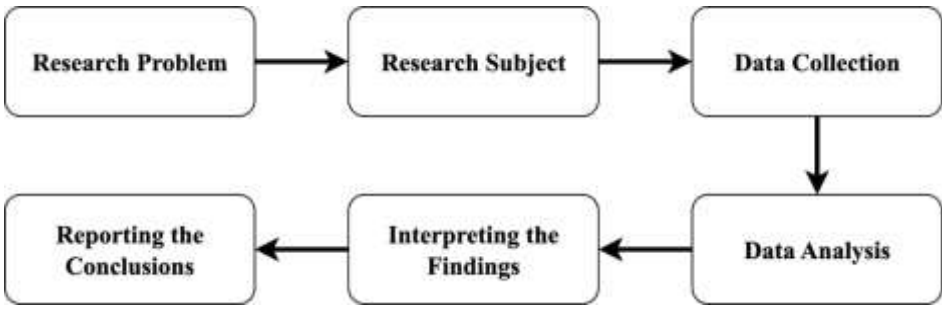

Fig. 1. Research Method

In the initial stage of problem formulation, this research identifies and defines the core issue to be addressed: the implementation and performance evaluation of the Deflate compression algorithm when applied to various digital document formats through a custom-built application developed using the Python programming language. The research seeks to determine how efficiently Deflate compresses files of differing structures and complexities by analyzing key performance metrics. In the object selection phase, the study focuses on four widely used file formats—DOCX, PDF, TXT, and PPTX—chosen based on their distinct characteristics and extensive application across different domains. DOCX was selected as a representative of

word processing documents that typically contain both text and formatting metadata; PDF was included due to its ability to preserve static layouts and cross-platform consistency; TXT serves as a baseline format with plain, unformatted text; and PPTX was chosen for its integration of both textual and multimedia components used in presentation materials. Each file format undergoes a controlled compression process using the Deflate algorithm, during which the required compression time is measured over multiple runs per sample and averaged to reduce timing variance and measurement noise, and the compression ratio is evaluated by comparing file sizes before and after the operation. Through these steps, the study aims to produce quantitative insights into the effectiveness and reliability of Deflate across document types that vary in complexity and structure.

In the data collection phase, a set of test documents was gathered in four commonly used digital formats—DOCX, PDF, TXT, and PPTX—to support the experimental assessment of the Deflate algorithm's performance. For each document type, multiple samples were selected from real-world user documents to ensure variation in file characteristics while maintaining consistency in file size, which was deliberately limited to a maximum of 50 MB per document. Although these documents reflect realistic usage scenarios, the corpus still represents a limited subset of possible document structures and domains, so the generalizability of the results to all document collections cannot be assumed. This constraint was imposed to preserve the stability and efficiency of the testing application during execution. The collected documents were curated to reflect diverse content structures and levels of complexity, aligning with real-world use cases. In the subsequent data analysis phase, each document format included three controlled variations that differed specifically in their internal data composition. For instance, the DOCX and PDF test samples were standardized with a consistent text length of approximately 20,000 words but varied based on image quality: the first version included high-definition (HD) graphics, the second contained standard-quality images, and the third featured a combination of both. This stratification allowed the researchers to evaluate how content complexity—particularly the presence and resolution of embedded images—influenced the Deflate algorithm's compression performance across different file types.

In the result interpretation phase, the performance of the Deflate algorithm was evaluated by applying it to the collected test documents using a tailored application developed specifically for this study. The implementation was carried out in the Python programming environment, utilizing the built-in Zlib library, which provides native support for the Deflate compression method. To facilitate this process, the application was developed following the prototype-based software development paradigm as proposed by Roger S. Pressman, Ph.D. This methodology emphasizes close alignment between user requirements and system functionality, making it particularly well-suited for research contexts that involve iterative testing and evaluation of lightweight software systems [18]. By adopting this development model, the research ensures that the compression application remains adaptable and responsive to the specific needs of document-based experimentation, enabling efficient interpretation of outcomes related to compression speed, ratio, and system compatibility. This approach also provides flexibility for adjustments based on preliminary findings, thereby enhancing the validity of the results derived from the applied Deflate algorithm [19], [20].

2.2 Deflate Algorithm

Previous research by Maulana [21] and Utomo [7] elaborates that the Deflate compression algorithm originated as an innovative solution introduced by Phil Katz, who sought to enhance data processing efficiency and address the rising demand for file archiving in the early 1990s. Developed initially for the PKZIP software—which soon became a leading archiving tool—Deflate represents a significant milestone in the realm of lossless data compression. The algorithm ingeniously integrates two well-established techniques: LZ77, which replaces recurring data patterns with references based on distance and length, and Huffman coding, which further optimizes the representation of symbols by assigning shorter codes to frequently occurring data [22]. This synergistic combination enables Deflate to achieve substantial reductions in file size while rigorously maintaining data integrity, ensuring the original information can be fully restored upon decompression. The reliability of Deflate as a lossless mechanism has led to its widespread adoption in standard file formats such as ZIP and GZIP, where accuracy and data preservation are paramount requirements [8]. The algorithm's enduring impact on digital document management is evident in both its historical significance and its continuing prevalence in contemporary compression practices.

According to research by Utomo [7], the Deflate algorithm eliminates data redundancy through a sequential two-phase process, leveraging both the LZ77 and Huffman coding techniques. Initially, Deflate utilizes LZ77 coding to search for repeated character sequences within each data block. When recurring patterns are identified, the algorithm replaces them with references that indicate the length of the sequence and the distance from its previous occurrence. This efficient handling of duplicate data is consistent with findings from studies by Nugraha [23] and Oktaviani [24], which describe how LZ77 employs a "sliding window" mechanism, consisting of a look-ahead buffer (containing input data for compression) and a search buffer (acting as a dictionary for previously encountered patterns). The reference format, typically denoted as (distance, length), enables the encoder and decoder to match and substitute repetitive content, such as replacing "ABCABC" with (3, 3), thereby conserving space by reducing redundancy. Subsequently, the tokens generated during the LZ77 stage undergo further compression through Huffman coding, which replaces frequently occurring symbols with shorter codewords based on their occurrence frequencies. This combined process not only minimizes the bit-length required for storage but also ensures that the original data structure is preserved, fulfilling the requirements of lossless compression. As demonstrated by several empirical studies, the synergy of LZ77 and Huffman algorithms within Deflate significantly enhances efficiency in data compression, especially when the sliding window and buffer configurations are optimally tuned to the characteristics of the input data.

The subsequent phase in the Deflate compression workflow involves utilizing the output generated by the LZ77 stage as input for binary encoding through the construction of a Huffman tree. As explained in research by Irliansyah et al. [8], the process begins with sorting the symbols produced by LZ77 according to their frequency of occurrence within the data. Next, pairs of symbols or subtrees with the lowest frequencies are iteratively merged to form a hierarchical binary tree structure, with this merging and reordering continuing until a single comprehensive Huffman tree is achieved. In this tree, the right-hand branches are designated with a binary value of 1, while left-hand branches receive a value of 0. To derive the final Huffman codes, the algorithm traverses paths from the root to each leaf node, collecting the sequence of binary digits encountered along the way, which uniquely represents each symbol or token according to its frequency-based encoding. This systematic process allows for efficient data compression, as frequently appearing symbols are assigned shorter binary codes, thereby optimizing the overall file size without loss of content integrity. The integration of LZ77 and Huffman coding within Deflate ensures that the redundant patterns and recurring data are minimized beforehand, resulting in improved space utilization and reliable data recovery upon decompression.

In his research, Irliansyah et al. [7] also provided a general description of the process in the deflate method which uses the LZ77 algorithm and then continues with the Huffman algorithm in compressing text. For example, there is a text to be compressed, namely "**IRWAN_NAWAWI**", which consists of 12 bytes where 1 character is 1 byte, and 1 byte is 8 bits, then the total size of the text to be compressed = 96 bits.

First step, the deflate method applies the LZ77 algorithm with the following conditions [25] and is illustrated in Table 1:

- Look ahead buffer: the set of characters to be compressed.
- Search buffer: a collection of characters that already exist in the dictionary.
- Result: is an output consisting of offset (distance from the current position to the previous same string), length (string length), and next symbol (next string).

Table 1. Compression Steps Using LZ77 Algorithm

Search buffer	Look ahead buffer	Result
	IRWAN_NAWAWI	0, 0, I
I	RWAN_NAWAWI	0, 0, R
IR	WAN_NAWAWI	0, 0, W
IRW	AN_NAWAWI	0, 0, A
IRWA	N_NAWAWI	0, 0, N
IRWAN	_NAWAWI	0, 0, _
IRWAN_	NAWAWI	2, 1, A
IRWAN_NA	WAWI	5, 2, W

IRWAN_NAW	AWI	1, 2, I
IRWAN_NAWAWI		NULL

Subsequently, the data produced by the LZ77 algorithm serves as input for further compression via the Huffman algorithm, where specific operational criteria must be met, such as analyzing the frequency of individual character occurrences, as exemplified in Table 2:

- Sort the output values in the LZ77 algorithm based on their frequency of occurrence.
- Create a binary tree and pair two binary trees that have frequencies from the smallest sequence until one binary tree is formed.
- Label the branches of the binary tree with the value 0 for the left side and 1 for the right side.
- Traverse the binary tree from the topmost branch to the root to get the binary values.

Table 2. Character Appearance Frequency Based on LZ77 Algorithm Output

Character	Frequency
I	2
W	2
A	2
R	1
N	1
-	1

Figure 2 illustrates a binary tree structure that was constructed by organizing the frequency of characters derived from the output of the LZ77 algorithm, serving as the basis for subsequent Huffman encoding.

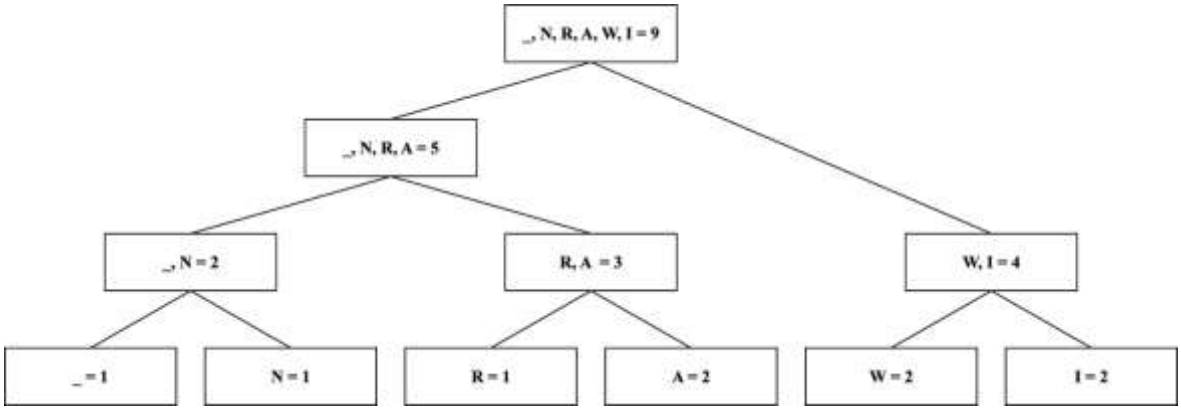

Fig. 2. Binary Tree Based on Huffman Algorithm

Based on the binary tree structure employed in Huffman algorithm compression, unique bit codes are generated for each character in the compressed data. This coding process assigns shorter bit sequences to more frequently occurring characters, thereby optimizing overall data size. As a result, the Deflate compression method significantly reduces the original text size; for example, an initial size of 96 bits can be compressed to as little as 23 bits. This demonstrates the efficiency of Huffman encoding in minimizing bit usage while preserving the ability to fully reconstruct the original data without loss.

Table 3. Deflate Method Compression Results

Char	Frequency	Bit	Size
I	2	11	2 x 2 bit = 4 bit
W	2	10	2 x 2 bit = 4 bit
A	2	011	2 x 3 bit = 6 bit
R	1	010	1 x 3 bit = 3 bit
N	1	001	1 x 3 bit = 3 bit
-	1	000	1 x 3 bit = 3 bit
Total			23 it

3. Results and Discussion

3.1 Research Subject

The research objects in this study consist of test documents in four widely used digital formats: DOCX, PDF, TXT, and PPTX. Each of these document formats will be subjected to compression using the Deflate algorithm to evaluate the compression speed and quality of the output files. For example, on the TXT test set, Deflate achieved a maximum compression ratio of 99.60% with an average compression time of 0.01 seconds, while on larger DOCX and PDF documents the algorithm still reduced file sizes substantially although with relatively higher processing times due to increased structural complexity. These quantitative results, presented in the subsequent tables, provide concrete evidence of Deflate's effectiveness in balancing compression efficiency and processing speed across different document types. The selection of these formats is strategically aligned with the research objectives, considering their relevance and typical usage scenarios. DOCX is included as a prevalent format for word processing, primarily used in Microsoft Word, making it representative of commonly edited and structured text documents. PDF is chosen due to its widespread adoption for document dissemination where maintaining a fixed layout is critical and alterations are generally prohibited. The TXT format represents the simplest form of text data, allowing evaluation of compression effectiveness on unformatted, plain text content. Lastly, PPTX files are incorporated because of their popularity in both business and academic presentations, often containing a mixed multimedia composition of text, images, and graphics. This diverse set of file types thus provides a balanced and comprehensive basis for assessing compression performance across varying data complexities and content structures.

3.2 Data Collection

The dataset utilized in this study comprised test documents in four distinct file formats: DOCX, PDF, TXT, and PPTX. To ensure comprehensive evaluation, multiple samples of each format were collected, with file sizes varying according to the testing requirements but constrained to a maximum of 50 MB per document. This limitation was deliberately imposed to maintain the stability and optimal performance of the compression application throughout the testing phase, preventing potential disruptions caused by excessively large files. The curated dataset thus reflects a balanced representation of common document types and sizes within practical operational boundaries. Detailed information regarding the collected test data, including file format distribution and size variations, is systematically presented in Table 4, serving as a foundation for subsequent compression performance analysis.

Table 4. Test Documents Data

No	File Name	File Type	Size
1	ujisatu	docx	20 MB
2	ujidua	docx	7.6 MB
3	ujitiga	docx	27.15 MB
4	ujisatu	pdf	2.8 MB
5	ujidua	pdf	2.4 MB
6	ujitiga	pdf	4.9 MB
7	ujilzw	txt	939 KB
8	ujideflate	txt	796 KB
9	ujikombinasi	txt	1.6 MB
10	ujisatuppt	pptx	223 KB
11	ujiduappt	pptx	5.7 MB
12	ujitigappt	pptx	8.2 MB

3.3 Data Analysis

After successfully collecting the test documents, an in-depth content analysis was conducted focusing on the number of characters and images contained within each file. The results of this analysis are summarized in Table 5, categorizing the documents by their file types—DOCX, PDF, and TXT. Specifically, the files labeled ujisatu.docx and ujisatu.pdf each consist of sixty pages with approximately 20,000 words and include several high-definition images, serving as a benchmark for comparison with other samples. Meanwhile, the ujidua.docx and ujidua.pdf documents also contain sixty pages and 20,000 words but feature images of standard resolution. The third group, comprising ujitiga.docx and ujitiga.pdf, extends to seventy pages, maintaining the 20,000-word count, yet incorporates a blend of both high-definition and standard-quality images. Consequently, these six test documents span three different file formats and possess varying file sizes, reflecting differences in image quality and content composition that are critical factors for evaluating the performance of the compression algorithm under varying data complexities.

Table 5. Data Analysis of Docx, PDF, and Txt File Types

No	File Name	File Type	Number of Pages	Number of Words	Number of Images	
					Standar Quality	High Quality
1	ujisatu	docx	60	20.000	0	10
2	ujidua	docx	60	20.000	10	0
3	ujitiga	docx	70	20.000	10	10
4	ujisatu	pdf	60	20.000	0	10
5	ujidua	pdf	60	20.000	10	0
6	ujitiga	pdf	70	20.000	10	10
7	ujilzw	txt	-	10.000	-	-
8	ujideflate	txt	-	10.000	-	-
9	ujikombinasi	txt	-	20.000	-	-

The analysis in Table 5 shows that documents with a higher number of high-quality images (e.g., ujisatu.docx/pdf) tend to exhibit lower effective compression ratios compared to text-dominant documents. This behavior is consistent with the fact that Deflate operates on the byte stream level and is highly effective for redundant textual data, but it cannot further compress embedded binary media such as JPEG or PNG images, which are already stored in a compressed format. Consequently, as image quality and image count increase, a larger portion of the file consists of incompressible binary payloads, leading to reduced overall compression efficiency despite Deflate's strong performance on the surrounding textual content. These results confirm that the presence of high-resolution images reduces the achievable compression ratio because Deflate cannot significantly reduce the size of pre-compressed binary image data embedded within the document.

Table 6 presents the detailed characteristics of the test files corresponding to the PPTX format. The study includes three separate PPTX documents, each comprising 10 slides, carefully selected to represent varying content compositions. The first document predominantly contains textual information, constituting approximately 70% of the content, with images occupying the remaining 30%. Conversely, the second document offers an equal distribution of textual and image content, each accounting for 50%. The third and final document is characterized by a higher image density, with 70% of its content being visual elements and the remaining 30% consisting of text. This deliberate variation in text-to-image ratios within the PPTX files is intended to facilitate a thorough evaluation of the Deflate algorithm's effectiveness in compressing presentation files with diverse content complexities and multimedia integration.

Table 6. PPTX File Type Data Analysis

No	File Name	Number of Pages	Text Composition (%)	Image Composition (%)
1	ujilzw	10	70	30
2	ujideflate	10	50	50
3	ujikombinasi	10	30	70

3.4 Interpreting the Findings

The Deflate compression process begins with the execution of the LZ77 algorithm, which takes an input file and sequentially reads each character within the file's content. Character matching is performed by comparing the current data within the look-ahead buffer against previously processed data stored in the search buffer, which initially is set to FALSE to allow the matching operation. When a matching sequence is detected in the look-ahead buffer, the search buffer is set to TRUE, and corresponding results are produced to represent these matches. Upon completion of the LZ77 phase, the Huffman coding method is subsequently applied to the resulting data. This involves calculating the frequencies of characters within the output, beginning with the

identification of the lowest-frequency characters. Pairs of characters and nodes are merged iteratively: nodes and characters with the lowest frequencies are combined, with previous elements deleted once they have been paired. This merging process continues repetitively until no further combinations remain, producing a complete Huffman tree. Each node in this final tree is then assigned a unique bit code, which serves as the substitution for the original characters, effectively compressing the data. This combined sequence of LZ77 followed by Huffman coding provides an efficient, lossless compression mechanism. A detailed representation of this workflow is provided in Figure 3, which illustrates the Deflate method through a flowchart [7].

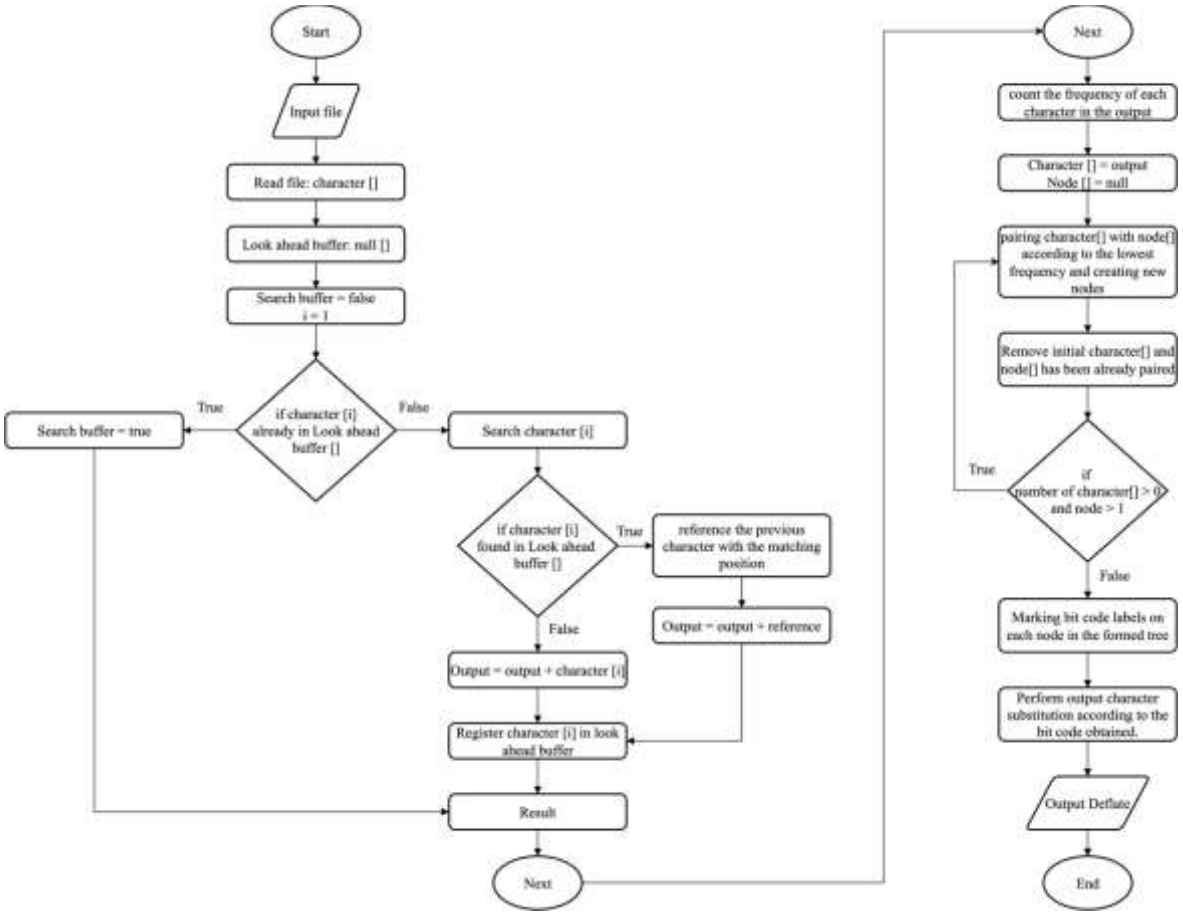

Fig. 3. Deflate method flowchart

The implementation of the document compression test utilizing the Deflate algorithm was executed through the Python programming language, employing the Zlib library as the core compression tool. The Zlib library encapsulates the complete Deflate compression and decompression process, thereby facilitating accurate, reliable, and consistent results without necessitating the manual reconstruction of the algorithm. This integration streamlines the development process and ensures adherence to the established Deflate standards. By leveraging Zlib, the study was able to focus on evaluating the algorithm's performance across diverse document formats, while relying on a robust and widely accepted Python-based implementation. An example of how the Deflate algorithm is implemented through Python using the Zlib library is provided to illustrate this approach.

```
1  import zlib
2
3  def compress_deflate(data):
4            return zlibcompress(data, level=9)
5
6  def decompress_deflate(data):
7            return zlib.decompress(data)
8
```

The flowchart in Figure 3 maps directly to the implementation of the Deflate algorithm using the Python **`zlib`** library. The stages labeled as compression input reading, Deflate-based encoding, and output file generation correspond to the sequential use of **`zlib.compress()`** (for producing the compressed byte stream) and **`zlib.decompress()`** (for restoring the original data), wrapped in file I/O operations that read the source document and write the resulting compressed or decompressed files. Thus, each logical block in the flowchart represents a concrete step in the Zlib-based processing pipeline implemented in the prototype application.

3.5 Reporting the Conclusions

The final stage involves reporting the findings derived from the comprehensive testing conducted using the developed application for comparing compression methods across various document formats. This phase focuses on systematically presenting the evaluated performance metrics of the compression algorithms applied to each prepared test document. The outcomes of the tests are consolidated into a detailed results table, complemented by graphical visualizations that illustrate the comparative compression efficiency of each method for all tested document types. These results, as summarized in Table 7 and visually depicted in Figure 4, provide clear and accessible insights into the relative effectiveness and practical implications of the evaluated compression techniques.

Table 7. Test Results

No	File Name	File Type	Initial Size	Size After Compression	Compression Time (Seconds)	Compression Ratio (%)
1	ujisatu	.docx	20 MB	19.82 MB	0.86	1.48
2	ujidua	.docx	7.6 MB	7.03 MB	0.32	0.59
3	ujitiga	.docx	27.15 MB	26.82 MB	1.23	1.23
4	ujisatu	.pdf	2.8 MB	2.67 MB	0.16	7.15
5	ujidua	.pdf	2.4 MB	2.11 MB	0.11	7.82
6	ujitiga	.pdf	4.9 MB	4.67 MB	0.21	4.47
7	ujilzw	.txt	939 KB	3.79 KB	0.01	99.60
8	ujideflate	.txt	796 KB	4 KB	0.01	99.45
9	ujikombinasi	.txt	1.6 MB	8.56 KB	0.02	99.51
10	ujisatuppt	.pptx	223 KB	194.28 KB	0.02	12.74
11	ujiduappt	.pptx	5.7 MB	5.60 MB	0.36	2.59
12	ujitigappt	.pptx	8.2 MB	7.99 MB	0.40	2.67

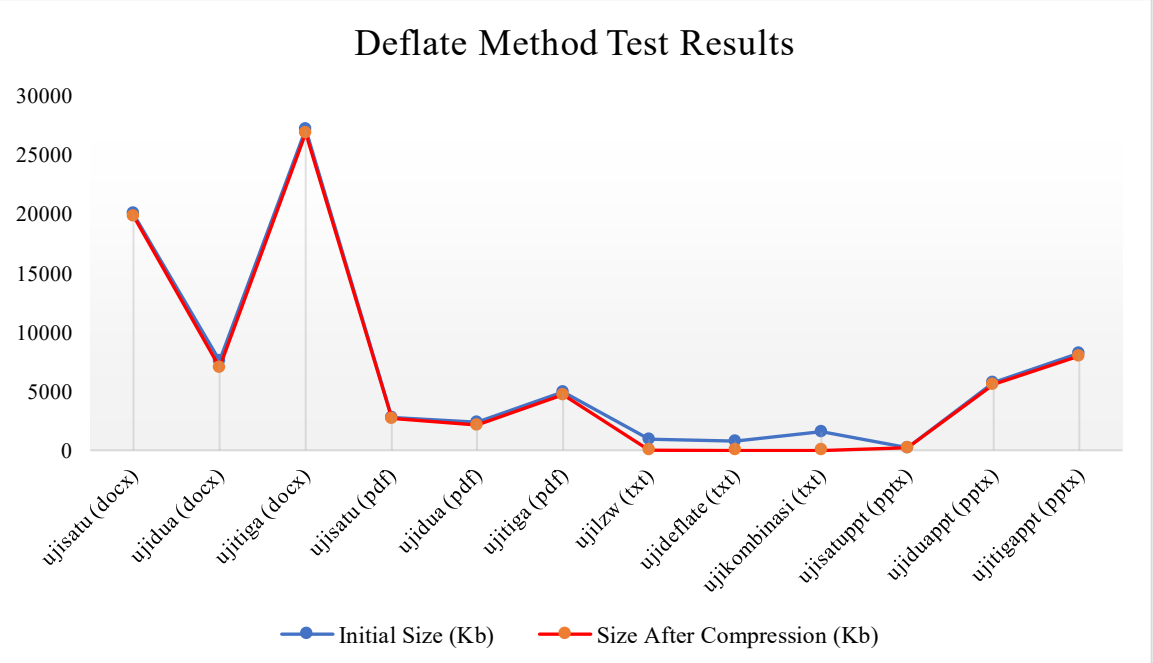

Fig. 4. Comparison of Test Results

4. Conclusion

This study was conducted to evaluate the performance of the Deflate algorithm in compressing various document formats, focusing primarily on the compression speed and quality achieved. The experimental results indicate that Deflate is highly effective in compressing text-based documents, such as .txt files, where compression ratios frequently exceed 90%, demonstrating the algorithm's efficiency in handling repetitive or patterned textual data. Conversely, its effectiveness diminishes significantly when applied to documents containing image elements, including formats like .docx, .pdf, and .pptx, regardless of whether the images are standard or high definition, as evidenced by considerably lower compression ratios in these cases. Despite this limitation, the Deflate algorithm performs the compression and decompression processes with relative speed and stability, supporting its applicability in large-scale document processing environments where time efficiency is crucial. A notable constraint of this research is the absence of a comparative analysis with other compression algorithms, which suggests a potential avenue for future work involving the development of hybrid compression techniques that integrate Deflate with alternative methods better suited for image-rich content. This direction is particularly relevant in contemporary industry workflows, where cloud storage services and collaborative document platforms routinely handle large office files dominated by embedded media (such as high-resolution images and graphics), and where relying solely on Deflate for such mixed-content documents may no longer provide optimal storage efficiency. Such hybrid approaches could enhance compression performance across diverse document types. Based on the findings, Deflate remains a suitable choice for implementation in archiving systems and applications dealing primarily with text-based digital documents.

References

[1] R. Lakshminarayanan, B. Kumar, and M. Raju, "Cloud Computing Benefits for Educational Institutions," May 2013. doi: https://doi.org/10.48550/arXiv.1305.2616.

[2] Y. P. Widyastuti, A. A. Setiyanti, K. Satya Wacana, D. Universitas, K. Satya, and W. Abstract, "Perancangan Private Cloud Storage Menggunakan Nextcloud untuk Meningkatkan Kinerja Administrasi di Sekolah," *Jurnal Ilmiah Wahana Pendidikan*, vol. 10, no. 15, pp. 697–709, 2024, doi: 10.5281/zenodo.13834069.

[3] A. Hanif, E. Wahyudi, H. Hadianto, and L. Martanto, "Performance Comparison of Lossless Data Compression Algorithms Using Compression Ratios and Space Savings," *Journal of Information and Technology (J-INTECH)*, vol. 11, no. 1, Jul. 2023, doi: https://doi.org/10.32664/j-intech.v11i1.863.

[4] J. Sitorus, "Implementasi Kompresi File PDF Menerapkan Algoritma Interpolative Coding," in *Konferensi Nasional Teknologi Informasi dan Komputer*, Medan: STMIK Budi Darma, Aug. 2024. doi: 10.30865/komik.v7i1.8057.

[5] R. A. Purba and L. Sitorus, "Analisis Perbandingan Algoritma Arithmetic Coding Dengan Algoritma Lempel Ziv Welch (Lzw) Dalam Kompresi Teks," *Jurnal Teknik Informatika UNIKA Santo Thomas*, vol. 3, no. 2, pp. 158–165, 2018.

[6] B. Alfredo Silaban, "Analisa Kompresi File Teks Dengan Kombinasi Metode Burrows-Wheeler Transform Dan Shannon-Fano," in *Konferensi Nasional Teknologi Informasi dan Komputer*, Medan: STMIK Budi Darma, Jul. 2022, pp. 707–715. doi: 10.30865/komik.v6i1.5760.

[7] D. A. Utomo, I. Kenedi, and J. Jumadi, "Perancangan Aplikasi Kompresi Menggunakan Metode Deflate," in *Seminar Nasional Ilmu Komputer (SNASIKOM)*, Bangkulu: Fakultas Ilmu Komputer Universitas Dehasen Bengkulu, Jul. 2021, pp. 212–19. Accessed: Aug. 01, 2025. [Online]. Available: https://proceeding.unived.ac.id/index.php/snasikom/article/view/63

[8] M. R. Irliansyah, S. D. Nasution, and K. Ulfa, "Penerapan Metode Deflate dan Algoritma Goldbach Codes Dalam Kompresi File Teks," in *KOMIK (Konferensi Nasional Teknologi Informasi dan Komputer)*, STMIK Budi Darma, Oct. 2017. doi: https://doi.org/10.30865/komik.v1i1.495.

[9] E. S. Panggabean, "Analisa Perbandingan Algoritma Lempel Ziv Welch Dan Algoritma Deflate Pada File Teks Dengan Metode Independent Sample T-Test," *Jurnal Pelita Informatika*, vol. 6, no. 3, 2018, Accessed: Aug. 02, 2025. [Online]. Available: https://ejurnal.stmik-budidarma.ac.id/pelita/article/view/622

[10] H. Yang, G. Qin, and Y. Hu, "Compression Performance Analysis of Different File Formats," Jul. 2023, [Online]. Available: http://arxiv.org/abs/2308.12275

[11] C. P. Nugraha, R. Gunawan Santosa, and L. Chrisantyo, "PERBANDINGAN METODE LZ77, METODE HUFFMAN DAN METODE DEFLATE TERHADAP KOMPRESI DATA TEKS," *Jurnal Informatika*, vol. 10, no. 2, 2014, doi: http://dx.doi.org/10.21460/inf.2014.102.327.

[12] X. Delaunay, A. Courtois, and F. Gouillon, "Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files," *Geosci Model Dev*, vol. 12, no. 9, pp. 4099–4113, Sep. 2019, doi: 10.5194/gmd-12-4099-2019.
[13] X. Kai and Z. Yuxiang, "Improving the performance of 3D image model compression based on optimized DEFLATE algorithm," *Sci Rep*, vol. 14, no. 1, Dec. 2024, doi: 10.1038/s41598-024-65539-7.
[14] K. Sayood, *Introduction to Data Compression*, Third Edition. Morgan Kaufmann, 2006.
[15] D. . Salomon, Giovanni. Motta, and David. Bryant, *Handbook of data compression*. Springer, 2010.
[16] N. Sharma and U. Batra, "EVALUATION OF LOSSLESS ALGORITHMS FOR DATA COMPRESSION," ZIbeline International Publishing, Jan. 2020, pp. 40–44. doi: 10.26480/etit.02.2020.40.44.
[17] P. S. Gray, J. B. Williamson, D. A. Karp, and J. R. Dalphin, *The research imagination : an introduction to qualitative and quantitative methods*, 1st ed. Cambridge University Press, 2007.
[18] R. M. Fauzi and F. Nurpandi, "Perancangan dan Pembangunan Aplikasi Rekrutmen Asisten Laboratorium Berbasis Mobile," *Media Jurnal Informatika*, vol. 11, no. 2, p. 15, Aug. 2020, doi: 10.35194/mji.v11i2.1015.
[19] Kurniati, "Penerapan Metode Prototype Pada Perancangan Sistem," *Journal of Software Engineering Ampera*, vol. 2, no. 1, Feb. 2021, doi: https://doi.org/10.51519/journalsea.v2i1.89.
[20] M. F. Arsa, A. S. Abdullah, and J. Rejito, "Pengembangan Sistem Informasi Geografis Kebun Binatang Berbasis Progressive Web Application (PWA) dengan Metode Prototype (Studi Kasus Kebun Binatang Bandung)," *Jurnal Nasional Teknologi dan Sistem Informasi*, vol. 7, no. 3, pp. 119–129, Dec. 2021, doi: 10.25077/teknosi.v7i3.2021.119-129.
[21] A. Maulana, "Analisis Perbandingan Algoritma Half Byte dan Algoritma Deflate Pada File Dengan Metode Independent Sampe T-Test," *Majalah Ilmiah INTI*, vol. 6, no. 2, Feb. 2019, Accessed: Aug. 02, 2025. [Online]. Available: https://ejurnal.stmik-budidarma.ac.id/inti/article/view/1411
[22] A. Bahrudin, P. Permata, and J. Jupriyadi, "Optimasi Arsip Penyimpanan Dokumen Foto Menggunakan Algoritma Kompresi Deflate (Studi Kasus: Studio Muezzart)," *Jurnal Ilmiah Infrastruktur Teknologi Informasi*, vol. 1, no. 2, pp. 14–18, Dec. 2020, doi: 10.33365/jiiti.v1i2.582.
[23] C. P. Nugraha, R. G. Santosa, and L. Chrisantyo A.A., "PERBANDINGAN METODE LZ77, METODE HUFFMAN DAN METODE DEFLATE TERHADAP KOMPRESI DATA TEKS," *Jurnal Informatika*, vol. 10, no. 2, Jan. 2015, doi: 10.21460/inf.2014.102.327.
[24] D. Oktaviani *et al.*, "Implementasi Kompresi Data dengan Modifikasi Algoritma Lempel-Ziv-Welch (LZW) untuk File Dokumen," *Journal of Informatics and Computer Science*, vol. 2, no. 1, Dec. 2019, doi: https://doi.org/10.26740/jinacs.v1n03.p128-137.
[25] R. D. Brown, "Reconstructing corrupt Deflated files," in *DFRWS 2011 Annual Conference*, Digital Forensic Research Workshop, 2011. doi: 10.1016/j.diin.2011.05.015.