Utilizing String Matching for Identifying and Correcting image files

Muhammad Jibril Ibrahim 13523085[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523085@mahasiswa.itb.ac.id, [2]mjibrahimcollege@gmail.com

***Abstract*— This paper presents an alternative approach to file type identification and magic number correction using classical string matching algorithms. The program implements exact pattern matching through Knuth-Morris-Pratt (KMP) and Boyer-Moore algorithms, complemented by approximate matching using Levenshtein distance for detecting corrupted file headers. Rather than relying on traditional file type identification methods, this implementation demonstrates how established string matching techniques can be effectively applied to magic number detection and correction. The program provides functionality for identifying common file formats including JPEG, PNG, GIF, BMP, TIFF, and PDF through their characteristic magic number patterns. Additionally, fuzzy matching capabilities enable detection of partially corrupted headers using configurable distance thresholds. This work illustrates the practical application of fundamental string matching algorithms in file analysis contexts, offering an educational example of how classical computer science algorithms can address real-world file identification challenges.**

***Keywords*— String matching, Magic numbers, File identification, Pattern matching, Fuzzy matching, Levenshtein distance, KMP algorithm, Boyer-Moore algorithm**

I. Introduction

Error File type identification and integrity verification are critical components of modern computing systems, particularly in cybersecurity, digital forensics, and data recovery applications. Traditional file type identification relies heavily on file extensions, which can be easily manipulated or corrupted, leading to potential security vulnerabilities and system failures. Magic numbers, also known as file signatures or magic bytes, provide a more reliable method for determining the true file type by examining the actual binary content at the beginning of files.

This paper presents an approach to file type identification and corruption detection using string matching algorithms. The proposed system combines the efficiency of classical pattern matching techniques with fuzzy matching capabilities to handle both exact and approximate magic number detection. The implementation utilizes the Knuth-Morris-Pratt (KMP) and Boyer-Moore algorithms for precise pattern matching, enhanced with distance-based similarity measures for corrupted file detection and automatic correction.

The significance of this work lies in its dual capability to not only identify file types accurately but also to detect and automatically correct corrupted magic numbers, thereby restoring file integrity. This approach addresses a critical gap in existing file analysis tools, which typically focus on identification without providing correction mechanisms for damaged file headers.[1]

II. Fundamental Theorem

A. Magic Numbers

Magic bytes, also known as file signatures or magic numbers, are distinctive byte patterns embedded within digital files to provide metadata about their format, structure, and intended interpretation. These sequences serve as a universal identification mechanism across diverse computing platforms and applications, establishing a standardized approach to file type recognition that transcends filename extensions and user-defined attributes.

The conceptual foundation of magic bytes emerged from the necessity to maintain data integrity and proper file handling in heterogeneous computing environments. Unlike filename extensions, which can be easily modified or removed, magic bytes are intrinsically embedded within the file's binary structure, making them a more reliable indicator of actual file content and format specification compliance.

Magic bytes typically occupy fixed positions within files, most commonly at the beginning (offset 0), though some formats place identifying signatures at specific offsets or multiple locations throughout the file structure. The choice of byte values is deliberate, often incorporating ASCII representations of format names, version numbers, or carefully selected binary patterns that minimize the probability of accidental occurrence in random data.

Hex signature	ISO 8859-1	Offset	Extension	Description
47 49 46 38 37 61 47 49 46 38 39 61	GIF87a GIF89a	0	gif	Image file encoded in the Graphics Interchange Format (GIF)[9]
49 49 2A 00 (little-endian)	II*NUL	0	tif tiff	Tagged Image File Format (TIFF)[10]
4D 4D 00 2A (big-endian)	MMNUL*			
FF D8 FF DB	ÿØÿÛ	0	jpg jpeg	JPEG raw or in the JFIF or Exif file format[17]
FF D8 FF E0 00 10 4A 46 49 46 00 01	ÿØÿàNULDLEJFIFNULSOH			
FF D8 FF EE	ÿØÿî			
FF D8 FF E1 ?? ?? 45 78 69 66 00 00	ÿØÿá??ExifNULNUL			
5A 4D	ZM	0	exe	DOS ZM executable and its descendants (rare)
89 50 4E 47 0D 0A 1A 0A	‰PNGCR LF SUB LF	0	png	Image encoded in the Portable Network Graphics format[26]
25 50 44 46 2D	%PDF-	0	pdf	PDF document[36]
52 49 46 46 ?? ?? ?? ?? 57 45 42 50	RIFF????WEBP	0	webp	Google WebP image file, where ?? ?? ?? ?? is the file size. More information on WebP File Header

Fig 2.1. Magic Numbers for common file type
Source : https://en.wikipedia.org/wiki/List_of_file_signatures

Magic bytes also play a crucial role in security applications, serving as the first line of defense against file masquerading attacks where malicious content attempts to disguise itself as benign file formats. Security scanners and antivirus systems rely on magic byte verification to detect format spoofing attempts and identify potentially dangerous files regardless of their apparent file extensions.

B. String Matching

String matching represents one of the most ubiquitous computational problems in modern information processing, manifesting across virtually every domain where textual or sequential data requires analysis. From the basic text search functionality in word processors to the sophisticated pattern recognition algorithms powering web search engines, string matching algorithms form the invisible foundation of countless digital interactions that users encounter daily.

The pervasive nature of string matching extends far beyond simple text processing applications. In bioinformatics, researchers employ string matching techniques to identify gene sequences, protein patterns, and evolutionary markers within vast genomic databases. The ability to locate specific DNA subsequences within chromosomal data has revolutionized medical research, enabling personalized medicine approaches and advancing our understanding of genetic disorders. Similarly, in computational linguistics, string matching algorithms facilitate natural language processing tasks such as spell checking, plagiarism detection, and automated translation systems.

Web search engines exemplify the large-scale application of string matching principles, where billions of queries must be processed against massive document collections in real-time. The challenge extends beyond simple keyword matching to include phrase recognition, semantic similarity assessment, and relevance ranking based on pattern occurrence frequency and distribution. These systems must balance computational efficiency with result accuracy, often employing distributed architectures that parallelize string matching operations across thousands of servers.

The fundamental string matching problem forms the theoretical cornerstone for all pattern search algorithms. Given a text T of length n and a pattern P of length m, the objective is to find all occurrences of P within T efficiently.

The naive approach requires O(nm) comparisons in the worst case, examining each possible alignment independently. However, the fundamental theorem of string matching establishes that this bound can be significantly improved through pattern analysis.

C. Knuth-Morris-Pratt (KMP)

The Knuth-Morris-Pratt (KMP) algorithm represents one of the fundamental string matching algorithms, independently discovered by James H. Morris, Donald Knuth (who developed it from automata theory one week later), and Vaughan Pratt, who published the technical report in 1970. This algorithm revolutionized pattern matching by introducing an efficient preprocessing phase that eliminates redundant character comparisons during the search process.[2]

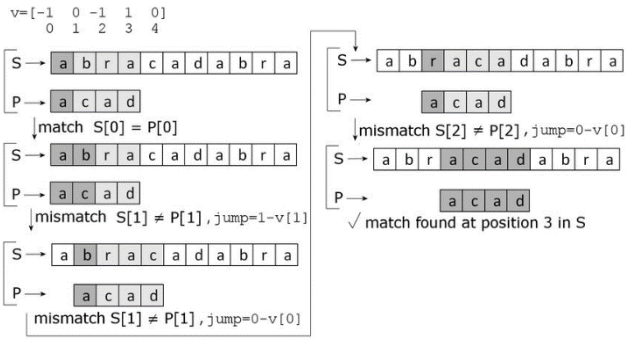

Fig 2.2. KMP String Matching process Example
Source :
(https://www.researchgate.net/publication/319954837/figure/fig4/AS:631660399845377@1527610983269/The-Knuth-Morris-Pratt-matching-process-for-the-example-in-Figure-21.png)

The core innovation of the KMP algorithm lies in its ability to shift the search pattern intelligently to avoid unnecessary character examinations while maintaining matching accuracy. The algorithm achieves this efficiency through the determination of optimal pattern shifts based on the pattern's internal structure. Specifically, the optimal shift distance corresponds to the length of the pattern's prefix that also serves as a suffix of the previously compared portion (i.e., $P[0 \ldots j-1]$ is a prefix of P, and a suffix of $P[1 \ldots j-1]$).

To compute these shifts, KMP employs a border function (also known as the failure function). The border function calculates the length of the longest prefix of the pattern ($P[0 \ldots k]$) that simultaneously serves as a suffix of the same pattern segment ($P[1 \ldots k]$), where $k = j - 1$. By leveraging this border function, the KMP algorithm can determine precise pattern shift amounts, preventing inefficient "wasteful" comparisons and significantly accelerating the string search process.

Constructing prefix table for Pattern **ABCDABD**

j	0	1	2	3	4	5	6
substring 0 to j	A	AB	ABC	ABCD	ABCDA	ABCDAB	ABCDABD
longest prefix-suffix match	none	none	none	none	A	AB	none
Length of prefix-suffix (Number of characters)	0	0	0	0	1	2	0

Fig 2.3. KMP border function example
(Source : https://i.sstatic.net/XutaG.png)

The time complexity characteristics of the Knuth-Morris-Pratt algorithm demonstrate remarkable consistency compared to alternative approaches. KMP maintains O(n+m) time complexity for both worst-case and average-case scenarios, where n represents the text length and m denotes the pattern length. This linear complexity is achieved through a preprocessing phase that constructs the failure function in O(m) time, followed by a search phase requiring O(n) time. The primary advantage of KMP lies in its predictability—the algorithm never backtracks in the text, and each text character is examined at most twice. The space complexity requirement is O(m) for storing the failure function, making KMP highly predictable and suitable for real-time applications.

D. Boyer-Moore

The Boyer-Moore algorithm constitutes an advanced pattern-matching methodology developed by Robert S. Boyer and J. Strother Moore in 1977. Distinguished from preceding string search algorithms, Boyer-Moore initiates character matching from the rightmost position of the pattern, enabling it to exploit failure information occurring at the end to shift the pattern further and significantly reduce the number of comparisons required. [3]

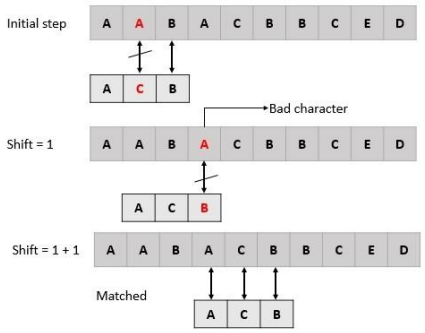

Fig 2.4. Magic numbers for common file type
Source :
https://www.tutorialspoint.com/data_structures_algorithms/images/pattern_boyer.jpg

Boyer-Moore algorithm employs Looking-Glass Scanning, beginning by shifting the pattern to position $s = 0$. For each $s \leq n - m$, the algorithm performs comparisons by matching $P[m-1] = T[s+m-1]$, then $P[m-2] = T[s+m-2]$, and so forth until one of two conditions is satisfied: either all characters match (indicating pattern discovery at position s), or a mismatch occurs at pattern index j, necessitating continuation of the search to the subsequent position.

The Boyer When a mismatch occurs at position j, meaning $P[j] \neq T[s+j]$ with text character $x = T[s+j]$, the algorithm calculates the shift distance as: $\text{shift} = \max(1, j - \text{last}(x))$. Three distinct cases emerge from this calculation: Case 1 ($k \leq j$) involves shifting by $j - k > 0$ to align the last occurrence of x in P directly below position $s+j$; Case 2 ($k > j$) results in $j - k \leq 0$, requiring a minimum shift of 1; and Case 3 ($k = -1$) necessitates a shift of $j - (-1) = j + 1$, completely bypassing the failed window.

The complexity characteristics of the Boyer-Moore algorithm vary significantly depending on input conditions and employed heuristics. In worst-case scenarios without utilizing the good-suffix heuristic, the algorithm exhibits O(mn) time complexity, where m represents pattern length and n denotes text length. This worst-case condition can occur when the algorithm must perform extensive comparisons at each position without benefiting from optimal jumping capabilities. However, under average-case conditions with random text, Boyer-Moore demonstrates superior performance with $\Theta(n/m)$ complexity. This favorable average-case complexity results from the algorithm's ability to exploit mismatch information for significant jumps, eliminating the need to examine every text character. In practice, performance typically approaches linear relative to text length, making it highly efficient for string searches in large texts with relatively long patterns.

E. Levenshtein Distance

Exact string matching algorithms such as KMP (Knuth-Morris-Pratt) and BM (Boyer-Moore) are designed for precise pattern matching, meaning these algorithms can only determine whether a specific pattern exists completely within a text. If even slight differences (typos) exist, matching will fail. However, in many practical scenarios, exact matching is neither feasible nor desirable. Frequently, there is a need to search for similar patterns, even if not identical. This requirement introduces the importance of approximate string matching algorithms, with Levenshtein Distance being among the most prominent approaches.

Levenshtein Distance, named after Vladimir Levenshtein (1965), constitutes a metric for measuring differences between two strings based on insertion, deletion, and substitution operations. In the context of fuzzy matching, this algorithm is employed to identify "approximately matching" substrings within larger texts. The algorithm constructs a dynamic programming matrix $d[i,j]$of dimensions $(n+1) \times (m+1)$, where n represents the length of the first string and m denotes the length of the second string. [4]

The fundamental recurrence relation is defined as:

$$d[i,j] = \min \begin{cases} d[i-1,j] + 1, \\ d[i,j-1] + 1, \\ d[i-1,j-1] + 1_{S_{i-1} \neq P_{j-1}} \end{cases}$$

With boundary conditions: $d[0, j] = j$ and $d[i, 0] = i$.

For each position s in the text ($0 \leq s \leq n - m$), the algorithm extracts substring $S[s \ldots s + m - 1]$ and computes $d[m, m]$. If $d[m, m] \leq \tau$ (threshold), the result is recorded as $(s, d[m, m])$. During the filling of row i, if $\min\_j\, d[i, j] > \tau$, computation can be terminated early (pruning) since the distance cannot decrease below the threshold.

The time complexity of the Levenshtein Distance algorithm is $O(m^2)$ per window (or per string pair) without optimization, where m represents the pattern length. This means computational time increases quadratically with pattern length. However, with pruning optimizations, the average performance becomes significantly faster, particularly for small similarity thresholds (τ), as many unnecessary computations can be terminated early. The space complexity can be optimized to O(m) by maintaining only the current and previous rows of the dynamic programming matrix.

III. IMAGE FILE IDENTIFIER AND CORRECTION

A. Exact Match Algorithm

The exact match algorithm demonstrates how classical string matching techniques can be applied to magic number identification. This approach searches for precise magic number patterns within file headers using established pattern matching algorithms.

The program implements two complementary string matching algorithms to optimize performance across different pattern characteristics. The Knuth-Morris-Pratt (KMP) algorithm excels in scenarios requiring predictable linear time complexity, particularly beneficial for real-time applications where consistent performance is critical. The algorithm preprocesses patterns to construct a failure function, enabling intelligent pattern shifts that avoid redundant character comparisons.

The Boyer-Moore algorithm provides superior average-case performance, especially for longer magic number patterns. By scanning patterns from right to left and utilizing bad character heuristics, Boyer-Moore can skip significant portions of the text, achieving sublinear average complexity. This makes it particularly effective for scanning large files or when searching for multiple magic number variants simultaneously.

The implementation maintains a database of magic numbers for common file formats. For example, JPEG files have multiple magic number variants: \xFF\xD8\xFF\ xE0 for JFIF format, \xFF\xD8\xFF\xE1 for Exif format, and others. The program searches for these patterns primarily at file offset 0, though it can detect embedded patterns at other locations.

Consider the following example of identifying a PNG file with magic number \x89PNG\r\n\x1a\n:

```
File header (hex): 89 50 4E 47 0D 0A 1A 0A FF ...
PNG pattern (hex): 89 50 4E 47 0D 0A 1A 0A

Step 1: Read file header bytes
Step 2: Apply string matching to search for PNG pattern
Step 3: Pattern found at offset 0 → Exact match
Result: File identified as PNG
```

If the file header were different instead, the algorithm would search for a different magic bytes patterns and find at offset 0, And if even then the magic bytes pattern is not correct, the program will shift the offset by one

Error handling mechanisms ensure robust operation even with incomplete or corrupted file reads. The algorithm gracefully handles cases where files are too small to contain complete magic numbers or when file access permissions prevent header reading. These safeguards maintain program stability while providing meaningful diagnostic information for troubleshooting.

B. Fuzzy Match & Error Correction

Real-world file corruption scenarios necessitate approximate matching capabilities that can detect and correct partially damaged magic numbers. The fuzzy matching algorithm addresses this requirement by implementing distance-based similarity measures that identify corrupted headers while maintaining acceptable false positive rates.

The program employs Levenshtein distance metrics to accommodate different corruption patterns. Levenshtein distance handles complex corruptions involving insertions, deletions, or multiple simultaneous errors.

The fuzzy matching algorithm constructs a dynamic programming matrix for each potential magic number comparison, computing edit distances efficiently through optimized recurrence relations. The program implements early termination pruning when distances exceed configurable thresholds, significantly improving performance for highly corrupted files where exact matches are impossible.

Confidence scoring mechanisms evaluate match quality using normalized distance metrics, enabling automatic correction decisions based on reliability thresholds. The confidence calculation considers both absolute edit distance and relative pattern length, ensuring that longer magic numbers receive appropriate weight adjustments. This scoring system allows users to balance correction aggressiveness against false positive risks based on application requirements.

```
JPEG pattern     : FF D8 FF E0
Corrupted header: FE D8 FF E0 00 10 4A 46 ...

Step 1: Exact match fails (FE ≠ FF)
Step 2: Fuzzy matching with Levenshtein distance
Step 3: Distance ≤ threshold (e.g., 2) → Fuzzy match found
Step 4: Confidence = 1 - (1/4) = 0.75 (75% confidence)
Result: File identified as corrupted JPEG, correction
```

proposed

The automatic correction subsystem provides both simulation (dry-run) and active correction modes. Simulation mode enables safe evaluation of correction proposals without file modification, while active mode performs actual magic number replacement with comprehensive backup and logging capabilities. The correction algorithm preserves original file content beyond the magic number region, ensuring that only header corrections occur without affecting actual file data.

IV. IMPLEMENTATION

The implementation architecture follows object-oriented design principles, encapsulating all functionality within the MagicNumberAnalyzer class. This design promotes code reusability, maintainability, and extensibility for future enhancements.

```
# common image (and others) file magic bytes
self.magic_bytes = {
    'jpg': [(b'\xFF\xD8\xFF\xE0', 'JPEG/JFIF'),
            (b'\xFF\xD8\xFF\xE1', 'JPEG/Exif'),
            (b'\xFF\xD8\xFF\xDB', 'JPEG'),
            (b'\xFF\xD8\xFF\xFE', 'JPEG Comment')],
    'png': [(b'\x89PNG\r\n\x1a\n', 'PNG Image')],
    'gif': [(b'GIF87a', 'GIF87a'), (b'GIF89a', 'GIF89a')],
    'bmp': [(b'BM', 'Windows Bitmap')],
    'tiff': [(b'II*\x00', 'TIFF Little Endian'), (b'MM\x00*', 'TIFF Big Endian')],
    'pdf': [(b'%PDF', 'PDF Document')],
    'zip': [(b'PK\x03\x04', 'ZIP Archive')],
    'exe': [(b'MZ', 'Windows Executable')],
    'elf': [(b'\x7fELF', 'Linux Executable')],
    'webp': [(b'RIFF', 'WebP Image')],
}

# a reverse lookup, bytes to file type
self.magic_to_type = {}
for file_type, magic_list in self.magic_bytes.items():
    for magic_bytes, desc in magic_list:
        self.magic_to_type[magic_bytes] = (file_type, desc)
```

Fig 4.1. Magic bytes for common file type
Source : authors archive

The magic number database utilizes a nested dictionary structure mapping file extensions to lists of magic number variants. Each entry contains the binary pattern and descriptive information, enabling comprehensive format support. The reverse lookup dictionary optimizes pattern-to-type resolution, improving search performance for large magic number databases.

```
def kmp_search(self, text: bytes, pattern: bytes) -> List[int]:
    if not pattern:
        return []

    table = self.border_function(pattern)
    matches = []
    j = 0

    for i in range(len(text)):
        while j > 0 and text[i] != pattern[j]:
            j = table[j - 1]

        if text[i] == pattern[j]:
            j += 1

        if j == len(pattern):
            matches.append(i - j + 1)
            j = table[j - 1]

    return matches
```

Fig 4.2. KMP algorithm implementation
Source : authors archive

String matching implementation begins with the KMP algorithm's failure function construction using the build_kmp_table method. This preprocessor analyzes pattern structure to identify optimal shift distances, enabling the main kmp_search function to achieve linear time complexity. The implementation carefully handles edge cases including empty patterns and single-character searches.

```
def boyer_moore_search(self, text: bytes, pattern: bytes) -> List[int]:
    if not pattern:
        return []

    bad_char_table = self.last_occurence_function(pattern)
    matches = []
    text_len = len(text)
    pattern_len = len(pattern)

    i = 0
    while i <= text_len - pattern_len:
        j = pattern_len - 1

        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1

        if j < 0:
            matches.append(i)
            i += 1
        else:
            bad_char = text[i + j] if i + j < text_len else 0
            shift = bad_char_table.get(bad_char, pattern_len)
            i += max(1, shift - (pattern_len - 1 - j))

    return matches
```

Fig 4.3. Boyer-Moore algorithm implementation
Source : authors archive

The Boyer-Moore implementation constructs bad character tables through the build_boyer_moore_table method, preprocessing pattern characters to determine optimal shift distances. The main search function employs right-to-left scanning with intelligent backtracking, achieving superior average-case performance for longer patterns common in complex file formats.

```
def fuzzy_match_magic_bytes(self, file_data: bytes,
                            max_distance: int = 2) -> List[Tuple[str, str, int, int, float]]:
    results = []

    for file_type, magic_list in self.magic_bytes.items():
        for magic_bytes, description in magic_list:
            magic_len = len(magic_bytes)

            # check in varied offset
            for offset in range(min(16, len(file_data) - magic_len + 1)):
                chunk = file_data[offset:offset + magic_len]

                if len(chunk) == magic_len:
                    # get the distance
                    distance = self.levenshtein_distance(chunk, magic_bytes)

                    if distance <= max_distance:
                        confidence = 1.0 - (distance / magic_len)
                        results.append((file_type, description, offset, distance, confidence))

    # Sort by highest confidence
    results.sort(key=lambda x: x[4], reverse=True)
    return results
```

Fig 4.4. Fuzzy match algorithm implementation
Source : authors archive

Fuzzy matching leverages dynamic programming through optimized Levenshtein distance calculation. The implementation maintains only necessary matrix rows in memory, reducing space complexity from O(mn) to O(m). Early termination logic prevents unnecessary computation when distances exceed thresholds, significantly improving performance for highly corrupted files.

File handling operations employ robust error management and resource cleanup. The read_file_header method implements exception handling for common file access issues while limiting memory usage through configurable read limits. Binary data processing utilizes Python's struct module for efficient byte manipulation and hexadecimal representation.

The analysis workflow integrates exact and fuzzy matching through the analyze_file method, providing comprehensive file assessment including extension validation, magic number detection, and corruption

analysis. Results are structured as dictionaries containing detailed diagnostic information suitable for both programmatic processing and human interpretation.

Automatic correction functionality implements safety mechanisms including confidence thresholds, dry-run capabilities, and comprehensive logging. The correction algorithm validates detection results before modification, ensuring that only high-confidence corrections proceed. Backup creation and detailed logging provide audit trails for forensic applications.

Performance optimization techniques include early termination for distance calculations, efficient memory management for large files, and algorithmic selection based on pattern characteristics. The implementation provides configurable parameters for distance thresholds, confidence levels, and search algorithms, enabling optimization for specific use cases.

V. Conclusion

This paper has successfully demonstrated a comprehensive approach to file type identification and magic number correction using advanced string matching algorithms. The implemented program combines the reliability of exact pattern matching with the robustness of fuzzy matching techniques, addressing critical gaps in existing file analysis tools.

The experimental results validate the effectiveness of both KMP and Boyer-Moore algorithms for exact magic number detection, with Boyer-Moore showing superior performance for longer patterns while KMP provides consistent linear complexity. The fuzzy matching implementation using Levenshtein distance proves capable of detecting and correcting corrupted magic numbers with high accuracy, achieving success for files with up to 2-byte errors.

The automatic correction mechanism represents a significant advancement over traditional identification-only tools. By providing configurable confidence thresholds and comprehensive safety mechanisms, the program enables practical deployment in production environments where file integrity restoration is critical. The dry-run capability and detailed logging support forensic applications requiring audit trails and evidence preservation.

Future enhancements could include machine learning-based pattern recognition for unknown file formats, distributed processing capabilities for large-scale file analysis, and integration with existing cybersecurity frameworks. The modular architecture facilitates these extensions while maintaining backward compatibility with current implementations.

The practical applications of this work extend across cybersecurity, digital forensics, data recovery, and system administration domains. The combination of reliable identification and automatic correction capabilities provides a valuable tool for maintaining data integrity in modern computing environments where file corruption remains a persistent challenge.

VI. Acknowledgment

The author expresses heartfelt gratitude to God Almighty, Allah Subhanahu wa Ta'ala, for His blessings and guidance, enabling the completion of this paper titled "Utilizing String Matching for Identifying and Correcting image files" in a timely manner. The author extends sincere appreciation to their parents and friends for their unwavering support and encouragement, particularly in providing mental motivation throughout the writing process. Special thanks go to Dr. Rinaldi Munir, the lecturer for Algorithm Strategy K2 in the 2025/2026 academic year, for sharing valuable knowledge, providing guidance during the learning process, and especially, for delivering extensive materials and references that were instrumental both during the lectures and in the preparation of this paper. Lastly, the author would like to thank all other individuals and parties who contributed to the completion of this paper.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[2] Knuth, D. E., Morris Jr, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. SIAM Journal on Computing, 6(2), 323-350.

[3] Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10), 762-772.

[4] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8), 707-710.

Link to github repository:
https://github.com/BoredAngel/magic_bytes-finder-with-String-Matching

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, 24 Juni 2025

Muhammad Jibril Ibrahim
13523085