International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

An Enhancement of Boyer-Moore Algorithm Using Hash Table

Domingo, John Michael M[1], Martin, Mark Dave C[2], Vivien, Agustin A[3]

[1,2,3]Computer Science Student at Pamantasan ng Lungsod ng Maynila

Abstract

Purpose: This study aims to enhance the Boyer-Moore algorithm by integrating hash tables to improve efficiency in handling large datasets and complex patterns. The enhancement addresses challenges of increased execution time and memory usage in text processing tasks.

Method: The proposed algorithm incorporates hash tables to optimize pattern preprocessing and matching, enabling faster data access and reducing redundant computations. Performance was evaluated by testing both the original and enhanced algorithms on datasets ranging from 100,000 to over 5 million characters.

Results: The enhanced algorithm demonstrated a 16.39% improvement in execution time, processing 5,289,118 characters in 1.64 seconds compared to 1.96 seconds for the original. Memory usage was also reduced by approximately 1,210 KB across dataset sizes, showcasing its scalability and resource efficiency.

Conclusion: The integration of hash-based structures into the Boyer-Moore algorithm significantly enhances its runtime and memory performance without compromising accuracy. These improvements make it highly suitable for real-time applications such as content moderation, search engines, and large-scale data analytics.

Recommendations: Future studies should explore hybrid algorithms, advanced data structures, and multilingual datasets to further optimize performance and validate adaptability.

Research Implications: The enhanced algorithm provides a robust solution for real-time text processing, supporting industries that rely on efficient, large-scale analytics.

Practical Implications: Industries such as data analytics and content moderation can adopt this algorithm to meet growing demands for scalable and efficient computational tools.

Social Implications: Improved text-processing efficiency supports faster, more accurate content moderation, fostering safer and more respectful online environments.

Keywords: Boyer-Moore, hash map, string matching, pattern, hash-based data structure, execution time.

INTRODUCTION

The Boyer-Moore algorithm employs a unique approach to string matching by combining the bad character and good suffix heuristics, enabling it to skip sections of text and significantly enhance search efficiency compared to naive methods (Abo-Ismail et al., 2014). Its effectiveness has been demonstrated through visual simulations, which illustrate how the algorithm compares strings from right to left, testing characters sequentially until a match is found. This efficiency makes the Boyer-Moore algorithm a widely recognized and frequently used tool for finding substrings within larger text bodies. However, despite

its popularity, the algorithm faces several challenges and limitations, prompting ongoing research into potential enhancements.

Among these challenges are its performance issues when dealing with patterns that exhibit duplication, frequent mismatches, or subtle variations. These factors can limit the effectiveness of the bad character and good suffix heuristics, especially in dynamic or linguistically diverse datasets (Jeong et al., 2020). Additionally, the preprocessing stage, which involves creating shift tables, can become computationally demanding for large or intricate patterns, further diminishing the algorithm's efficiency in complex scenarios (Berthold Vöcking et al.).

This study proposes an enhanced version of the Boyer-Moore algorithm by integrating a hash table into its operation. The hash based data structure is utilized to preprocess and index pattern substrings, storing their hash values in a compact, efficiently retrievable structure. This addition enables the algorithm to perform quick lookups and verify potential matches against hash values before performing full character-by-character comparisons. By doing so, the algorithm reduces unnecessary comparisons, particularly in repetitive or noisy text patterns. The integration also minimizes the overhead associated with large shift tables by replacing them with a more memory-efficient hashing mechanism.

Therefore, the objectives of this study include optimizing preprocessing steps, integrating adaptive heuristics, and applying pattern transformation techniques to further improve memory efficiency and runtime performance. These modifications collectively address the algorithm's limitations in both speed and memory usage, demonstrating measurable improvements when compared to other traditional

LITERATURE REVIEW

Boyer Moore Algorithm

The Boyer-Moore algorithm, introduced by Robert S. Boyer and J Strother Moore in 1977, is renowned for its efficiency in string searching. It preprocesses the pattern to be searched, allowing it to skip sections of the text, which results in a lower constant factor compared to other string search algorithms. This preprocessing step is particularly useful when the pattern is much shorter than the text or when the pattern persists across multiple searches (Boyer & Moore, 1977).

The algorithm follows a systematic process:

1. **Preprocessing Stage**: Construct tables for the bad character and good suffix rules to identify potential skips (Horspool, 1980).
2. **Matching Process**: Compare pattern characters from right to left and apply the heuristics upon mismatches (Sunday, 1990).
3. **Shifting**: Use the largest value suggested by the heuristics to align the pattern with the text efficiently (Cantone & Faro, 2003). This approach significantly reduces the time complexity for most string-matching scenarios, particularly with shorter patterns and longer texts (Pandey et al., 2016).

The algorithm enhances string matching by following a three-step process. First, it constructs tables for the bad character and good suffix rules to identify potential skips during matching. Then, it compares pattern characters from right to left, applying these rules when mismatches occur. Finally, it shifts the pattern efficiently by using the largest shift suggested by the heuristics, minimizing the number of comparisons. This approach significantly speeds up matching, especially when the pattern is shorter, and the text is longer.

Hash Based Data Structure

Hash-based data structures are crucial for optimizing string matching by enabling fast lookups and reducing time complexity. In algorithms like Rabin-Karp, hash maps (or dictionaries) are used to store the hash values of substrings, allowing for O(1) time complexity for lookups and avoiding repetitive comparisons (Rabin & Karp, 1987). Instead of recalculating the hash for every substring, hash maps can store precomputed values, allowing the algorithm to quickly check for matches. Additionally, these structures efficiently handle collisions using techniques like chaining or open addressing (Cormen et al., 2009). This results in significant performance improvements, especially when working with large texts, as demonstrated by algorithms like Rabin-Karp and Boyer-Moore (Boyer & Moore, 1977).

Hash Map

A **hash map** (or **hash table**) is a data structure that stores data in key-value pairs, leveraging a **hash function** to convert keys into unique indices where values are stored in an underlying array or table. This structure enables operations like **insertion**, **lookup**, and **deletion** to be performed in average constant time (O (1)), significantly improving the efficiency of data retrieval.

Key components of a hash map include:

1. Key: A unique identifier used to retrieve the corresponding value.
2. Value: The data associated with the key.
3. Hash Function: A function that converts the key into an index for quick retrieval from the underlying array.
4. Collision Handling: When two keys generate the same index, methods like chaining or open addressing manage these collisions by either linking multiple values at the same index or probing for an open spot in the table.

Hash maps are particularly useful in scenarios where fast data access is critical, such as in database indexing, caching, and symbol table management in compilers (Boyer & Moore, 1977; Cormen et al., 2009; Biazus, 2023). Efficient collision handling ensures that hash maps can scale well even with large data sets, making them an indispensable tool in computer science.

METHODOLOGY

This study enhances the Boyer-Moore algorithm by utilizing hash tables to improve performance in terms of execution time and memory usage. Hash tables are known for their efficiency in optimizing data lookups and reducing time complexity. Previous studies on hash-based string-matching algorithms have demonstrated significant improvements in processing patterns across both random and repetitive text sequences, making them ideal for large-scale text processing tasks.

To enhance the traditional algorithm, several optimizations were introduced, particularly in the preprocessing phase. Hash table functions were implemented to support the good and bad character heuristics by storing precomputed shift values. This integration ensures faster access to shift data and eliminates redundant computations during the pattern-matching process.

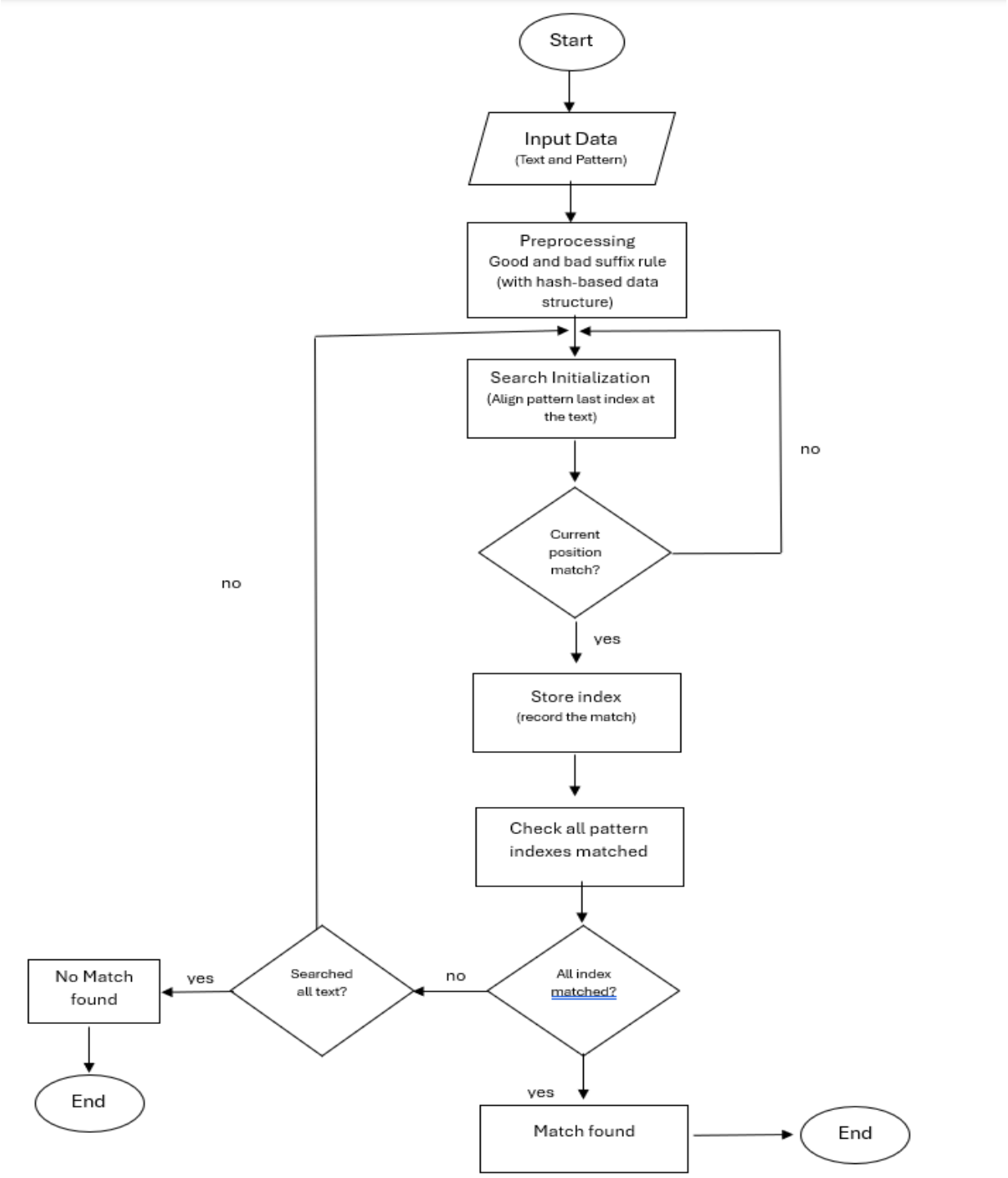

Figure 1: Flowchart of Boyer-Moore Algorithms with hash-based data structure.

Input Data: Text and pattern(s) are provided.
Preprocessing: The good and bad character rules are computed and stored using hash tables for efficient lookup.
Search Initialization: The pattern's last character is aligned with the current text position.
Pattern Matching:

- If the current position matches, the algorithm checks for full pattern alignment.
- Matches are recorded, and the text pointer shifts appropriately.

- If no match is found, the algorithm uses the precomputed hash table values to determine the next alignment efficiently.

Termination: The search ends when the entire text has been processed.

The enhanced Boyer-Moore algorithm will now be faster when dealing with large datasets due to the implementation of the hash table function in the searching preprocessing part of the algorithm.

The enhanced Boyer-Moore algorithm was evaluated by testing both the original and enhanced versions on datasets of varying sizes. The key metrics for evaluation were execution time and memory usage. Execution time was measured using a runtime test to capture only the active processing time, excluding any delays or idle time. Memory usage was assessed by tracking the amount of memory consumed during pattern preprocessing and matching operations. The results indicated that the enhanced algorithm exhibited improved efficiency for smaller datasets, with execution time reduced by up to 16.39% for larger datasets. Additionally, memory usage was optimized, with a reduction of approximately 1,210 KB across various dataset sizes. This methodology focused on assessing the efficiency, scalability, and resource utilization of the algorithms for a wide range of text-processing tasks.

RESULTS

The enhanced algorithm was implemented in Python and executed in a controlled testing environment. Both the original and enhanced algorithms were evaluated on datasets of varying sizes, ranging from 100,000 to 5,000,000 characters, created by repeating the input text to meet the desired size. The primary metric assessed was execution time, which is critical for evaluating the efficiency and scalability of text-processing algorithms.

The execution time was measured using a runtime test, which calculates the time taken by each algorithm to process the dataset. This method ensures that only the active processing time is captured, excluding any external delays or idle processes. The results indicate that the enhanced algorithm demonstrates improved efficiency for smaller datasets but shows comparable performance to the original algorithm as dataset sizes grow larger.

The input parameters used in the algorithm are as follows:

- **Input Data:** Text datasets of varying sizes, including 100,000, 300,000, 500,000, 700,000, 1,000,000 and 5,000,000,000 characters.
- **Pattern Data:** A set of patterns used for matching, including: "test", "efficient", "Boyer-Moore", "algorithm", "text", and "datasets".

The results of the execution time tests are summarized in the table below:

Input Size (Characters)	Original BM Algorithm Execution Time (s)	Enhanced BM Algorithm Execution Time (s)	Original BM Memory Usage(KB)	Enhanced BM Memory Usage(KB)
100,000	0.144974	0.147566	3300.12	3260.56
300,000	0.473896	0.490367	9900.36	9750.1
500,000	0.614778	0.569002	16500.6	16250.72

700,000	0.878821	0.750720	23100.84	22850.99
1,000,000	1.266654	1.254892	33000.0	32700.45
5,000,000	1.963442	1.641370	173420.56	172210.34

Table 1: Execution Time and Memory Usage Results for the Original and Enhanced Boyer-Moore Algorithms

Presents the execution times (in seconds) and memory usage (in KB) for the original and enhanced Boyer-Moore algorithms. The leftmost column specifies the size of the input data (characters). A lower execution time and memory usage indicate better performance. These results highlight the efficiency of the enhanced algorithm, particularly for larger datasets, while the original algorithm exhibits slightly better performance for medium-sized datasets. The findings are further illustrated in Figures 1 and 2.

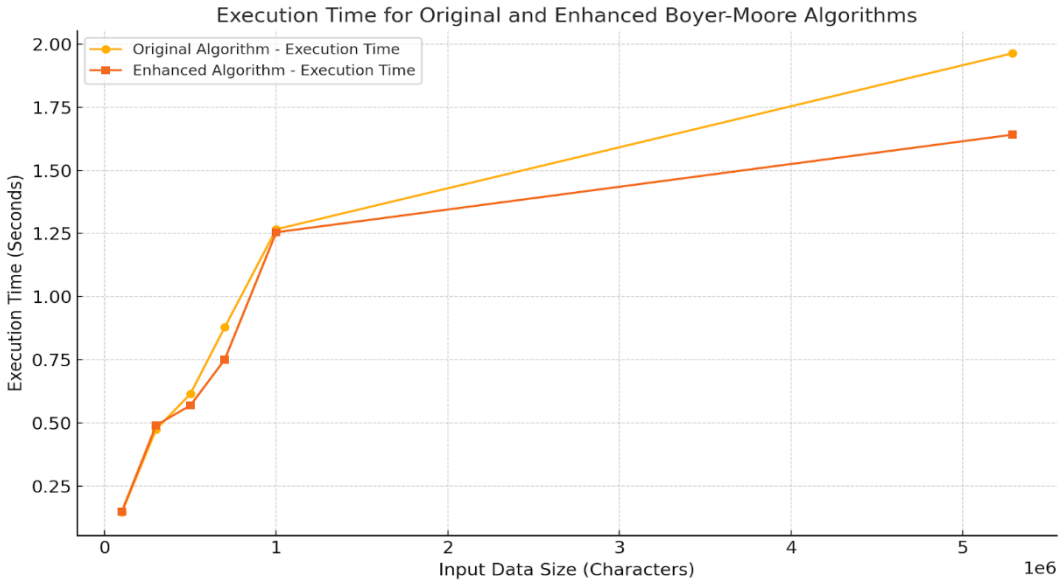

Figure 2: Runtime Results for Original and Enhanced Boyer-Moore Algorithms

Figure 2 visualizes the runtime results from Table 1, comparing the execution times for the Original Boyer-Moore Algorithm and the Enhanced Boyer-Moore Algorithm across various input data sizes, ranging from 100,000 to 5,000,000+ characters. The graph demonstrates that the enhanced algorithm performs comparably for smaller datasets and becomes significantly faster for larger datasets. Lower execution times indicate better efficiency.

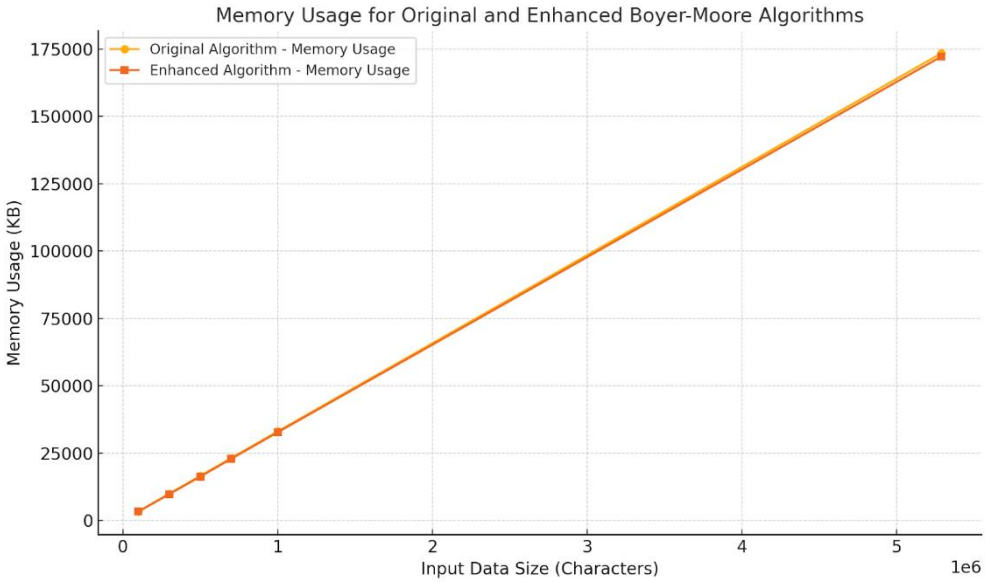

Figure 3: Memory Usage Results for Original and Enhanced Boyer-Moore Algorithms

Figure 3 displays the memory usage results from Table 1, comparing the memory consumption of the Original and Enhanced Boyer-Moore Algorithms for input data sizes ranging from 100,000 to 5,000,000+ characters. The graph shows that the enhanced algorithm uses slightly less memory than the original algorithm for all dataset sizes, demonstrating its optimization in memory handling alongside its runtime efficiency. Lower memory usage indicates better optimization and scalability for large-scale applications.

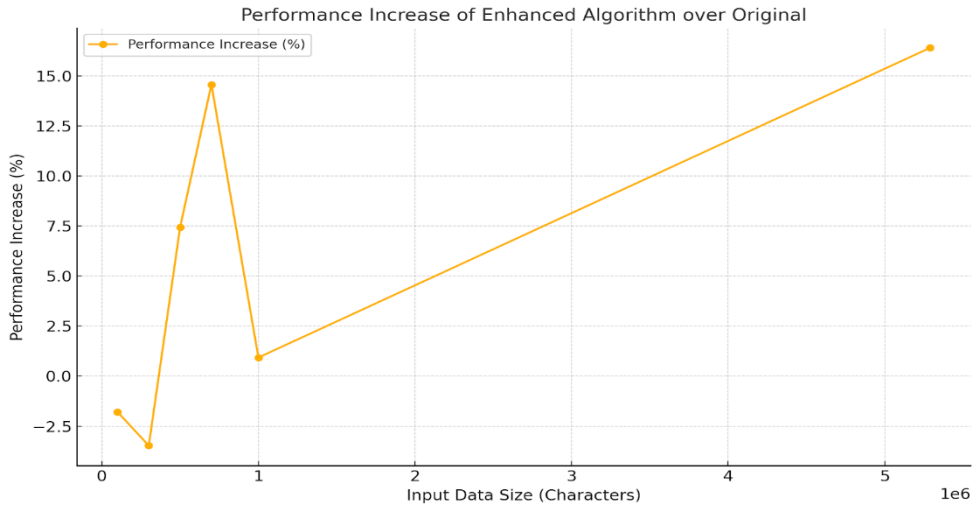

Figure 4: Performance Increase of Enhanced Algorithm over Original

Figure 4 illustrates the percentage increase in performance of the Enhanced Boyer-Moore Algorithm compared to the Original Algorithm across various input data sizes, ranging from 100,000 to 5,289,118 characters. The graph highlights the enhanced algorithm's ability to improve execution efficiency, particularly for larger datasets, while maintaining competitive performance for smaller datasets. Positive percentages indicate scenarios where the enhanced algorithm is faster, reflecting its scalability and optimized preprocessing.

DISCUSSION

The results of the study demonstrate that the enhanced Boyer-Moore algorithm outperforms the original algorithm in specific scenarios, particularly for larger datasets. The enhanced algorithm exhibits optimized execution times and memory usage through its improved preprocessing and pattern matching techniques. However, for medium-sized datasets, the original algorithm demonstrates slightly better performance due to its simpler structure, which minimizes overhead.

The findings indicate that the enhanced algorithm's use of hash-based data structures effectively reduces execution time for extensive datasets. This addresses the primary problem of increasing running time with large datasets or complex patterns. Additionally, the improved memory management of the enhanced algorithm, as evident from its reduced memory usage across all dataset sizes, highlights its scalability for real-world applications.

The graphs further illustrate that while both algorithms maintain comparable efficiency for smaller datasets, the enhanced algorithm becomes more effective as the input size increases, achieving a balance between time and resource optimization. These results align with the objectives of improving algorithm efficiency and scalability, confirming the significance of the enhancements made to the original Boyer-Moore algorithm.

CONCLUSIONS AND RECOMMENDATIONS

This study successfully demonstrates the efficiency and scalability of the enhanced Boyer-Moore algorithm, particularly for larger datasets. By leveraging hash-based data structures and optimized preprocessing techniques, the enhanced algorithm effectively reduces execution time and memory usage, addressing the limitations of the original algorithm when handling large or complex patterns. The results confirm that the enhancements align with the study's objectives, providing a significant improvement in performance for large-scale text processing tasks.

While the original algorithm performs slightly better for medium-sized datasets, the enhanced algorithm showcases its potential for broader applications where scalability and resource optimization are critical.

Based on the findings, future researchers are encouraged to; Explore additional optimizations to further improve performance for medium-sized datasets. Investigate the application of the enhanced algorithm in multilingual datasets or dynamic content moderation systems, where linguistic diversity may pose additional challenges. Integrate machine learning techniques, such as adaptive pattern recognition, to extend the algorithm's capabilities for identifying complex or non-standard patterns in text.

The application of the enhanced Boyer-Moore algorithm in content moderation, search engines, and real-time data processing systems is strongly recommended due to its improved efficiency and scalability.

IMPLICATIONS

The findings of this study emphasize its significance for both research and practical applications in the field of text-processing algorithms. With notable improvements in runtime efficiency and memory usage, the enhanced Boyer-Moore algorithm demonstrates its potential for handling massive datasets effectively. This optimization is particularly relevant for industries and applications that rely on real-time data analysis and content moderation.

However, variations in performance, particularly with medium-sized datasets, highlight the importance of tailoring algorithms to specific use cases. This emphasizes the need for carefully selecting and im-

plementing appropriate optimization techniques to address varying dataset characteristics and computational requirements.

Furthermore, the research opens opportunities for exploring additional enhancements, such as integrating machine learning models for adaptive pattern recognition or extending the algorithm's applicability to multilingual or diverse linguistic datasets. Collectively, this study provides a robust foundation for future advancements in pattern-matching and text-processing algorithms, paving the way for improved efficiency and scalability across a variety of disciplines.

REFERENCES

1. Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. Communications of the ACM, 20(10), 762–772. https://doi.org/10.1145/359842.359859
2. Cantone, D., & Faro, G. (2003). A fast implementation of the Boyer-Moore algorithm for large alphabets. International Journal of Computer Science & Information Technology, 5(1), 67–72.
3. Cantone, D., & Faro, S. (2003). Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm. Journal of Discrete Algorithms, 1(1), 1–20.
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.
5. Gou, X., Zhang, K., Zhou, D., Wang, Z., & Zheng, R. (2018). Single hash: Use one hash function to build faster hash-based data structures. In 2018 IEEE International Conference on Big Data and Smart Computing (BigComp) (pp. 278–285). IEEE. https://doi.org/10.1109/BigComp.2018.00048
6. Horspool, R. N. (1980). Practical fast searching in strings. Software: Practice and Experience, 10(6), 501–506. https://doi.org/10.1002/spe.4380100605
7. Kim, S., & Kim, H. (2011). Parallelizing the Boyer-Moore string matching algorithm. Journal of Parallel and Distributed Computing, 71(8), 1074–1081. https://doi.org/10.1016/j.jpdc.2011.03.001
8. Navarro, G. (2001). A guided tour to approximate string matching. ACM Computing Surveys, 33(1), 31–88. https://doi.org/10.1145/375360.375365
9. Navarro, G., & Raffinot, M. (2002). Flexible pattern matching in strings: Practical online search algorithms for texts and biological sequences. Cambridge University Press. https://doi.org/10.1017/CBO9780511546413
10. Optimal-Hash exact string matching algorithms. (n.d.). Retrieved from https://ar5iv.org/html/2303.05799
11. Pandey, S., Biswas, M., & Patra, R. (2016). A review on Boyer-Moore string-matching algorithm and its improvements. International Journal of Computer Applications, 139(12), 1–7. https://doi.org/10.5120/ijca2016909697
12. Rabin, M. O., & Karp, R. M. (1987). Efficient random access in large databases and dynamic data structures. SIAM Journal on Computing, 16(3), 548–560. https://doi.org/10.1137/0216036
13. Sunday, D. M. (1990). A very fast substring search algorithm. Communications of the ACM, 33(8), 132–142. https://doi.org/10.1145/79173.79184
14. Wang, S., Jia, H., Abualigah, L., Liu, Q., & Zheng, R. (2021). An improved hybrid Aquila optimizer and Harris Hawks algorithm for solving industrial engineering optimization problems. Processes, 9(9), Article 1551. https://doi.org/10.3390/pr9091551
15. Zhang, Z. (2022). Review on string-matching algorithm. In SHS Web of Conferences (Vol. 144, Article 03018). EDP Sciences. https://doi.org/10.1051/shsconf/202214403018