Comparative Analysis of String Matching Algorithms

Christof Renner (22301943)
Manuel Friedl (12306626)
Felix Hahl (22302429)

June 29, 2024

1 Introduction and Problem Description

String matching, better known as pattern matching, is an essential problem in the field of computer science and finds application in various domains such as text editing, information retrieval, bioinformatics, and data mining. At its core, string matching is about finding one, or all occurrences, of a pattern string (a substring) within a text string.

This document explores three string matching algorithms *Brute Force, Knuth-Morris-Pratt (KMP)*, and *Boyer-Moore* by performing a comparative analysis of their execution times on large text documents with varying-length patterns. The brute force method, while straightforward, often results in suboptimal performance. In contrast, *KMP* and *Boyer-Moore* algorithms incorporate preprocessing of the pattern and intelligent heuristics to expedite the search process.

The following sections will delve into the details of each algorithm's operational principles and their theoretical computational complexity. The study is conducted using algorithm implementations on synthesized text documents. A series of experiments showcase the practical execution times of these algorithms, offering insights into their relative performances and laying the groundwork for selecting appropriate string matching techniques in various applications.

This report aims to provide a clear understanding, through a structured comparative analysis, of how each algorithm performs under different conditions and the practical considerations that need to be taken into account when selecting a string matching solution.

2 Algorithm Descriptions

This section discusses the operational details and implementation specifics of three classic string matching algorithms: the *Brute-Force Algorithm*, the *Knuth-Morris-Pratt (KMP) Algorithm*, and the *Boyer-Moore Algorithm*. Each algorithm has its unique approach to the problem of pattern matching, and the implementation of these algorithms has been designed to optimize the process of searching for patterns within a larger text.

2.1 Brute-Force Algorithm

The Brute-Force algorithm, often considered the simplest approach, involves checking for the presence of the pattern at every possible position in the text until a match is found or the text is completely searched.

Implementation Detail: The implementation consists of two nested loops: the outer loop scans through the text, while the inner loop matches the pattern against the text starting at the current position. Upon a successful match, the algorithm returns the index; otherwise, it proceeds to the next position.

Operational Principle: It works by checking for the presence of a pattern in the text at every possible position. This method involves comparing the pattern to every substring of the text of the same length, moving one character at a time from the beginning to the end of the text.

Time Complexity: The brute-force algorithm has a worst-case time complexity of $O((n-m+1)m)$. In the worst case, each of the $n-m+1$ possible positions requires m character comparisons.

Potential for Improvement: The primary disadvantage of the Brute-Force approach is its inefficiency, especially when dealing with large texts and patterns. This inefficiency is attributed to its redundancy in comparisons, making it a candidate for optimization through more advanced string matching algorithms.

2.2 Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP) algorithm improves upon the brute-force approach by avoiding redundant comparisons. It achieves this through preprocessing the pattern to build a partial match table, also known as the "failure function." This table is used to skip ahead in the text, reducing the overall number of comparisons.

Implementation Detail: The KMP algorithm precomputes a longest proper prefix array (LPS) for the pattern, which is used to skip characters in the text that have already been matched with parts of the pattern. This preprocessing allows the algorithm to reduce the search space significantly.

Operational Principle: When a mismatch occurs after a sequence of matches, the LPS array is consulted to determine the longest prefix of the pattern that is a suffix up to the current match point. This information enables the algorithm to continue searching from the next possible match position without re-examining the characters.
The KMP algorithm consists of two main phases:

- Preproccessing Phase: Constructing the Partial Match Table (Failure Function)
- Searching Phase: Using the Partial Match Table to Perform the search

Time Complexity: The preprocessing phase involves a single pass through the pattern, making its time complexity ($O(m)$, where m is the length of the pattern. The searching phase involves a single pass through the text and the pattern. Thanks to the partial match table, each character in the text and pattern is compared at most once. This makes the time complexity $O(n)$, where n is the length of the text. So, in total there is a time complexity of $O(n + m)$.

Potential for Improvement: While KMP performs better than the Brute-Force algorithm, its efficiency can decrease with patterns containing many repeating elements. Optimizing the LPS array construction can yield performance gains, especially for such patterns.

2.3 Boyer-Moore Algorithm

The Boyer-Moore algorithm is one of the most effective string matching techniques due to its use of two heuristics that allow for faster pattern searching by skipping larger sections of the text.

Implementation Detail: It employs the bad character rule and the good suffix rule as its primary heuristics. The bad character rule shifts the pattern by comparing the mismatched character in the text with its rightmost occurrence in the pattern, possibly skipping several characters. The good suffix rule takes advantage of the information from a matching suffix when a mismatch occurs to move the pattern forward.

Operational Principle: Boyer-Moore starts matching from the end of the pattern and moves backwards. When a mismatch occurs, the heuristics determine the optimal shift for the pattern, allowing it to bypass irrelevant sections of the text.

Time Complexity: The time complexity of the Boyer-Moore algorithm can be analyzed based on its two main components:

- Preprocessing Phase: the worst-case time complexity is $O(m + o)$, where m is the length of the pattern and o is the size of the alphabet.
- Searching Phase: The worst-case time complexity for the searching phase is $O(mn)$, where n is the length of the text and m is the length of the

pattern.

In conclusion, while the worst-case time complexity is O(mn), the Boyer-Moore algorithm is very efficient on average and is widely used due to its good performance in typical scenarios.

Potential for Improvement: Boyer-Moore's performance is heavily reliant on the preprocessing of the pattern. By further refining these heuristics, particularly for patterns with many repeating sequences, the algorithm can achieve more significant performance improvements.

3 Description of Test Documents

The performance of string matching algorithms can vary significantly based on the nature of the text and the patterns being searched. Therefore, a careful selection and preparation of test documents are crucial for a comprehensive analysis. This study uses a set of synthetic text documents generated to mimic various real-world scenarios that influence the performance of string matching algorithms. The characteristics of these documents are as follows:

Source of Text: The text for the documents has been generated using the Faker library, a Python package that produces realistic-looking, but ultimately synthetic data. The generated text encompasses a range of linguistic constructs, including common words, rare words, punctuation, and different case usages, thus providing a varied search space for the algorithms.

Text Length: To evaluate the scalability and efficiency of the algorithms, texts of varying lengths are used. The lengths are measured in terms of the number of paragraphs, with each paragraph containing a random number of sentences. The specific lengths chosen for the study are 2034, 10073, 20121, 100510, 200969, 1004879, 2009765 and 10048711 characters, representing a broad spectrum from small to large documents.

Paragraph Length: To ensure that the results of the tests are not falsified, we have truncated the generated paragraphs after a length of 200 characters, as the Faker library generates paragraphs with a length of 200 - 250 characters.

Pattern Insertion: To ensure the presence of the pattern within the text, it is artificially inserted into the document at a random location. This approach guarantees that each search is conclusive, with the pattern being present exactly once in the text. This setup provides a controlled environment to measure the worst-case performance, where the algorithm must scan the entirety of the text to find the pattern.

Pattern Variability: The patterns searched within the texts are derived from the word **"foobabuschubdidududub"**, a non-dictionary string chosen to prevent unintended matching with naturally occurring words within the text. Patterns of varying lengths are extracted from this string, specifically

full-length, half-length, quarter-length, and one-eighth length, to observe how pattern length affects algorithm performance.

Case Sensitivity: An additional test case includes a three-word pattern with proper case sensitivity, **"Match Multiword Strings!"**, to assess the algorithms' performance in a case-sensitive search scenario, which is common in many practical applications.

Repetition and Randomness: Each test is repeated multiple times to account for the inherent variability and randomness of computational processes. The average running time from these repetitions provides a more reliable measure of performance.

These documents provide a robust framework for evaluating the algorithms' effectiveness and efficiency across a range of text sizes and pattern complexities. The detailed design of these test documents is intended to challenge the algorithms in a manner that is reflective of their use in real-world applications, where texts can be of various lengths and complexity and patterns can vary in size and case.

4 Results and Discussion

This section presents the results of the experiments conducted to compare the relative speeds of the Brute-Force, Knuth-Morris-Pratt (KMP), and Boyer-Moore pattern-matching algorithms. The results are discussed in the context of different text sizes and varying pattern lengths.

4.1 Running Time Comparisons

The graphs below illustrate the running times of the three algorithms across different text sizes and pattern lengths. Each graph compares the average, worst-case and best running time taken by the algorithms to find a pattern in the text. In order to be able to compare the graphs, we have combined one algorithm with a fixed pattern length into a graph and one with a fixed search text pattern length. Of course, the same values were used for each algorithm in order to achieve comparability.

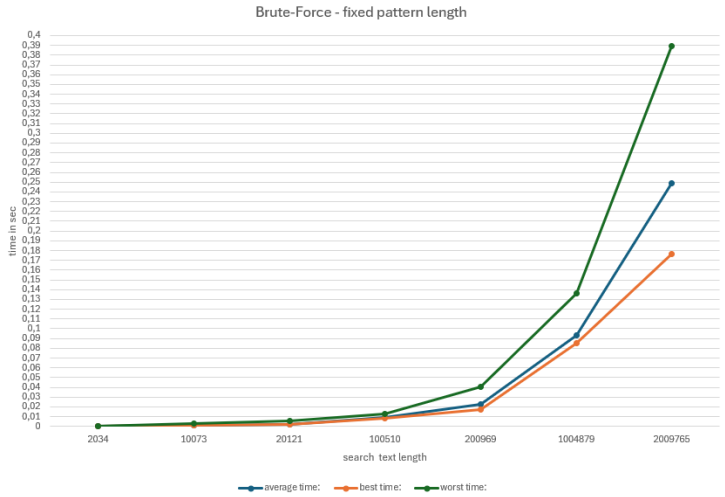

Figure 1: Running times of Brute-Force Algorithm with fixed pattern length: 42

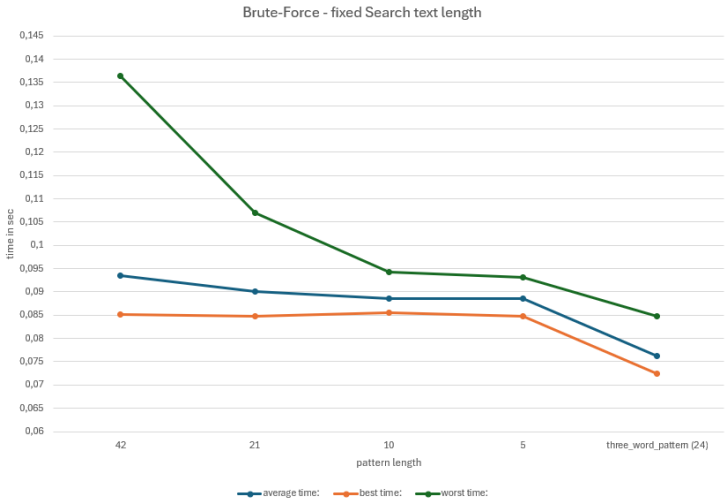

Figure 2: Running times of Brute-Force Algorithm with fixed text length: 1004879

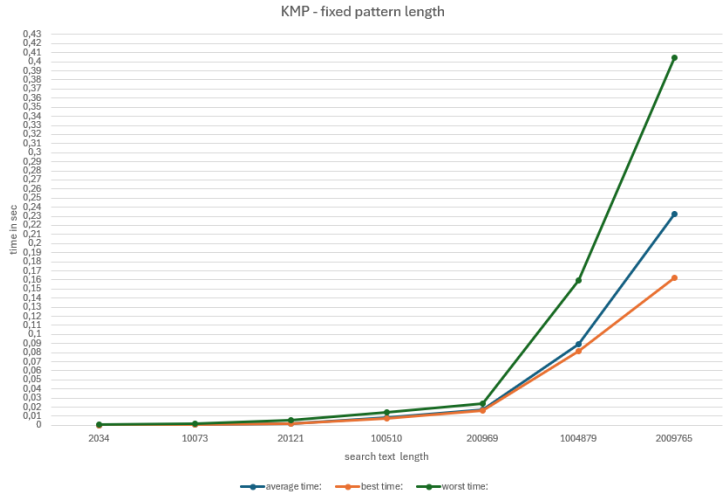

Figure 3: Running times of KMP Algorithm with fixed pattern length: 42

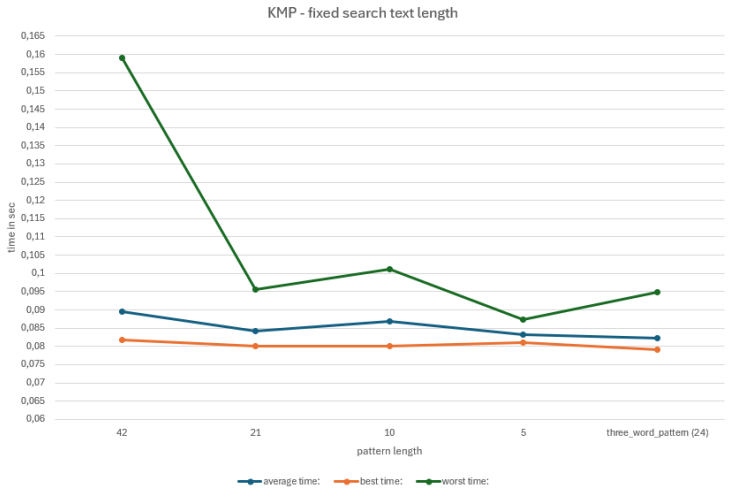

Figure 4: Running times of KMP Algorithm with fixed text length: 1004879

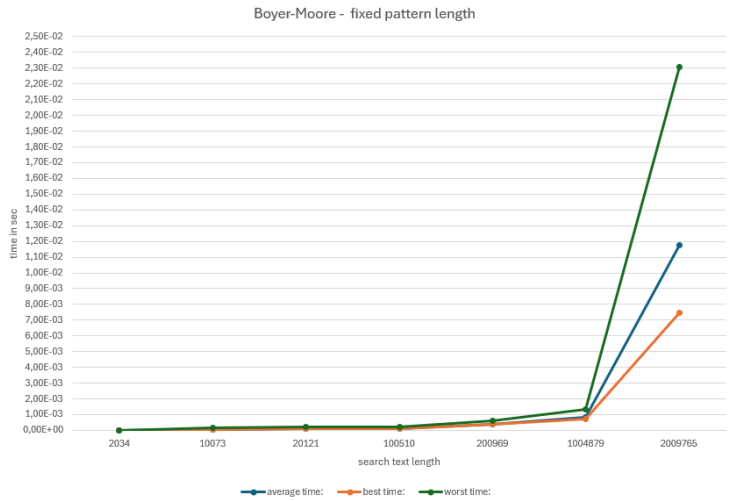

Figure 5: Running times of Boyeer-Moore Algorithm with fixed pattern length: 42

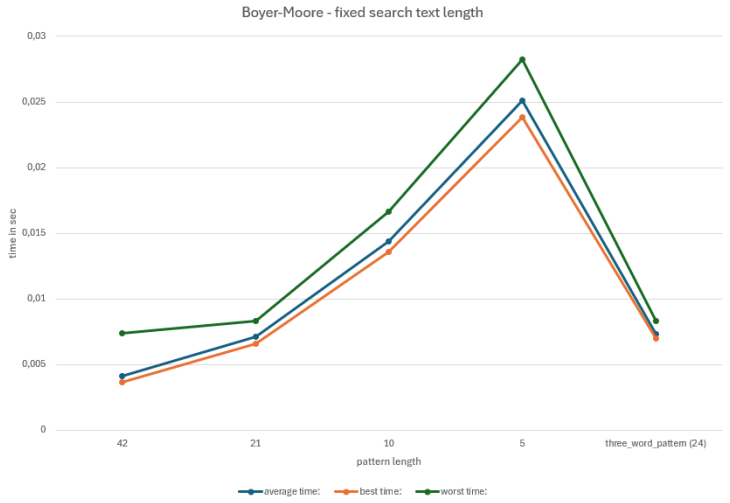

Figure 6: Running times of Boyer-Moore Algorithm with fixed text length: 1004879

4.2 Analysis of Results

Impact of Text Size: The running times for all algorithms increased with the size of the text, as expected. However, the increase was not linear, particularly for the KMP and Boyer-Moore algorithms, which showed a sub-linear growth

in running time, demonstrating their efficiency in handling larger texts.

Impact of Pattern Length: The variation in pattern length had a noticeable impact on the Brute-Force algorithm's running time, with longer patterns significantly increasing the search duration. In contrast, KMP and Boyer-Moore algorithms exhibited robustness against increasing pattern lengths, attributable to their preprocessing steps which minimized the performance penalty.

Algorithm Comparison: Among the three algorithms, Boyer-Moore showed the best performance, particularly in texts with a high density of matching candidates. Its efficient heuristic of skipping sections of the text not only reduced the number of comparisons but also minimized the impact of worst-case scenarios. KMP, while not as fast as Boyer-Moore on average, provided consistent performance regardless of the text characteristics. The Brute-Force algorithm, as predicted, performed well on short texts and patterns but its running time grew rapidly as the size of the text and pattern increased.

4.3 Theoretical vs. Practical Performance

The practical performance of the algorithms closely followed the expected theoretical behavior. The Brute-Force algorithm's practical performance, with its direct approach, aligned with its theoretical time complexity of $O(nm)$. KMP's more nuanced strategy of leveraging information about the pattern effectively reduced the expected number of comparisons, aligning with its average-case time complexity of $O(n + m)$. Boyer-Moore's performance was the most variable, often performing significantly better than its worst-case time complexity of $O(nm)$, thanks to its effective use of the bad character and good suffix heuristics.

Concluding Remarks: The experimental results reinforce the importance of algorithm selection based on the specific requirements of the application. For applications dealing with large texts and requiring frequent searches, the Boyer-Moore algorithm is the preferred choice due to its superior performance. For scenarios where consistent performance is desired over varying text and pattern sizes, KMP provides a reliable solution. The Brute-Force algorithm, while not suitable for large-scale applications, offers simplicity and might still be considered in applications where text and patterns are consistently small, and the computational overhead is negligible.

Future Directions: Further research could investigate the impact of different character sets, text encodings, and the incorporation of parallel computing techniques to enhance the performance of these algorithms. Additionally, exploring modifications to the preprocessing steps of KMP and Boyer-Moore might offer improvements in their handling of patterns with complex repetitive structures.

5 Conclusions

The comparative analysis of the Brute-Force, Knuth-Morris-Pratt (KMP), and Boyer-Moore algorithms provided valuable insights into the efficiency and applicability of these classic string matching techniques. Each algorithm exhibits distinct performance characteristics that make them suitable for different types of string matching tasks.

The Brute-Force algorithm, with its straightforward approach, demonstrates reasonable efficiency on smaller texts or where pattern occurrences are likely to be near the beginning of the text. However, its performance degrades significantly with the increase in text and pattern size due to its $O(nm)$ time complexity. This algorithm serves as a baseline for comparison and reminds us of the importance of algorithmic efficiency in processing large datasets.

The KMP algorithm offers a marked improvement over the Brute-Force approach, especially where the pattern is complex, and self-overlapping patterns are present. The preprocessing of the pattern into the LPS array allows for intelligent skipping over the text, thus avoiding redundant comparisons. KMP is particularly effective in applications where the pattern does not change frequently, allowing the preprocessing cost to be amortized over multiple searches.

Boyer-Moore stands out as the most efficient algorithm in most cases due to its use of bad character and good suffix heuristics. These heuristics often allow the algorithm to skip large sections of the text, providing a significant performance advantage, particularly in cases where the pattern is relatively small compared to the text, and patterns do not often occur within the text.

In conclusion, the selection of a string matching algorithm must be informed by the specific requirements of the application, such as text size, pattern complexity, and the frequency of searches. While the Boyer-Moore algorithm often offers the best performance, KMP remains a competitive choice due to its consistent behavior, and the Brute-Force algorithm's simplicity could be advantageous in certain constrained environments. This study underscores the importance of understanding underlying algorithmic principles to make informed decisions in software development and data processing tasks.