Determining the Efficiency Limits of KMP and Boyer-Moore Algorithms Based on Alphabet Size in Text

Sakti Bimasena - 13523053
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: sbimasena@gmail.com , 13523053@std.stei.itb.ac.id

***Abstract*—This paper experimentally examines the efficiency limits of Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) string matching algorithms, focusing on the impact of varying alphabet size. Through controlled experiments with different alphabet sizes (from 2 to 95 printable ASCII characters) and pattern lengths (5 to 50), performance was measured by execution time on fixed-length texts. Results show KMP's efficiency remains largely consistent regardless of alphabet size. However, Boyer-Moore's performance significantly improves as alphabet size increases, driven by its Bad Character Heuristic. While KMP may be competitive in very small alphabets with short patterns, Boyer-Moore consistently demonstrates superior efficiency for larger, more practical alphabets and longer patterns. These findings offer empirical guidance for algorithm selection based on data characteristics.**

Keywords—boyer-moore; knuth-morris-pratt; string matching; alphabet size; efficiency

I. Introduction

A wide variety of computational programs are based on string matching, which is the task of finding a shorter pattern in a longer text. An effective string matching algorithm is very important for many applications, including the common search function in text editors and web browsers, with specialized work in bioinformatics (like DNA sequence analysis), network intrusion detection, and data compression. Even though brute-force technique exists, it is very ineffective in handling large datasets. That is why complex algorithms like Knuth-Morris-Pratt (KMP) and Boyer-Moore have been developed.

The KMP Algorithm offers a deterministic left-to-right scanning method that is known for its worst-case linear time complexity of O(m+n) with m being the length of the pattern an n being the length of the text. This algorithm avoids unnecessary character comparisons and backtracking when a mismatch occurs by preprocessing the text into an LPS array (Longest Proper Prefix Suffix). Because of that, KMP is very strong, especially when working with patterns that show a significant amount of repetition[1].

On the other hand, Boyer-Moore is praised with its amazing performance which oftentimes reach sublinear time complexities on average. The poor character and good suffix rules are two effective heuristics used in its right-to-left comparison. When there is a mismatch, this criteria allows BM to make longer jumps (shifts) throughout the text, oftentimes avoiding long parts that do not fit the pattern. Its heuristics excel for applications that need a large alphabet and natural language texts.

While KMP is robust against worst-case scenarios, BM typically excels in average-case scenarios, especially when the alphabet is large and the pattern is long. The worst case and average case complexities of these algorithms are usually the main focus of the majority of classical analyses. However, one key factor that receives less attention in classical theory is the size of the input alphabet. In real-world applications, the alphabet may vary greatly, from small sets such as {A, C, G, T} in DNA sequences to extended ASCII or Unicode in multilingual texts. The number of unique characters in the alphabet can significantly influence the practical behavior of string matching algorithms, particularly those like BM that rely on character-based heuristics.

This paper aims to answer how alphabet size within a text and pattern effect the efficiency of KMP and BM algorithms, by doing a comparative analysis of the two algorithms by running them on a number of tests, carefully controlling the input alphabet size. With the result being a measurement of key performance metrics such as runtime and the number of character comparisons to uncover efficiency boundaries not apparent from complexity theory alone.

II. Theoretical Foundation

A. String Matching Problem

The string matching problem is a task that involves finding the first occurrence of pattern P of length m within a text T of length n, with the assumption that m is far smaller then n and both strings are composed of characters from a common alphabet. This operation is critical in many domains, such as text editing, search engines, data compression, and bioinformatics. An occurrence of P in T is defined by a shift s,

such that $0 \le s \le n - m$, and $T[s..s + m - 1] = P[0..m - 1]$. That is, for all j from 0 to m – 1, $T[s + j] = P[j]$[2].

This problems naive solution would be to check all potential alignment of P in T, which leads to a worst-case time complexity of O(nm). More effective algorithms like Knuth-Morris-Pratt and Boyer-Moore have been created to improve on this. By preprocessing the pattern and using heuristics, these algorithms avoid making unnecessary comparisons.

B. Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm (created by Knuth, Morris, and Pratt in 1977) is an effective linear-time string matching algorithm because it avoids repeated comparisons when there is a mismatch. Different to naive algorithms that maybe would re-examine characters repeatedly to find the optimal shift, KMP uses information about the pattern that has been processed beforehand to find the optimal shift.

1) Preprocessing: LPS Array

The preprocessing step of creating the Longest Proper Prefix Suffix (LPS) array, also known as "border function" or "failure function", is the core of the KMP algorithm. For a pattern P with length m, LPS[0..m-1] would store the length of the longest proper prefix from P[0..i], that is also a proper suffix of P[0..i]. A proper prefix is any prefix that is not the string itself, and same with a proper suffix. In simpler terms, the LPS array tells the algorithm how much of the matched prefix can be reused after a mismatch[1].

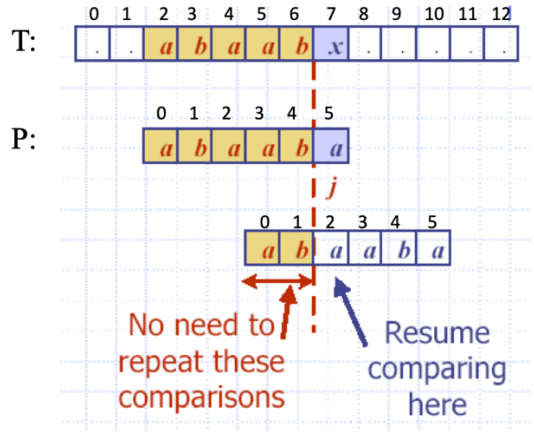

Fig 2.1. KMP Algorithm demonstration

Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf

For example, if P = "ABABCA":

- LPS[0] = 0 (for 'A', no proper prefix/suffix)
- LPS[1] = 0 (for 'AB')
- LPS[2] = 1 (for 'ABA', 'A' is a proper prefix that is also a proper suffix)
- LPS[3] = 2 (for 'ABAB', 'AB' is a proper prefix that is also a proper suffix)
- LPS[4] = 0 (for 'ABABC')
- LPS[5] = 1 (for 'ABABCA', 'A' is a proper prefix that is also a proper suffix)

The table is constructed in O(m) time by iterating through the pattern and tracking the longest border (prefix = suffix) at each position.

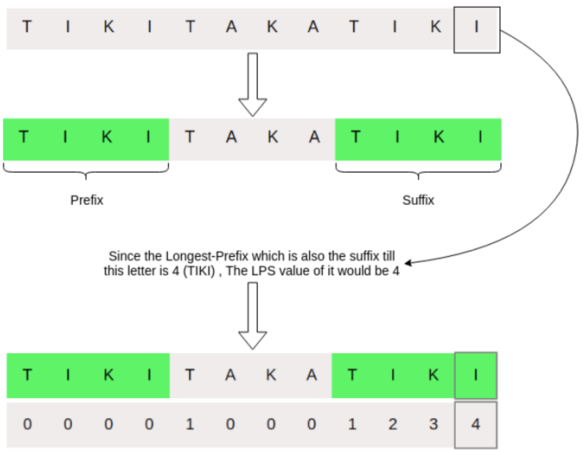

Fig 2.2. LPS Array preprocessing visualization

Source: https://ds2-iiith.vlabs.ac.in/exp/kmp-algorithm/preprocessing-of-kmp-algorithm/concept-and-strategy-preprocessing.html#:~:text=The%20preprocessing%20for%20the%20KMP,is%20also%20a%20proper%20suffix.

2) Searching

Once the LPS Array has been created, the algorithm then starts the searching process. It uses two pointers, i to index text T (from 0 to n-1), and j to index pattern P (from 0 to m-1). The algorithm then compares T[i] and P[j]. If T[i] = P[j], both pointers are increased by one, this indicates the characters at those points match. If T[i] != P[j]:

- If j = 0, No prefix of the pattern matches the current text segment. The text pointer i is simply incremented (i = i+1), and j remains 0.
- If j != 0, The pattern pointer j is shifted backward to LPS[j−1]. This means we shift the pattern to align the longest proper prefix (which is also a suffix of P[0..j−1]) with the previously matched characters in the text.

If j = m, we have found an occurrence of P.

Because each character in the text is visited at most two times, the searching phase has a worst case time complexity of O(n). If preprocessing and searching is combined, the total time complexity of KMP algorithm is O(m+n). With its linear time guarantee and deterministic character, this algorithm is very reliable. Especially when the text has many repeating patterns.

C. Boyer-Moore (BM) Algorithm

The Boyer-Moore algorithm was introduced by Boyer and Moore in 1977 and was known for being very practical. Even beating KMP in average case scenarios. Guided by two strong heuristics, the bad character rule and the good suffix rule, its

strength lies in its ability to make large jumps (shifts) by comparing the pattern with the text from right to left.

1) Bad Character Heuristic (BCH)

The BCH table, which is an array with the size being the size of the alphabet used (A), holds the last occurrence of each character in the pattern. For every character c∈A, BCH[c] is the index for the rightmost occurrence of c in P, not including P[m-1]. If c is not found in P, its value in the BCH table is typically set to -1.

When T[i] and P[j] are mismatched (where j is the index in the pattern that caused the mismatch, scanning from right to left), the pattern is shifted to the right as much as max(1, j – BCH[T[i]]). This rule guarantees that the character T[i] that was mismatched is aligned with its last occurrence in pattern P, or if T[i] is not in P, the pattern is shifted past T[i]. The assumption is that any smaller shift would cause an immediate mismatch.

For example, if P = "TATGTG":

- BCH[A] = 1, the last occurrence of A is at index 1
- BCH[G] = 5, the last occurrence of G is at index 5
- BCH[T] = 4, the last occurrence of T is at index 4

Fig 2.3. BCH demonstration

Source: https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/

2) Good Suffix Heuristic (GSH)

The GSH aims to make shifts based on the matched suffix of the pattern that led to a mismatch. If a suffix P[j+1..m−1] of the pattern matches a corresponding segment in the text, but a mismatch occurs at P[j] (T[s+j] != P[j] where s is the current shift), the good suffix rule identifies the largest shift such that:

- The matched suffix (or a part of it) in the text aligns with another occurrence of the same string within P.
- If no such occurrence exists, the longest proper prefix of P that is also a suffix of the matched segment is found and aligned.

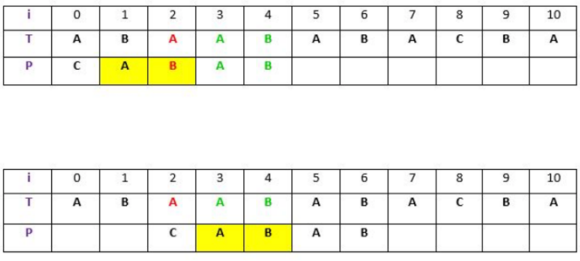

Fig 2.4. GSH demonstration

Source: https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/

The computation of the good suffix shift table is more complex than BCH, typically involving arrays to store border information, and takes O(m) time.

The preprocessing for Boyer-Moore takes O(m+A) time due to the construction of both heuristic tables.

3) Searching

While searching, the BM algorithm uses a pointer s for the current alignment of the pattern in the text (s represents the starting index of the pattern in the text). The algorithm aligns P with T[s..s + m – 1]. It then compares characters from right to left, starting from P[m - 1] and T[s + m - 1]. If a mismatch occurs at P[j] and T[s + j]:

- Calculate the shift suggested by the Bad Character Heuristic: s_{BCH} = j – BCH[T[s + j]]
- Calculate the shift suggested by the Good Suffix Heuristic, s_{GSH}, based on the matched suffix P[j + 1 .. m – 1].
- The pattern is then shifted to the right by max(s_{BCH}, s_{GSH})

The searching phase of Boyer-Moore has a worst-case time complexity of O(m+n) like KMP. But, because of its ability to make large shifts, especially in cases with large alphabets and different patterns, the average-case performance is much better and reaches sublinear time of O(n/m).

D. The Influence of Character Distribution

Even though the theoretical worst-case time complexity of both KMP and BM is O(m + n), the practical performance can be significantly different depending on the characteristics of the input text and pattern. An important factor, yet often under-examined, is alphabet size in the text.

1) Influence on KMP

The internal pattern structure that is captured by the LPS array primarily dictates the KMP algorithms shift. This means that KMP's performance relatively independent from the alphabet size of the text itself. The comparison amounts per character in the text is usually constant regardless if the text is uniform, highly repetitive, or complex. That said, a larger alphabet may reduce the number of mismatches when the pattern has less chance of aligning accidentally with the text. However, the algorithm's fundamental shift logic remains tied to the pattern's self-similarity, not the diversity of characters in the text. As such, KMP maintains consistent linear performance, even when operating over extremely small or large alphabets.

2) Influence on BM

Boyer-Moore on the other hand, is highly sensitive to alphabet size, especially through its Bad Character Heuristic[3]. A larger alphabet usually causes more frequent mismatches

with characters that rarely appear, which leads to bigger shifts and results in better average performance.

For example, when a mismatch occurs on a character that does not appear in the pattern, the bad character heuristic would allow for shifts of the full pattern length. This is much more likely in a large alphabet, where most characters are unique or rare in the pattern. As a result, BM can skip large parts of the text and have a sublinear time complexity in the average case.

But, in a smaller alphabet, like a binary string ({0,1}) or DNA sequence ({A,C,G,T}), the likelihood of finding a character that is in the pattern increases, which would decrease the size of the shift. In worst cases, like a text almost entirely made up of one character ('aaaa…'), BM would only make minimal shifts, leading to a decrease in performance, almost to brute-force levels.

Because the good suffix heuristic is based on the suffix structure of the pattern itself rather than the character values of the text, it would offer a more stable fallback. However, the overall efficiency of Boyer-Moore still strongly depends on the effectiveness of the bad character heuristic, which in turn is influenced by the size of the alphabet.

III. Methodology

This chapter explains the design of the experiment used to see how the size of the input alphabet affects the practical performance of KMP and BM algorithm. The goal of the experiment is to observe how increasing the number of unique characters in the text and pattern influences runtime and the number of comparisons for the two algorithms

A. Variables and Metrics

To precisely analyze the influence of alphabet size, we manipulated a set of independent variables while measuring specific dependent performance metrics.

- Independent Variable:
  Alphabet size (|A|). This variable is the main variable that is being investigated.
  - |A|=2 (a binary alphabet like {'0', '1'})
  - |A|=3 ({'A', 'B', 'C'})
  - |A|=4 (a DNA-like alphabet like {'A', 'C', 'G', 'T'})
  - |A|=8 ({'a'...'h'})
  - |A|=26 ({'a'...'z'})
  - |A|=62 (alphanumeric alphabet, {'a-z', 'A-Z', '0-9'})
  - |A|=95 (all printable ASCII)

  Pattern length (m). This is also a primary independent variable. The test will consist of patterns of lengths: 5,10,25, and 50 characters.
- Controlled Variables:
  - Text length (n): 10^6 characters
  - Character distribution: uniform (all characters equally likely
- Dependent Variables:
  - Execution time: measured in milliseconds

Each combination of algorithm and alphabet size is tested multiple times (5 repetitions), and the results are averaged to reduce measurement noise.

B. Data Generation Strategy

Precise control over text and pattern generation was fundamental for isolating the effect of alphabet size.

- Text and Pattern Generation: Text with 10^6 characters and patterns of the specified lengths (5,10,25,50 characters) is created for each alphabet size. Both are created by using characters that are uniformly selected from the active alphabet. For example, when testing |A| = 2, the text and pattern consists of only '0' and '1' with each having a probability of around 50% of showing. This ensures that all differences in performance is caused by the amount of characters available, not frequency.
- Reproducibility: To make sure that all text and patterns created, even though random, can be replicated if needed for verification, a random number generator is seeded appropriately for every test.

C. Algorithm Implementations

A standard implementation of both KMP and Boyer-Moore, implemented in Python, is used for this experiment. The core logic for both algorithms is available and strictly follows the well-established theoretical principles, such as the LPS array for KMP and the bad character and good suffix heuristic for BM. The main component for this implementation is the addition to count accurately every character comparison that is done in the searching phase. This allows for the collection of performance data that is precise.

IV. Results

Table 4.1 Data from experiment

Pattern length (m)	Alphabet size (|A|)	AVG Runtime	
		KMP (ms)	BM (ms)
5	2	161.58	168.66
	3	146.69	105.06
	4	137.11	106.67
	8	120.70	74.83
	26	111.54	60.80
	62	110.61	59.99
	95	118.14	65.22
10	2	171.32	122.68
	3	150.94	83.45
	4	137.30	83.71
	8	123.65	49.05
	26	112.21	33.51
	62	112.23	31.63

	95	113.75	30.78
25	2	161.37	80.26
	3	149.55	58.97
	4	138.32	64.65
	8	120.79	35.84
	26	111.59	17.77
	62	107.39	13.74
	95	106.95	12.97
50	2	157.39	59.81
	3	148.03	62.05
	4	139.17	52.62
	8	143.27	34.85
	26	111.53	12.77
	62	108.25	8.76
	95	108.21	7.59

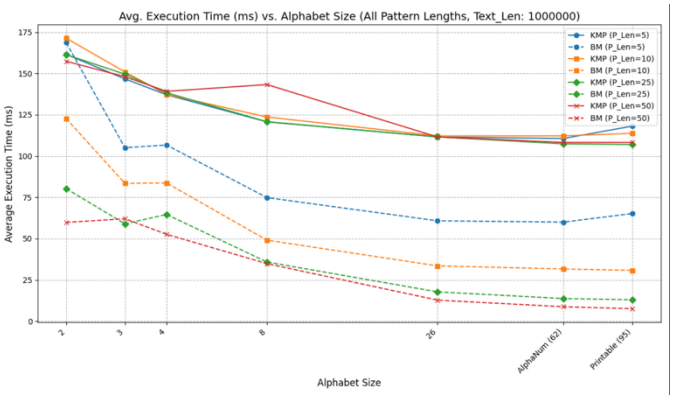

Fig 4.1 Graph of experiment data

Source: Author

The data presented in Table 4.1 and Figure 4.1 reveals a clear and distinct pattern in the average execution times for the KMP and Boyer-Moore algorithms across varying alphabet sizes and pattern lengths.

For the KMP algorithm, average execution time remains remarkably consistent across the entire range of alphabet sizes for a given pattern length. For instance, with a pattern length of 50, KMP's runtime varied from approximately 157.39 ms for a binary alphabet (|A|=2) to 108.21 ms for the printable ASCII alphabet (|A|=95). Similar stability is observed across other pattern lengths. For m=5, KMP's runtime ranged from 161.58 ms (|A|=2) to 118.14 ms (|A|=95). These minor fluctuations indicate a performance profile that is largely independent of the number of unique characters in the alphabet.

On the other hand, the Boyer-Moore algorithm consistently demonstrates a significant decrease in average execution time as the alphabet size increases, a trend clearly depicted by the steep downward-sloping dashed lines in Figure 4.1. The most substantial performance gains for Boyer-Moore are observed when transitioning from very small alphabets (|A|=2, 3, 4) to larger ones. For a pattern length of 50, Boyer-Moore's runtime drastically reduced from 59.81 ms (|A|=2) to a mere 7.59 ms (|A|=95). While significant improvements are seen across the board, the rate of performance enhancement for Boyer-Moore appears to become less pronounced for very large alphabet sizes (from |A|=62 to |A|=95), suggesting a point where additional alphabet expansion yields smaller gains in speed.

A direct comparison between the two algorithms highlights their differing sensitivities to alphabet size. For the smallest alphabet size (|A|=2), Boyer-Moore generally exhibits a shorter execution time compared to KMP across all pattern lengths tested. For instance, at pattern length 50, KMP recorded 157.39 ms while Boyer-Moore completed in 59.81 ms. As the alphabet size increases to 3 and beyond, Boyer-Moore consistently and increasingly outperforms KMP in terms of execution time. The magnitude of this performance advantage grows notably with both increasing alphabet size and increasing pattern length. For example, at a pattern length of 50 and for the printable ASCII alphabet (|A|=95), Boyer-Moore's runtime of 7.59 ms is substantially faster than KMP's 108.21 ms.

V. Analysis

A. KMP Algorithm Performance

Consistent data from the experiment shows that KMP algorithm has a relatively stable execution time in different alphabet sizes. The performance curve for KMP remains mostly flat, shown in Figure 4.1. With not so significant changes, for example in m = 50, variation from 157.39ms at |A|=2 to 108.21ms at |A|=95.

The stability shown goes inline with the fundamental theory of KMP. Its LPS array and deterministic shifting mechanism is the main factor in KMP's efficiency. The preprocessing phase and searching phase are both linear in terms of pattern and text length (O(m) and O(n)). Most importantly, both of these phases are not dependent on alphabet size (|A|), other than requiring basic character equality comparisons. While larger alphabets statistically would cause less repetition in the pattern and text that is randomly generated, which could lead to less backtracking or internal pattern comparisons in the construction of LPS array, the main time complexity is still the same. Shifts in KMP are fundamentally based on pattern properties, rather than character characteristics that are mismatched. Because of that, KMP is tough against alphabet size changes.

B. Boyer-Moore Algorithm Performance

The performance of BM is very different from KMP. The data from the experiment shows that BM is very sensitive with alphabet size changes. The trend shows a significant improvement in performance with the increase of alphabet size, like shown by the dotted lines in Figure 4.1.

The bad character heuristic for Boyer-Moore explains this phenomenon the best. BCH searches for the character that caused the mismatch (from the text) in a table that has been preprocessed from the pattern. The algorithm can make a large

shift if the mismatched character is not in the pattern or the last occurrence is far from the point of mismatch.

Characters that repeatedly occur in the pattern or text because of a very small alphabet size increases the chance that the mismatched character will be found in the pattern, and often closer to the mismatch point. Because of that, one of the biggest benefits of Boyer-Moore decreases as there are more bad and small shifts. For example, when m=50, the runtime for |A|=2 is 59.81 ms. BCH becomes very weak so the algorithm has to rely on the Good Suffix Heuristic more often. Even though its preprocess time is good (O(m)), GSH most often cannot give the same amount of large shifts as BCH.

With the increase of alphabet size, the chance of a mismatched character not appearing in the text or appearing very rarely increases drastically. This allows for BCH to make much more larger shifts, avoiding a significant portion of the text. For m=50, the runtime decreases drastically from 59,81 ms (|A=2) to 7,59 ms (|A|=95). In practice, Boyer-Moore can reach its well know sublinear performance because of the large alphabet size.

After a certain point, the benefits of increasing the size of alphabet becomes less apparent. This is shown by the flattening of BM's performance curve at large alphabet sizes (|A|=62 and |A|=95).

C. Comparison Analysis

The comparative analysis of KMP and Boyer-Moore performance shows their suitability at different alphabet sizes.

For the smallest alphabet size (Binary), KMP mostly shows an equal or better performance than BM for shorter patterns. For m=5, KMP is slightly better than BM at 161,58 ms. But even at this smallest alphabet, there is a tipping point. With an increase in pattern size (m=25 and m=50), BM becomes significantly faster. This shows that even with an inefficient BCH in a binary alphabet, the combined power of Boyer-Moore's shifts (including the GSH) for longer patterns can still lead to better performance.

For alphabet sizes of 3 or more, BM consistently outperforms KMP in runtime. The performance gap increases significantly as the alphabet size increases, showing a clear advantage in this matter. For example, BM is 14 times faster than KMP at m=50 and |A| = 95 (108.21 ms and 7.59 ms).

According to this analysis, even though KMP offers a predictable and stable linear-time performance independent from alphabet size, the practical efficiency of BM is very dependent on an alphabet size big enough to fully take advantage of its bad character heuristic. The tipping point where Boyer-Moore becomes a better choice shifts towards smaller alphabet sizes as the pattern size increases, showing that longer pattern sizes allow Boyer-Moore to get an advantage even though its bad character heuristic is limited by the alphabet size.

VI. Conclusion

This study shows how alphabet sizes affect the efficiency of string matching algorithms KMP and Boyer-Moore. The performance of KMP is largely stable and not affected by alphabet size, showing its deterministic shifting purely based on pattern repetition. However, Boyer-Moore quickly establishes a significant performance advantage in very limited alphabets (such as binary), although KMP can be competitive or slightly faster in very short patterns, but the efficiency of Boyer-Moore is highly dependent on the alphabet size, showing significant performance improvements as the alphabet grows, mainly due to the increased effectiveness of its Bad Character Heuristic. These results demonstrate that the characteristics of the input alphabet are critical when selecting the best string matching algorithm for a given application.

References

[1] Knuth, Donald Ervin, James H. Morris and Vaughan R. Pratt. "Fast Pattern Matching in Strings." *SIAM J. Comput.* 6 (1977): 323-350.

[2] SaiKrishna, Vidya, Akhtar Rasool, and Nilay Khare. "String matching and its applications in diversified fields." International Journal of Computer Science Issues (IJCSI) 9, no. 1 (2012): 219.

[3] Robert S. Boyer and J. Strother Moore. 1977. A fast string searching algorithm. Commun. ACM 20, 10 (Oct. 1977), 762–772.

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

Sakti Bimasena – 13523053