A Comparative Analysis Of String Matching Algorithms

1st Md Aslam Hossain
Department Of Computer Science And Engineering
Rajshahi University Of Engineering And Technology
Rajshahi, Bangladesh
mdaslamhossainopi@gmail.com

2nd Samiul Farhan Sumel Joy
Department Of Computer Science And Engineering
Rajshahi University Of Engineering And Technology
Rajshahi, Bangladesh
farhansumel9@gmail.com

3rd Maliha Tababsum Promi
Department Of Computer Science And Engineering
Rajshahi University Of Engineering And Technology
Rajshahi, Bangladesh
malihatababsum2000@gmail.com

4th Rukaiya Haque
Department Of Computer Science And Engineering
Rajshahi University Of Engineering And Technology
Rajshahi, Bangladesh
rukaiyahaque1229@gmail.com

***Abstract*—Text processing, bioinformatics, information retrieval, and data mining are just a few of the industries that use the fundamental computer science operation of string matching.The various pattern matching algorithms are used to locate every instance of a constrained set of patterns inside an input text or input document in order to examine the content of the documents.Three string matching algorithms—the Rabin Karp, Knuth-Morris-Pratt, and Naive String Matching algorithms—are used in this research endeavor to complete the assignment. Each algorithm is fully described, and following rigorous experimentation, its individual strengths and flaws are assessed. We evaluate them in various settings for their practicality, time complexity, and space complexity. The outcomes of this comparative analysis offer useful information for choosing the best string matching algorithm depending on the particular needs of a given application. This study can be used as a guide to help researchers, programmers, and practitioners choose the right string matching method for their projects.**

I. Introduction

String-searching algorithms, also known as string-matching algorithms, are a significant family of string algorithms used in computer science. These algorithms look for the location of one or more strings (also known as patterns) inside a larger string or text.

When the pattern and the searched text are both arrays of alphabetic (finite set) items, this is a simple example of string searching. may be a human language alphabet, such as the letters A through Z, or it may be used in bioinformatics applications that use a DNA alphabet (= A,C,G,T) or a binary alphabet (= 0 and 1). [1]

The necessity of effective string matching algorithms has increased recently due to the development of digital data and the complexity of text-based information. These algorithms serve as the foundation for several essential applications, including text search engines, plagiarism detection programs, spell checks, and many others. So choosing the best algorithm for a given set of use cases is an important challenge for both researchers and practitioners.

This research paper's main goal is to give a thorough comparative examination of several string matching methods. This analysis tries to provide information about the functionality, effectiveness, and applicability of these algorithms in various contexts.The performance and scalability of various applications can be significantly impacted by the efficacy and efficiency of the algorithm chosen for this task.

This study paper conducts a thorough investigation of numerous string matching algorithms, each of which is created with unique features and optimizations. The simple yet popular Naive String Matching algorithm, the Knuth-Morris-Pratt (KMP) algorithm renowned for its pattern preprocessing method, and the Rabin-Karp algorithm utilizing hashing for pattern matching are among the algorithms to be examined.

In conclusion, this research work aims to make a contribution to the field of string matching by providing a thorough study of numerous algorithms. The ultimate objective is to facilitate intelligent algorithm selection and optimization in many real-world circumstances. The specifics of each method, their experimental analysis, and the ramifications of our findings will all be covered in the following sections.

II. Methodology

A. Single pattern string matching algorithms

The primary objective of this research project is to match textual patterns by employing three well-known string matching algorithms to analyze the contents of the texts.The brute force approach also know as naive string matching algorithm, the rabin-karp algorithm, and the Knuth-Morris-Pratt (KMP) algorithm are all names for the string matching algorithm.We examine each of these algorithms.Also calculate the complexity of time and space.We use a graph and a table to compare one algorithm to another.

B. Naive String Matching Algorithm

A simple method of string matching is called naive string matching, commonly referred to as the brute force approach.The naive method examines every conceivable positioning of Pattern P [1.....m] in relation to Text T [1.....n]. We test shifts of 0, 1,..., n, and m for each shift s. T [s+1.....s+m] and P [1.....m] are compared.With a loop that examines the statement P [1.....m] = T [s+1.....s+m] for each of the n - m +1 possible values of s, the naive approach determines all.

NAIVE-STRING-MATCHER (T, P)

1) n ← length [T]
2) m ← length [P]
3) for s ← 0 to n -m
4) do if P [1.....m] = T [s + 1....s + m]
5) then print "Pattern occurs with shift" s [3]

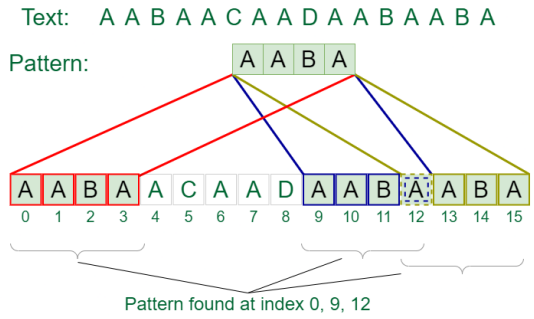


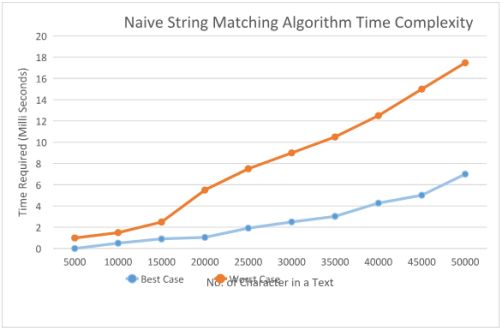
Fig. 1. Naive Algorithm [2]

- **Best Case Scenario:**
  The Naive String Matching Algorithm performs best when the pattern is absent from the text or appears at the beginning of the text. In this instance, the algorithm just needs to compare the pattern's characters with the text's first m characters (where m is the pattern's length) to determine that there are no matches. The time complexity as a result is O(m). [6]

- **Worst Case Scenario:**
  When the pattern appears at the conclusion of the text or in every feasible point inside the text, the Naive String Matching Algorithm performs poorly. Since there are many starting points for the pattern, the algorithm will need to compare each character of the pattern with each character of the text in this situation. As a result, the time complexity is O((n-m+1)m), where n is the text's length and the pattern's length is measured in m. [6]

- **Time Complexity Graph**
  The temporal complexity of a naive algorithm is calculated using a number of inputs under various circumstances.the amount of characters in a text are on the X-axis of the graph, and the time needed (in milliseconds) is on the Y-axis.

Fig. 2. Naive Algorithm time complexity graph



C. Rabin-Karp Algorithm

The Rabin-Karp algorithm is a string-searching method for quickly locating instances of a pattern in a text. It is a randomized algorithm that hashes data to achieve linear average-case time complexity. As a result, it is especially helpful for huge texts and patterns. In 1987, Michael O. Rabin and Richard M. Karp created the algorithm.The core principle of the Rabin-Karp algorithm is to compute hash values for overlapping substrings of the text and the pattern using a rolling hash function. The hash values are then compared to see if there is a possible match.Simulaion of Robin-Karp algorithm is given below (Fig-3).The Algorithm is:

RABIN-KARP-MATCHER(T,P,d,q)

1) N=T.length
2) M=P.length
3) h=d^{m-1} mod q
4) p = 0
5) t_0=0
6) for i=1 to m
7) p =(dp+P[i])mod q
8) t_0=(dt_0+t[i])mod q
9) for s = 0 to n-m
10) if p == t_s
11) if p[1..m] == T[s+1...s+m]
12) Print Pattern occcurs with shift "s
13) If s less than n-m
14) $t_s + 1$ =(d(t_s – T[s+1]h)+T[s+m+1]) mod q [1]

- **Approach:**
  The Rabin-Karp algorithm slides the pattern one by one, just like the Naive Algorithm. The Rabin Karp algorithm, in contrast to the Naive approach, compares the pattern's hash value to the hash value of the currently-selected substring of text, and only begins matching individual characters if the hash values match.

  We require a hash function that has the following property in order to efficiently calculate hash values for

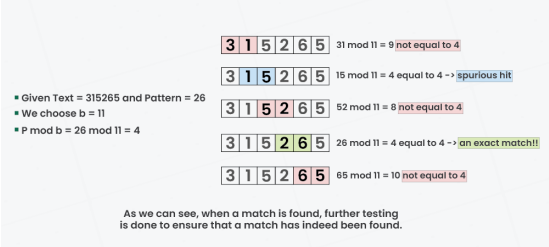


Fig. 3. Rabin Karp Algorithm [4]

all text substrings of size m:

1) Hash at the following shift must be efficiently computed from the previous shift's hash value and the following character in the text, or we can say hash(txt[s+1.. s+m]) must be efficiently computed from hash(txt[s.. s+m-1]) and txt[s+m] To put it another way, hash(txt[s+1.. s+m]) = rehash(txt[s+m], hash(txt[s.. s+m-1]))

2) Rehash needs to run as an O(1) operation.

3) The formula for rehashing is as follows: (d (hash(txt[s.. s+m-1]) = (hash(txt[s+1.. s+m]) "- txt[s]*h" + "- txt[s + m" Hash value at shift s hash(txt[s+1.. s+m]): mod q hash(txt[s.. s+m-1]): Next shift's hash value (or shift s+1)
d: The total number of letters in the alphabet.
q: a prime quantity
h: d(m-1)

- **Best And Average Case Scenario:**
The time complexity of the Rabin-Karp algorithm is typically O(N + M), where N is the length of the text and M is the length of the pattern. This is assuming a suitable hash function that distributes hash values evenly and reduces hash collisions. This is because, on average, the method computes and compares hash values using constant-time operations. The best case's time complexity is O(N+M).
- **Worst Case Scenario:**
The Rabin-Karp algorithm's worst-case time complexity is O(N * M). Hash collisions can lead to this worst-case scenario. There is a greater possibility of numerous substrings having the same hash value when the hash algorithm being utilized is not evenly distributed. In these circumstances, the algorithm could have to compare each potential match character by character, resulting in the highest possible temporal complexity.

- **Complexity Graph:**
Plotting the Rabin-Karp algorithm's complexity for the best and worst case scenarios. Where the Y axis denotes the amount of time (in milliseconds) and the X axis the number of characters in a text, respectively.

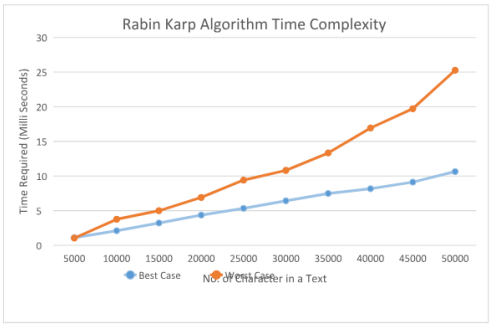


Fig. 4. Complexity Of Rabin Karp Algorithm

D. Knuth-Morris-Pratt Algorithm(KMP):

With the abettance of Knuth-Morris-Pratt, it was possible to achieve a linear-time string-matching method in contrast to previously stated polynomial time algorithms by demolishing the need for back-tracking. Using an auxiliary function π, the matching time achieved is θ(n), and pre-computation time from the pattern results in time θ(m) which can be kept in array $\pi[1 \ldots m]$. As a result, total time complexity becomes O(m + n). [5]

- **The Prefix Function π :**
A pattern's prefix function π contains information about how the pattern compares to shifts of itself, giving it the upper hand by avoiding unnecessary shifts like other algorithms. The π function has the ability to provide a tabular view which is commonly coined as LPS, or Longest Prefix Suffix table.
An example (Fig 5(a)) of a shift s is shown by the Naive String Matching algorithm when a template containing the pattern P = xyxyxzx is applied against a string T that results in q = 5 for the characters that have matched successfully. However, the 6th pattern character mismatches with the corresponding text character which practically determines q characters preceding it has matched, this information allows the determination of the corresponding text character and that certain shifts are invalid. Since the first pattern character (x) in the example in the figure would be aligned with a text character that we know does not match the first pattern character but does match the second pattern character (y), the shift s + 1 is inherently erroneous. However, the shift s' = s + 2 depicted in part (Fig-5(b)) of the figure aligns the first three pattern characters with three required text characters.
The response to the following queries is generally helpful: Given that pattern characters P[1… .q] match text characters T[s+1…s+q], what is the least shift s' greater than s such that for some k less than q, P[1… .k] = T[s'

+ 1… .s’ + k] ..... (1) where s’ + k = s + q ?

In other words, knowing that P_q] T_{s+q}, we want the longest proper prefix Pk of P_q which is also a suffix of T_{s+q}. (Since s’ + k = s + q, if we are given s and q, then finding the smallest shift s’ is tantamount to finding the longest prefix length k.) By adding the difference q - k in the lengths of these prefixes of P to the shift s to arrive at our new shift s’, so that s’ = s + (q – k). In the best case, k = 0, so that s’ = s + q, and we immediately rule out shifts s + 1, s + 2, …, s + q - 1. In any case, at the new shift s’ we don’t need to compare the first k characters of P with the corresponding characters of T, since equation (1) guarantees that they match.

We can precompute the necessary information by comparing the pattern against itself, as Figure 5(c) demonstrates. Since T[s’ + 1… .s’ + k] is part of the known portion of the text, it is a suffix of a string P_q. Therefore, we can interpret equation (1) as asking greatest k less than q such that P_k] P_q. Then, the new shift s’=s+(q-k) is the next potentially valid shift. We will find it convenient to store, for each value of q, the number of k matching characters at the new shift s’ rather than storing say, s’-s. We formalize the information that we precompute as follows. Given a pattern P [1…m], the prefix function for the pattern P is the function π : {1,2,…,m} {0,1,…,m-1} such that π[q] = max{k : k less than q and P_k] P_q } . Section can be redacted That is, π[q] is the length of the longest prefix of P that is proper suffix of P_q. Figure 5(a) gives the complete prefix function π for the pattern xyxyxzx.

KMP-MATCHER (T, P)

1) n = T.length
2) m = P.length
3) π = **COMPUTE-PREFIX-FUNCTION (P)**
4) q = 0
5) for i = 1 to n
6) while q greater than 0 and P[q+1] not equal T[i]
7) q = π[q]
8) if P[q+1] == T[i]
9) q = q + 1
10) if q == m
11) print “Pattern occurs with shift” i – m
12) q = π[q]

COMPUTE-PREFIX-FUNCTION (P)

1) m = P.length
2) let π[1 .. m] be a new array
3) π[1] = 0
4) k = 0
5) for q = 2 to m
6) while k greater than 0 and P[k+1] not equal P[q]
7) k = π[k]
8) if P[k+1] == P[q]
9) k = k + 1
10) π[q] = k
11) return π

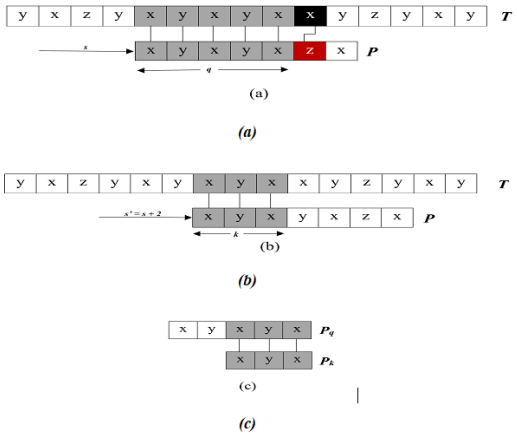


Fig. 5. 1 The prefix function π. (a) The pattern P = xyxyxzx aligns with a text T so that the first q = 5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s + 1 is invalid, but that a shift of s’ = s + 2 is consistent with everything we know about the text and therefore is potentially valid. (c) We can precompute useful information for such deductions by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P5 is P3. We represent this precomputed information in the array π, so that π [5] = 3. Given that q characters have matched successfully at shifts, the next potentially valid shift is at s’ = s+(q- π [q]) as shown in part (b). [5]

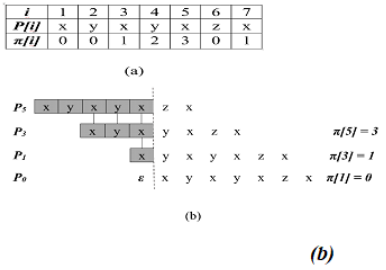


Fig. 6. An illustration for the pattern P = xyxyxzx and q = 5. (a) The π function for the given pattern. Since π [5] = 3, π [3] = 1, and π [1] = 0, by iterating π we obtain π^*[5] = {3, 1, 0}. (b) We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix ofP_5; we get matches when k = 3, 1, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after $P - 5$. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_5. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, {k: k less than 5 and P_k] P_5} = {3, 1, 0}. Lemma 32.5 claims that π^* [q] = {k: k less than q and P_k] P_q} for all q. [5]

- **Running-time analysis:**
KMP algorithm exploits the degenerating property (a

These two procedures have much in common because both match a string against the pattern P : KMP-MATCHER matches the text T against P, and COMPUTE-PREFIX-FUNCTION matches P against itself. [5]

pattern's tendency to repeat the same sub-patterns) to reduce complexity in contrast to others.
Preprocessing entails generating an auxiliary LPS array with the same size as the pattern string that corresponds to the pattern string. After asymptotic analysis was done it was found that pre-processing requires O(m) time and space complexity, and searching requires O(n + m) time complexity (regardless of the size of the alphabet).
When the pattern is absent from the text string, the time complexity is O(n), which is the best-case scenario. Average time complexity yields O(n) ; when n ¿ ¿ m Theoretically, KMP-MATCHER has a worst-case and average-case time complexity of O(n + m). Note that above, n is the length of the text string and m is the length of the pattern to be found.
The outcomes of the theoretical analysis of time complexity are reflected in experimental evidence. Using C++ programming and plotting the running time of the algorithm produced graph retains resemblance to figurative plot of its asymptotic analysis. Hence, it establishes the fact that the KMP algorithm has a linear time complexity.

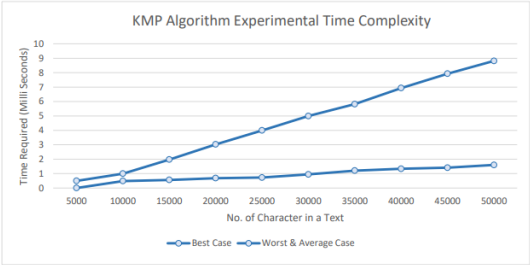


Fig. 7. Time Complexity Of KMP Algorithm

- **Patterns Used:**
  Best Case: XRRRRRRRRUET
  Worst Case: RRRRRRRRUETX

- **Space Complexity:**
  Space complexity of KMP is O(m), where m is the size of the LPS array. The experimental Space complexity found mirrors the same result which can be seen from the graph below.

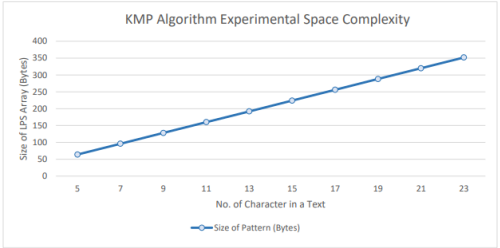


Fig. 8. Space Complexity Of KMP Algorithm

III. Comparative Analysis

- **Comparative Analysis between previous Algorithms and KMP:**

  From the theory part it can be seen that KMP has best and average case complexity O(n) in contrast to Rabin Karp having O(m + n) which is higher than KMP. Rabin-Karp in worst case has O(mn) but can reach to O(n2) if pattern and text string matches, resulting to a quadratic function which is resolved by KMP making its worst case O(m + n) i.e. linear time complexity. Naïve String Matching Algorithm has best case complexity of O(n-m) which can be resulted to O(n) however, it has worst case complexity O((n - m + 1) * m) that can approach to Quadratic time complexity in comparison to KMP matcher. So by comparing all the complexity theoretically a conclusion can be drawn that KMP is the best among all the algorithms.
  But same can't be said for the space complexity because space complexity of KMP is O(m) compared to Naïve and Rabin-Karp which is O(1) i.e. constant which can significantly large depending on the pattern size.
  Now coming to the practical analysis, the graphs drawn from each algorithm mirrors their theoretical analysis (with some errors here and there). From the graph of KMP we can see, during the best, average and worst case the graph is linear in contrast to Rabin-Karp and Naïve which is linear at best and average case but can extend to quadratic in worst case. In worst case of previous two algorithms, after 10000 character Text String input for a given pattern the graph starts to gradually show its quadratic nature which is absent in case of KMP matcher. Coming to the discussion of space complexity, KMP has space complexity graph which is proportional to the size of the pattern however Naïve and Rabin-Karp has constant space complexity mostly parallel to axis which has size of pattern.

- **Advantages of KMP over other Two string matcher algorithms:**

  1) Greatest advantage of KMP is worst case linear time complexity.
  2) Efficiency of KMP is far higher than other two, it improves efficiency, especially for lengthy texts and intricate patterns, by reducing pointless character comparisons.
  3) The KMP algorithm can be used for a range of pattern matching tasks, such as detecting the frequency of a pattern within a text or simultaneously searching for numerous patterns hence it's versatile.
  4) Mostly there is no worst case for accidental inputs.

- **Disadvantages of KMP over other Two string matcher algorithms:**

KMP Algorithm will be complicated in a scenario where the need for resources, like as CPUs and memory for processing very big datasets is large

IV. Conclusion

This research's evaluation of string matching algorithms demonstrates the effectiveness of the Naive string matching method, the Knuth-Morris-Pratt (KMP) algorithm, and the Rabin Karp algorithm while also providing guidance on how to select the most effective approach. The decision is made based on the user's particular use case and performance needs. When no pre-processing is necessary, the naive technique is especially favorable because its running time is equal to the matching time, whereas KMP greatly reduces search durations compared to the Brute Force method The Rabin-Karp technique is extremely effective for big text corpora because it can conduct string matching in linear time complexity by utilizing a rolling hash function.Algorithms for accurate and approximate string matching are both essential for solving many computing problems.The choice of a string matching technique depends on specific requirements, the characteristics of the data at hand, and the trade-offs one is willing to accept in terms of time complexity, memory usage, and predictability. Our research analysis led us to conclude that the Knuth-Morris-Pratt (KMP) approach should be used. In cases where efficiency and predictability are of the utmost concern, the Knuth-Morris-Pratt (KMP) method emerges as a strong choice for the exact matching of strings.Time efficiency in many areas of computer science might be greatly enhanced by fostering creativity and innovation in the field of string matching algorithms.

Acknowledgment

We feel privileged to express our deepest sense of gratitude. To our guides **Prof. Dr. MD Ali Hossain** (Rajshahi University Of Engineering And Technology) and **A.F.M Minhazur Rahman** (Rajshahi University Of Engineering And Technology). Their prompt and kind help led to completion of work.

References

[1] Sri, Mukku Bhagya, Rachita Bhavsar, and Preeti Narooka. "String Matching Algorithms." International Journal Of Engineering And Computer Science 7.03 (2018): 23769–23772. Web.

[2] "Naive Algorithm for Pattern Searching." GeeksforGeeks, GeeksforGeeks, 24 Feb. 2023, www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/.

[3] "DAA Naive String Matching Algorithm", javatpoint,No Date Given, https://www.javatpoint.com/daa-naive-string-matching-algorithm

[4] "Rabin-Karp Algorithm for Pattern Searching", GeeksforGeeks,September 06,2023,https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

[5] Cormen, Thomas H., et al. Introduction to Algorithms. Mit Press, 2009.

[6] "The Naive String Matching Algorithm", Medium,September 08, 2019,https://medium.com/krupa110/the-naive-string-matching-algorithm-be7992ebbd1d.