# String Matching Algorithms

Samar Ahmed, Abdallah Ibrahim, and Bushra Alqhatani

Computer Science and Engineering Department, The American University in Cairo

CSCE 1101

Dr. Howaida Ismaeel

4/29/2023

**Table of Contents:-**

# String Matching Algorithms

## 1. 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. Even though the practical applications fall under the purview of computer science research, string matching is a classic and ongoing problem. Within a well-built string or "Text" in this domain, one or more strings called " Patterns" must be searched. Algorithms or strategies for string matching play a crucial part in a variety of applications or challenges in the real world. Spell checkers, spam filters, intrusion detection systems, search engines, plagiarism detectors, bioinformatics, digital forensics, information retrieval systems, and others are only a few of its crucial applications. The string-searching algorithms would align the pattern with the beginning of the text and proceed until a match was found or the text's end was reached. The most typical method will operate as follows: the text will be scanned and its size set to m. By comparing the characters in the text to the characters in the pattern, it will be possible to determine if the pattern appears in each segment of the text that depends on m. There are numerous string-matching methods; however, the issues are numerous and challenging to investigate and solve (Al-Khamaiseh & ALShagarin, 2014). If an effective algorithm is applied, exact string pattern matching's time performance can be significantly enhanced. It is worthwhile to adopt this effective method considering the increasing volume of text processed in the electronic patient record. This study will compare the speed of various algorithms and show how some of them might be improved. It will also present several published publications that suggested surveys of string-matching techniques. allowing the string-matching field to be organized such that the issues and solutions can be seen clearly.

## 2. Motivation

There has been working done in recent years to find quick, effective string-matching algorithms. Strategies or algorithms for string matching play a crucial part in many applications or challenges found in the real world. Therefore, it is crucial to work tirelessly all the time to improve these algorithms. The goal of this research article is to examine several string-matching algorithms in order to identify their differences as well as the benefits and drawbacks of each approach. The fact that the research will be beneficial and useful regardless of the findings is a major driving force behind it.

## 3. Problem Definition

Finding patterns in a vast body of text today relies heavily on the idea of string-matching algorithms. These algorithms have a wide range of uses, including applications in bioinformatics, intrusion detection systems, and plagiarism detection (Al-Khamaiseh & ALShagarin, 2014). A comparison of the string-matching algorithms' running times has been done utilizing a variety of published publications. There are three inputs: a text string, a pattern string, and an integer k. Finding every instance of the pattern in the text with at most k differences was the experiment's task (JOKINEN et al., 1996). Four distinct algorithms will be taken into consideration in this research. It would also come to light that there can be significant speed differences between the methods.

## 4. Literature Survey of Algorithms

### 4.1 Introductory to Algorithms

Al-Khamasieh and al-Shagarin (2014) conducted the first survey that will be used; their primary goal was to identify key characteristics of the string-matching mechanisms; they made no recommendations regarding particular string-matching algorithms. They used two basic categories of search models to design their survey. Depending on the language, the first thing is the word. Secondly, will any of the sequences begin in an index point?

By comparing the various character comparison techniques in the main text and the pattern, the first algorithms that will be used are the Knuth-Morris (KMP) algorithms. It is regarded as a simple string match algorithm and the pattern moves from the left to the right. The KMP, may detect mismatches and adjust the pattern to avoid repeated comparisons. As a result, it offers the benefit of never decrementing the text pointer (Al-Khamaiseh & ALShagarin, 2014). The third algorithm is the Boyer-Moore (BM)  Algorithm. This algorithm, on the other hand, matches the pattern from right to left and keeps two heuristics in place in the event of a mismatch. The first is referred to as the bad character heuristic and involves inserting the mismatched character into the search pattern. The second, known as the good suffix heuristic, shifts the search pattern to the next occurrences when a mismatch occurs in the middle of the search string.

Other than the classical methods there are also deterministic finite automaton (DFA) methods. A data structure called DFA allows for quick string matching by storing all of a string's suffixes. Any general automation is changed into deterministic automation, which reduces the amount of memory needed. The fourth algorithm, The Automaton Matcher Algorithm is a well-known algorithm. It scans from the left-to-right of the text character by character and automation transitions. Preprocessing is necessary for this approach (Al-Khamaiseh & ALShagarin, 2014).

Lastly, the Hashing Method provides a quick and easy way to avoid using quadratic numbers of characters. The Sixth and final algorithm that will be talked about in this paper is the Karp-Rabin (KR) Algorithm is a good illustration of this since it computes the hashing function for each m-character in the text and determines whether it matches the hashing function of the pattern. A couple of these algorithms can be improved to make them more effective and quick after researching these techniques and other algorithms (Al-Khamaiseh & ALShagarin, 2014).

In a following survey, Jokinen et al (1996) attempted to find the endpoints of the occurrences from the substring. Dynamic programming can solve this problem in time O(mn); this was a result of Galil and Giancarlo's research. Later, Landau and Vishkin provided other algorithms that preprocess the pattern in time O(m^2). In order to contrast the quicker algorithms The k-difference problem was used by Jokinen et al. (2020) to compare the algorithms. The idea of edit distance is used to calculate the distance between strings A and B in this experiment. The k difference issue asks you to identify all j such that the edit distance between p and the substring is at most k when both the pattern and the text are given with the integer k (JOKINEN et al., 1996). In the section after this, examples of these six algorithms will be provided, following a deeper analysis and comparison between their speed and efficiency.

**4.2 Algorthims used in solving the problem:**

**A. <u>Rabin Karp</u>**

```
#define d 256

void search(char pat[], char txt[], int q)

{
```

```
int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0; // hash value for pattern

int t = 0; // hash value for txt

int h = 1;

// The value of h would be "pow(d, M-1)%q"

for (i = 0; i < M - 1; i++)

   h = (h * d) % q;

for (i = 0; i < M; i++) {

   p = (d * p + pat[i]) % q;

   t = (d * t + txt[i]) % q;

}

      for (i = 0; i <= N - M; i++) {

   if (p == t) {

      for (j = 0; j < M; j++) {

         if (txt[i + j] != pat[j]) {
```

```
                    break;

                }

            }

  if (j == M)

                cout << "Pattern found at index " << i

                     << endl;

        }

        if (i < N - M) {

            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            if (t < 0)

                t = (t + q);

        }

    }

}
```

**(GeekforGeeks, 2011b)**

This program implements the Rabin-Karp algorithm for string matching. The algorithm uses hashing to efficiently compare a pattern string with all possible substrings of a larger text string. The function search takes three arguments: pat (the pattern string to search for), txt (the larger text string to search within), and q (a prime number used in the hashing function). The program first calculates the hash values of the pattern string and the first window of the text string using the specified prime number q. It then slides the pattern window over the text string one position at a time, recalculating the hash value of the current window using a rolling hash technique. If the hash values of the current window and the pattern match, the program checks character by character to confirm the match. If a match is found, the program outputs the index of the match in the text string. This algorithm has a time complexity of O(n+m) in the worst case, where n is the length of the text string and m is the length of the pattern string (Al-Khamaiseh & ALShagarin, 2014).

## B. Knuth-Morris-Pratt (KMP)

```
void KMPSearch(char* pat, char* txt)

{

        int M = strlen(pat);

        int N = strlen(txt);

        int lps[M];

   computeLPSArray(pat, M, lps);

        int i = 0; // index for txt[]

        int j = 0; // index for pat[]
```

```
while ((N - i) >= (M - j)) {

if (pat[j] == txt[i]) {

j++;

i++;

}

if (j == M) {

printf("Found pattern at index %d ", i - j);

j = lps[j - 1];

}

// mismatch after j matches

else if (i < N && pat[j] != txt[i]) {

if (j != 0)

        j = lps[j - 1];

else

        i = i + 1;

}

}
```

```
}

void computeLPSArray(char* pat, int M, int* lps)

{

        int len = 0;

        lps[0] = 0; // lps[0] is always 0

        int i = 1;

        while (i < M) {

        if (pat[i] == pat[len]) {

        len++;

        lps[i] = len;

             i++;

        }

        else // (pat[i] != pat[len])

        {

        if (len != 0) {

                len = lps[len - 1];

        }
        else // if (len == 0)

        {

                lps[i] = 0;

                i++;

        }

        }

        }

}
```

**(GeekforGeeks, 2011a)**

The code implements the Knuth-Morris-Pratt algorithm for pattern searching in a given text. The function KMPSearch() takes two arguments, the pattern to search (pat) and the text in which to search (txt). It first calls the function computeLPSArray() to compute the Longest Prefix Suffix (LPS) array for the pattern. It then iterates over the text and pattern using two indices i and j respectively, and compares the characters at the corresponding positions. If they match, it increments both indices. If j reaches the end of the pattern, a match is found and the index of the match is printed. If there is a mismatch, it resets j to the last matching character and increments i. The computeLPSArray() function takes the pattern (pat), its length (M), and an array lps to store the LPS values. It uses two indices i and len, where len is the length of the LPS for the previous index. It iterates over the pattern and computes the LPS values for each index using a dynamic programming approach (Kumar et al., 2011).

**C. Boyer Moore**

```
# define NO_OF_CHARS 256

void badCharHeuristic( string str, int size,

                        int badchar[NO_OF_CHARS])

{

    int i;

    for (i = 0; i < NO_OF_CHARS; i++)

        badchar[i] = -1;

    for (i = 0; i < size; i++)

        badchar[(int) str[i]] = i;

}

void search( string txt, string pat)

{

    int m = pat.size();

    int n = txt.size();

    int badchar[NO_OF_CHARS];

    badCharHeuristic(pat, m, badchar);
```

```
   int s = 0;

   while(s <= (n - m))

   {

      int j = m - 1;

      while(j >= 0 && pat[j] == txt[s + j])

         j--;

    if (j < 0)

      {

         cout << "pattern occurs at shift = " <<  s << endl;

         s += (s + m < n)? m-badchar[txt[s + m]] : 1;

      }

      Else{

 s += max(1, j - badchar[txt[s + j]]);

   }

}
```

(GeekforGeeks, 2012b)

The code implements the Boyer-Moore string-searching algorithm using the bad character heuristic. It takes two input strings: the text to search for a pattern in, and the pattern to search for in the text. The badCharHeuristic() function preprocesses the pattern by creating an array that stores the rightmost occurrence of each character in the pattern. The search() function iterates over the text and shifts the pattern according to the bad character heuristic until a match is found or all shifts have been exhausted. If a match is found, it prints the index of the shift where the pattern occurs in the text. The time complexity of the algorithm is O(mn), where m is the length of the pattern and n is the length of the text. However, in practice, it often runs much faster than this worst-case bound (CMU CS, 2016).

## D. Finite Automaton Algorithm

```
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)

{

    if (state < M && x == pat[state])

    return state+1;

    int ns, i;

    for (ns = state; ns > 0; ns--)

    {

    if (pat[ns-1] == x)
```

```
        {
        for (i = 0; i < ns-1; i++)
                if (pat[i] != pat[state-ns+1+i])
              break;
        if (i == ns-1)
                return ns;
        }
        }
        return 0;
}
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
  int state, x;
  for (state = 0; state <= M; ++state)
      for (x = 0; x < NO_OF_CHARS; ++x)
        TF[state][x] = getNextState(pat, M, state, x);
}
```

```
/* Prints all occurrences of pat in txt */

void search(char *pat, char *txt)

{

   int M = strlen(pat);

   int N = strlen(txt);

   int TF[M+1][NO_OF_CHARS];

   computeTF(pat, M, TF);

   // Process txt over FA.

   int i, state=0;

   for (i = 0; i < N; i++)

   {

      state = TF[state][txt[i]];

      if (state == M)

         printf ("\n Pattern found at index %d",

                                   i-M+1);

   }

}
```

**(GeekforGeeks, 2012a)**

This C program implements the Finite Automata Pattern Searching algorithm to find occurrences of a given pattern in a given text. The program first builds a Transition Function (TF) table for the pattern, which represents a Finite Automaton. Then, it processes the text over the Finite Automaton and updates the state based on the current character of the text. If the state becomes the final state, it means that the pattern has been found, and the program prints the index of the starting position of the pattern in the text. The program also includes a function to compute the next state based on the current state and input character (Al-Khamaiseh & ALShagarin, 2014).

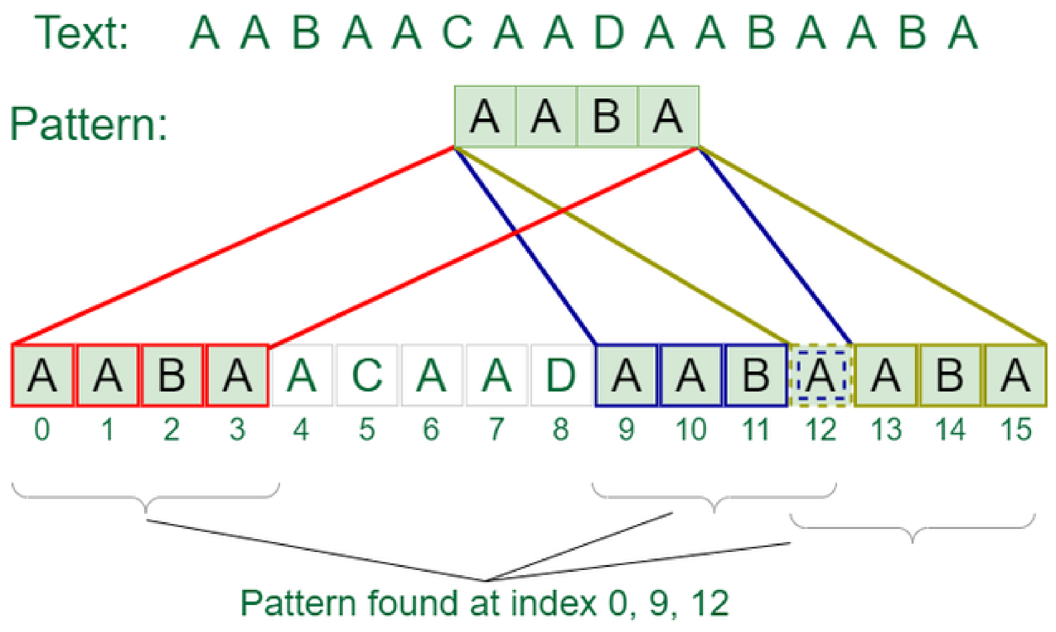
## 5. Examples:

### 5.1 Rabin Karp

If an array of text is given with an unknown size, txt[0….n-1] and an array of pattern patt[0…m-1], Rabin Kaprt can print all the occurrences of pat[] in txt[] assuming that n>m.
For instance, if the input for txt is txt[] = "This is a test for the text ", and the pattern was pat[] = "test" then the output would be that the pattern is found at index 10 (GeekforGeeks, 2011b).

## 5.2 Knuth-Morris-Pratt (KMP)

A similar example that has been given is also for the number of occurrences of a certain pattern in a text.

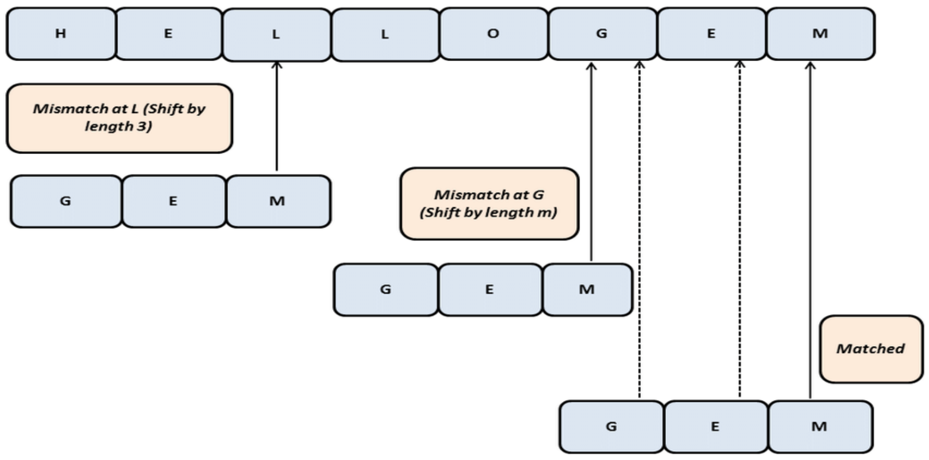

Text: A A B A A C A A D A A B A A B A
Pattern:
A A B A
A A B A A C A A D A A B A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Pattern found at index 0, 9, 12

(GeekforGeeks, 2011a)

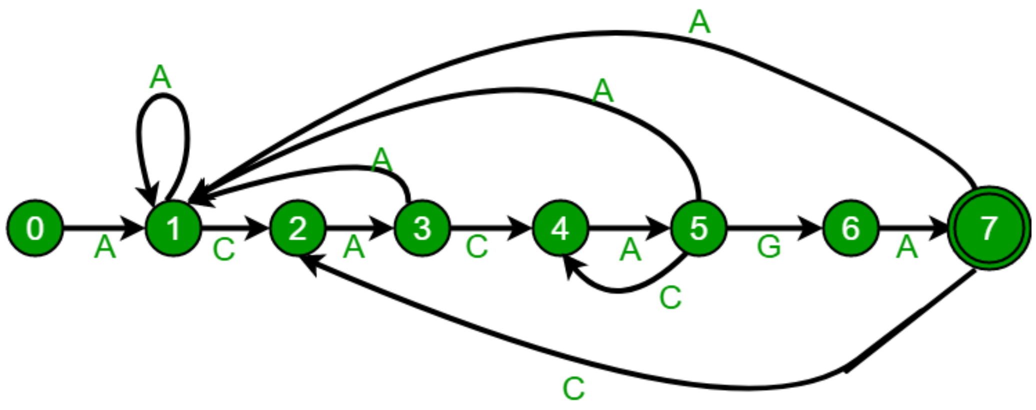
If a text has been inputted similar to the image above txt[] = "AABAACAADAABAABA" and the pattern was pat[]=" AABA" then the algorithm would search for the indexes that match this pattern and output the starting index of each pattern in the text. Therefore the index 0, 9, and 12 will be outputted.

### 5.3 Boyer Moore

The Boyer Moore is a combination of the Bad Character Heuristic and the Good Suffic Heuristic, it slides the pattern over the text one by one, and it does preprocessing so that the pattern can be shifted therefore, an example of this is from ResearchGate (2019):


H
E
L
L
O
G
E
M
Mismatch at L (Shift by length 3)
G
E
M
Mismatch at G (Shift by length m)
G
E
M
Matched
G
E
M

### 5.4 Finite Automaton Algorithm:



This is an efficient method of searching for a pattern since it will examine each text character once. The main construct of the Finite Automaton is to get the next state from the current state for every character. The length of the longest prefix of the pattern is also the suffix of pat[0…k-1]x. For instance, the image shown above shows that the length of the longest prefix is 4 "ACAC". Therefore the next state from state 5 is 4 for the character "C". Another example could be as follows:

1. Initialization
2. Looking for "m"
3. Recognized "m", looking for "a"
4. Recognized "ma", looking for "i"
5. Recognized "mai", looking for "n"
6. Recognized "main"

(Rochester University, 2020).

## 6. Analysis and Critiques of Algorithms

### 6.1 Rabin Karp:

The Rabin-Karp algorithm is a string-searching algorithm that finds the occurrence of a pattern in a given text in linear time. It was developed by Michael O. Rabin and Richard M. Karp in 1987 and is based on the concept of hashing (Addanki, 2011).

The basic idea of the Rabin-Karp algorithm is to calculate the hash value of the pattern and each substring of the text of the same length as the pattern. If the hash values of the pattern and a substring match, then the algorithm compares the characters of the pattern and the substring to confirm the match. If a match is found, the algorithm returns the starting position of the substring (Al-Khamaiseh & ALShagarin, 2014).

The Rabin-Karp algorithm has a time complexity of O(n+m), where n is the length of the text and m is the length of the pattern. The algorithm performs m hash calculations and m character comparisons for each substring of the text, which gives a total of n-m+1 substrings. Therefore, the total number of operations is (n-m+1) * (m + hash calculation time), which is linear in the length of the text (Kumar et al., 2011).

One advantage of the Rabin-Karp algorithm is that it can handle multiple patterns in a single pass. To do this, the algorithm simply calculates the hash values of all the patterns and checks each substring of the text against all the pattern hash values.

However, the Rabin-Karp algorithm has some limitations. One limitation is that the algorithm may generate false positives due to hash collisions, where two different strings have the same hash value. To minimize the risk of collisions, the algorithm uses a hash function that distributes the hash values evenly. Another limitation is that the algorithm may require a large amount of memory to store the hash values of all substrings.

In conclusion, the Rabin-Karp algorithm is a simple and efficient string-searching algorithm that can handle multiple patterns in a single pass. It has a linear time complexity and can be used in a wide range of applications, such as text editors, compilers, and database systems. However, it has some limitations, such as the risk of false positives and the requirement of a large amount of memory.

## 6.2 Knuth-Morris-Pratt (KMP):

The KMP algorithm is a string-matching algorithm that works by searching for occurrences of a pattern within a larger string, using a technique known as the "prefix function" or "failure function" (CMU CS, 2016). The algorithm is named after its inventors, Donald Knuth, Vaughan Pratt, and James Morris.

The main idea behind the KMP algorithm is to preprocess the pattern to be searched (which we will refer to as P) and compute an auxiliary array, which we will call F, that provides information about the longest proper prefix of P that is also a suffix of a prefix of P. This array F is computed using a recursive procedure that iteratively checks whether the prefix of P ending at each position has a proper suffix that is also a prefix of P. The time complexity of this preprocessing step is O(m), where m is the length of P (Kumar et al., 2011).

Once the array F has been computed, we can use it to search for occurrences of P within a larger string, which we will refer to as S. We start by aligning the first character of P with the first character of S, and then we compare each character of P to the corresponding character of S, moving forward in both strings as long as the characters match. However, if at any point we find a mismatch between a character of P and the corresponding character of S, we use the information in the array F to determine the next position in P to compare against the current position in S. Specifically, we move to the position in P corresponding to the value of F at the current position in P, and continue the comparison from there (Kumar et al., 2011). This allows us to avoid re-examining the previously matched characters in P that we know do not contribute to a match.

The time complexity of the string-matching step is O(n), where n is the length of S because we only examine each character of S at most once. Thus, the overall time complexity of the KMP algorithm is O(m + n), which is linear in the size of the input (CMU CS, 2016). This makes it an efficient algorithm for searching for patterns within large strings.

In terms of space complexity, the KMP algorithm requires O(m) additional memory to store the array F, which is the same as the preprocessing time complexity. The space complexity of the string-matching step is O(1) because we only need to keep track of a few pointers to the current positions in S and P (Al-Khamaiseh & ALShagarin, 2014).

Overall, the KMP algorithm is a powerful and efficient string-matching algorithm that is widely used in practice. It is linear time complexity and low space requirements make it a good choice for searching for patterns within large data sets.

## 6.3 Boyer-Moore Algorithm:

The Boyer-Moore algorithm is a string-matching algorithm that is used to find occurrences of a pattern in a given text. The algorithm was developed by Robert Boyer and J Strother Moore in 1977 and is known for its practice efficiency (UC Davis, 2011).

The algorithm consists of two main phases, the preprocessing phase, and the matching phase. In the preprocessing phase, the algorithm constructs two arrays, the bad character rule, and the good suffix rule.

The bad character rule is used to skip over characters in the text that do not match the corresponding character in the pattern. The rule is constructed by calculating the last occurrence of each character in the pattern (UC Davis, 2011). If a character in the text does not match the corresponding character in the pattern, the algorithm will shift the pattern to align the last occurrence of that character with the mismatched character in the text.

The good suffix rule is used to handle cases where the pattern contains a suffix that matches a substring of the pattern. The rule is constructed by calculating the length of the longest suffix of the pattern that matches with a substring of the pattern. If a suffix of the pattern matches with a substring of the pattern, the algorithm will shift the pattern to align the matching suffix with the corresponding substring in the text (CMU CS, 2016).

In the matching phase, the algorithm slides the pattern over the text, starting from the end of the pattern and moving toward the beginning (UC Davis, 2011). If a mismatch occurs, the algorithm uses the bad character rule and the good suffix rule to determine the amount to shift the pattern.

The time complexity of the Boyer-Moore algorithm is O(m + n), where m is the length of the pattern and n is the length of the text. The space complexity of the algorithm is O(m), which is the space required to store the bad character rule and the good suffix rule (CMU CS, 2016).

In practice, the Boyer-Moore algorithm is known for its efficiency and is commonly used in applications such as text editors, search engines, and virus scanners. However, the algorithm may not perform well on patterns with many repeating characters, as this can cause many shifts to be made in the matching phase.

### 6.4 Finite Automaton Algorithm

The main advantage of this tool is that it can examine every character in the text exactly once and report all the valid shift O (n) times. Its main aim is to accept or reject an input depending on whether the pattern defined by the Finite Automaton occurs in the input. The main input sequence is as follows. First, the Finite Automaton will begin at the start state, if the next input character matches the label on a transition it will go to the new state. Finite Automaton will continue to make transitions, if there is no move possible then it will stop and if in accept state it will accept (Rochester University, 2020).

The computeTF() is the function that constructs the Finite automaton. Its time complexity is O(m^3*NO_OF_CHARS) where the m is the length of the pattern and the NO_OF_CHARS is the size of the alphabet. The function tries all the possible prefixes starting with the longest possible that be a suffix. The auxiliary space is O(m) (GeekforGeeks, 2012a). Furthermore, there are limitations to this algorithm which are that the algorithm can not recognize a set of binary strings and the set of strings over '(' and ')' (Rochester University, 2020).

Overall, this algorithm is known to be one of the most useful tools that can be used for string searching. Since pattern searching is an important problem in computer science, this algorithm is able to make that process easier and more efficient.

## 7. Conclusion:

There are many mechanisms in the string-matching field. This paper makes an effort to eliminate ambiguity, organize the knowledge in this field, explain how various methods are likely to cooperate and pinpoint any remaining flaws that may need further investigation. In order to highlight the distinctions between temporal complexity and auxiliary space, this study included four different methods along with thorough explanations of each. To further highlight the differences between them and any loopholes and unacknowledged advantages and disadvantages of each algorithm, each has its own case.

## References

Addanki, R. (2011). *Karp-Rabin*. https://cs.indstate.edu/~raddanki/abstract.pdf

Al-Khamaiseh, K., & ALShagarin, S. (2014). A Survey of String Matching Algorithms. *Journal of Engineering Research and Applications*, *4*(7), pp.144-156. https://www.researchgate.net/profile/Shadi-Alshagarin/publication/279442892_A_Survey_of_String_Matching_Algorithms/links/5592883108ae49b1fd6f0a67/A-Survey-of-String-Matching-Algorithms.pdf

CMU CS. (2016). *Lecture #9: String Matching*. https://www.cs.cmu.edu/~15451-f17/lectures/lec09-kmp.pdf

Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., & Rytter, W. (1994). Speeding up two string-matching algorithms. *Algorithmica*, *12*(4-5), 247–267. https://doi.org/10.1007/bf01185427

Galil, Z., & Park, K. (1990). An Improved Algorithm For Approximate String Matching. *SIAM Journal on Computing*, *19*(6), 989–999. https://doi.org/10.1137/0219067

GeekforGeeks. (2011a, April 3). *KMP Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

GeekforGeeks. (2011b, May 18). *Rabin-Karp Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

GeekforGeeks. (2012a, April 24). *Finite Automata algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/

GeekforGeeks. (2012b, May 26). *Boyer Moore Algorithm for Pattern Searching*. GeeksforGeeks. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

JOKINEN, P., TARHIO, J., & UKKONEN, E. (1996). A Comparison of Approximate String Matching Algorithms. *Software: Practice and Experience*, *26*(12), 1439–1458. https://doi.org/10.1002/(sici)1097-024x(199612)26:12%3C1439::aid-spe71%3E3.0.co;2-1

Kumar, K., Algorithm, M. K.-M.-P., & Mandumula, K. (2011). *Knuth-Morris-Pratt Algorithm*. https://cs.indstate.edu/~kmandumula/presentation.pdf

Rochester University. (2020). *Finite Automata*. Www.cs.rochester.edu. https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/fa.html#:~:text=A%20finite%20automaton%20(FA)%20is

UC Davis. (2011). *0.1 Introduction*. https://www.cs.ucdavis.edu/~gusfield/cs224f11/bnotes.pdf