An Algorithm for String Searching

Rawan Ali Abdeen
Department of Computer Information Systems
Al-Balqa Applied University
Al-Salt, Jordan

ABSTRACT
Many problems encountered require string searching to solve them. Thus, string searching algorithms are important. They play the vital role in various fields and applications, including text editing, finding part of DNA in bio informatics engineering, text searching, computer security, linguistics, artificial intelligence and web search engines. In this paper, a string searching algorithm is proposed. The proposed algorithm aims to improve the brute-force searching algorithm. It finds all the occurrences of a pattern within a given text. The pattern and the text are to be preprocessed before the actual searching starts.

General Terms
String searching, string matching, pattern matching, algorithm.

Keywords
Searching, text, pattern, segment, string matching, string searching, pattern matching.

1. INTRODUCTION
Everyday people read, write and deal with character strings. They often need to find substrings (patterns or words) that match parts of the original text to solve many problems. To achieve this, string searching algorithms are required. String searching sometimes called string matching is the act of checking the existence of a substring (called the **pattern**) of length **m** in a sequence of characters (called the **text**) of length **n** (**where n ≥ m**) [1-4] and finding its location in that sequence or text.

A lot of algorithms were created and still is being created to improve and perform string searching. Each algorithm utilizes the pattern and text in the process of searching in different ways. Some of these algorithms require to preprocess the pattern [5-7], others need to preprocess the text; also there are algorithms that require both the pattern and the text to be preprocessed before searching [8]. However, there are algorithms that do not perform preprocessing neither for the text nor for the pattern. The **Brute-force** algorithm is one of the simplest string searching algorithms that were proposed. The problem is that it has the worst performance among the current existing algorithms. Therefore, various string searching algorithms were created to improve it. From those well-known algorithms: the **Knuth-Morris-Pratt** (KMP), **Boyer-Moore** (BM) and **Karp and Rabin** algorithms. Still to determine which of the algorithms is the best to use depends on the domain of the problem to be solved or the application were the algorithm is to be applied.

2. BRUTE-FORCE ALGORITHM
The Brute-force algorithm also called the "naïve" [1, 9, 10, 11] is the simplest algorithm that can be used in pattern searching. No preprocessing phase is required for the pattern or the text. It searches all the positions in the text between 0 and n-m. In this algorithm, the searching is done character by character between the pattern and a segment taken from the text. It starts with matching the first character of the pattern with the corresponding character of the segment taken. If the character of the pattern is equal to the corresponding character of the segment of the text then the next character in the pattern will be compared with the next character in the segment taken from the text (shifting one character forward at a time for the pattern as well as the text). If a mismatch occurs, then the algorithm will return to the first character of the pattern while taking the next segment of the text to be compared with the pattern. In other words, in the case of a mismatch, we shift forward ahead by one character of text and start matching it with the first character of the pattern. The process continues until the end of the text is reached.

The running time (time complexity) of the brute force algorithm is: O((n-m+1)m) which is O(nm) [9]. In the worst case, when n and m are equal, this algorithm has a quadratic running time [1]. The brute-force algorithm can be used when the problem to be solved is simple or when the speed of solving the problem is not as important as the simplicity of solving it. Moreover, for smaller text and pattern size, brute-force algorithm outperforms other algorithms [12].

3. RELATED WORK
The following sub-sections include descriptions of some of the algorithms that have improved the Brute-force algorithm.

3.1 Knuth-Morris-Pratt (KMP) Algorithm
The Knuth-Morris-Pratt (KMP) Algorithm uses information about the characters of the pattern to determine how much to move along the text after a mismatch occurs [10, 11] [13-16]. It exploits the fact that every time a match or a mismatch occurs, the pattern contains enough information to decide where the new examination should begin from. It achieves a running time of O(n+m).

3.2 Rabin–Karp Algorithm
The Rabin–Karp algorithm computes a hash function from the pattern to seek for it within a given text [11]. The hashing method used in this algorithm helped to avoid the quadric number of character comparisons in most practical situations.

This algorithm calculates a numerical or hash value for the pattern and for each m-character of the segment or substring of the given text (where m is equal to the number of characters of the pattern). The calculated numerical values are then used to perform the comparison instead of using the actual symbols. When a match is found, the algorithm compares the pattern with the substring character by character using the brute-force approach. Its time complexity is O(nm) and its expected running time in average would be O(n+m).

3.3 Boyer-Moore Algorithm

The Boyer-Moore algorithm preprocesses the pattern and then uses the information gathered during the preprocess step to avoid sections of the text. It works by searching the pattern from right to left, while moving it left to right along the text [10][14 -17]. Its time complexity is O(nm).

3.4 Occurrences Algorithm

Occurrences algorithm [18] finds all the occurrences of the pattern in the text. It performs preprocessing for the pattern and the text before searching. The searching process depends on the character that is repeated the most in the pattern. In this algorithm, after the preprocessing processes have been done, an array is created. This array will be used to determine which segments of the text will be compared with the pattern. Its time complexity is O(eleNum*m) where **eleNum** is the number of segments to be considered in the comparison.

3.5 Start-to-End Algorithm

The Start-to-End algorithm [19] does not preprocess neither the pattern nor the text to perform searching. It begins the search process by comparing the first character of the pattern with the first character of the segment taken from the text, if they match, then it compares the last character of the pattern with the last character of the segment, if a match occurs, then it will allow to perform character by character matching between the segment taken from the text and the pattern for the rest of the characters. Its time complexity is O(nm) in the worst case.

If the first character of the pattern does not match with any of the characters of the segment taken from the text then its time complexity would be O(n-m+1). However, if the first character of the pattern matches the first character of the segment of the text while the last character does not match then its time complexity would be O((n-m+1) *2).

4. THE PROPOSED ALGORITHM

In this paper, the proposed algorithm preprocesses the pattern and the text before the searching process is performed. It depends on the results of the preprocessing to find all the occurrences of the pattern in the text.

4.1 Preprocessing the Pattern

The algorithm of preprocessing the pattern finds the character that is of the highest occurrence in the pattern. **Figure 1** shows the flowchart of this process. Note that, if two or more characters have the same repetition within the pattern, then the algorithm chooses the character that occurs before the other in the pattern.

4.2 Preprocessing the Text

The text is divided into segments in which the first segment begins from the first character of the text, the second segment begins at the second character of the text and so on. That is each segment to be taken is shifted one character than the previous one. Note that, the number of characters in each segment is equal to the number of characters of the pattern. The algorithm takes each segment and checks if the character of the highest occurrence in the pattern exists in the segment or not. If it exists, then it calculates the number of occurrences of that character in that segment. If the number calculated is equivalent to the number of occurrences of that character in the pattern, then segment's index is stored in an array. At the end of the preprocessing of the text, the array will contain the indexes of the segments that will be considered in the comparison process. **Figure 2** shows the steps through which this process is performed.

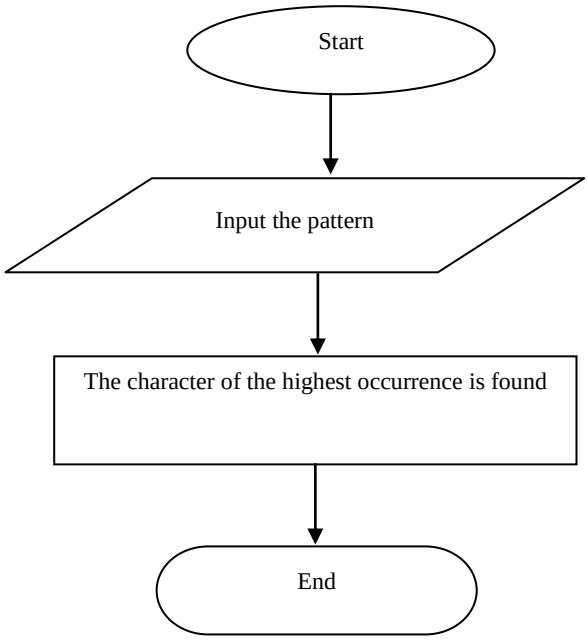

Figure 1: A general flowchart for preprocessing the pattern.

4.3 The Searching Process

After preprocessing the pattern and the text, the comparison will be done. This comparison will be between the pattern and the segments that their indexes are stored in the array.

For each segment taken from the created array, the first character in the pattern is compared with the first character in the segment, if they match, then the algorithm continues by comparing the last character in the pattern with the last character in the segment taken from the array. If the corresponding last characters match then the algorithm allows to perform character by character matching between the pattern and the segment taken from the text. In the case of a mismatch in any step, the algorithm will directly take the next segment in the array, else it will continue comparing.

If all the characters of the pattern match the corresponding characters of the segment taken from the created array then a message will be shown that informs that the pattern was found and at which location in the text it was found. After that, the algorithm checks if there are other segments in the array that have not been compared with the pattern. If so, then it takes the next segment and compares it with the pattern in the same manner as described above to find other occurrences of the pattern in the text.

If all of the segments determined by the array have been compared with the pattern without matching the pattern, then a not found message is shown. **Figure 3** illustrates the algorithm in a flowchart to show how to find the **first occurrence** of the pattern within the text.

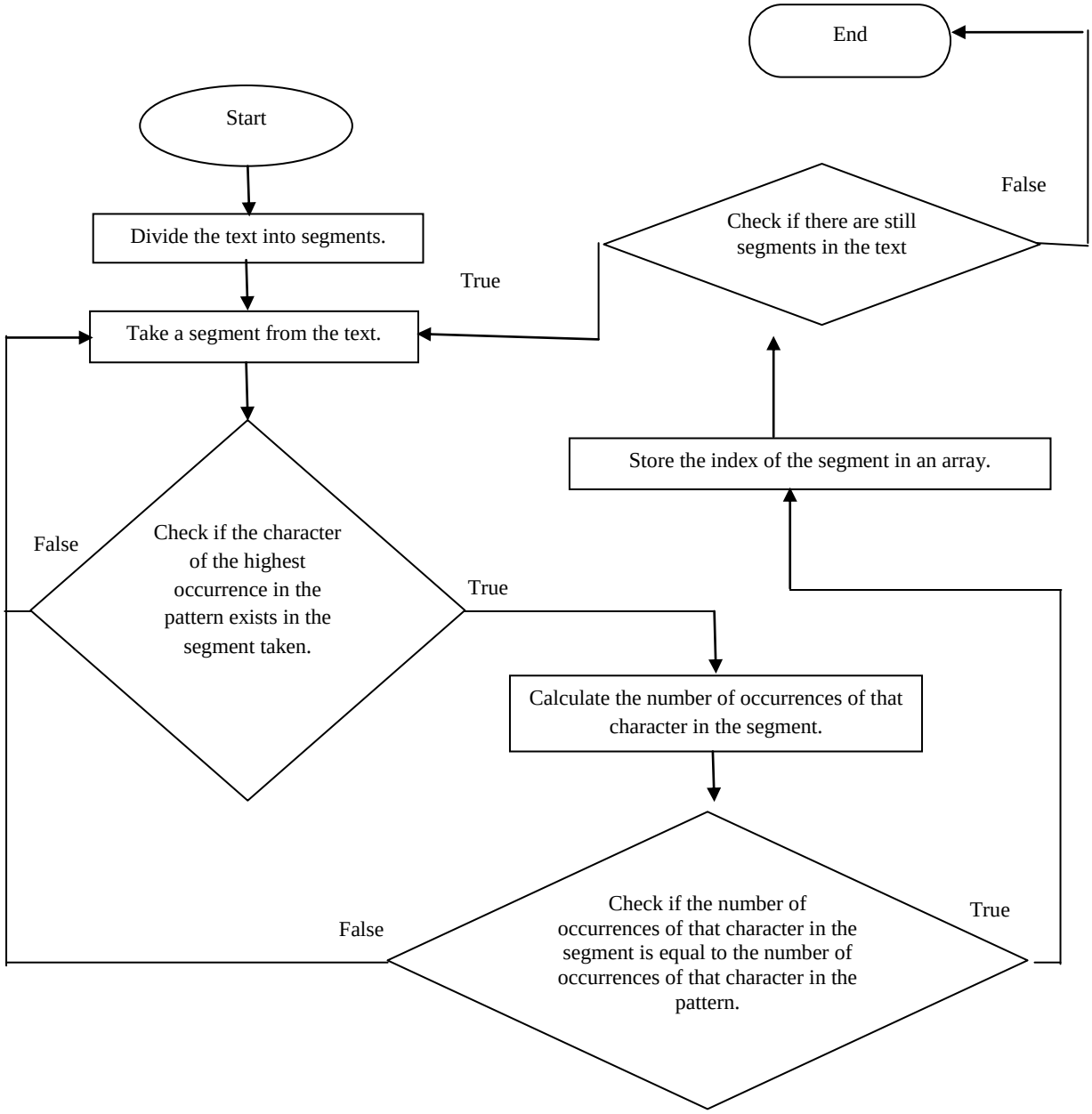

Figure 2: A flowchart for preprocessing the text.

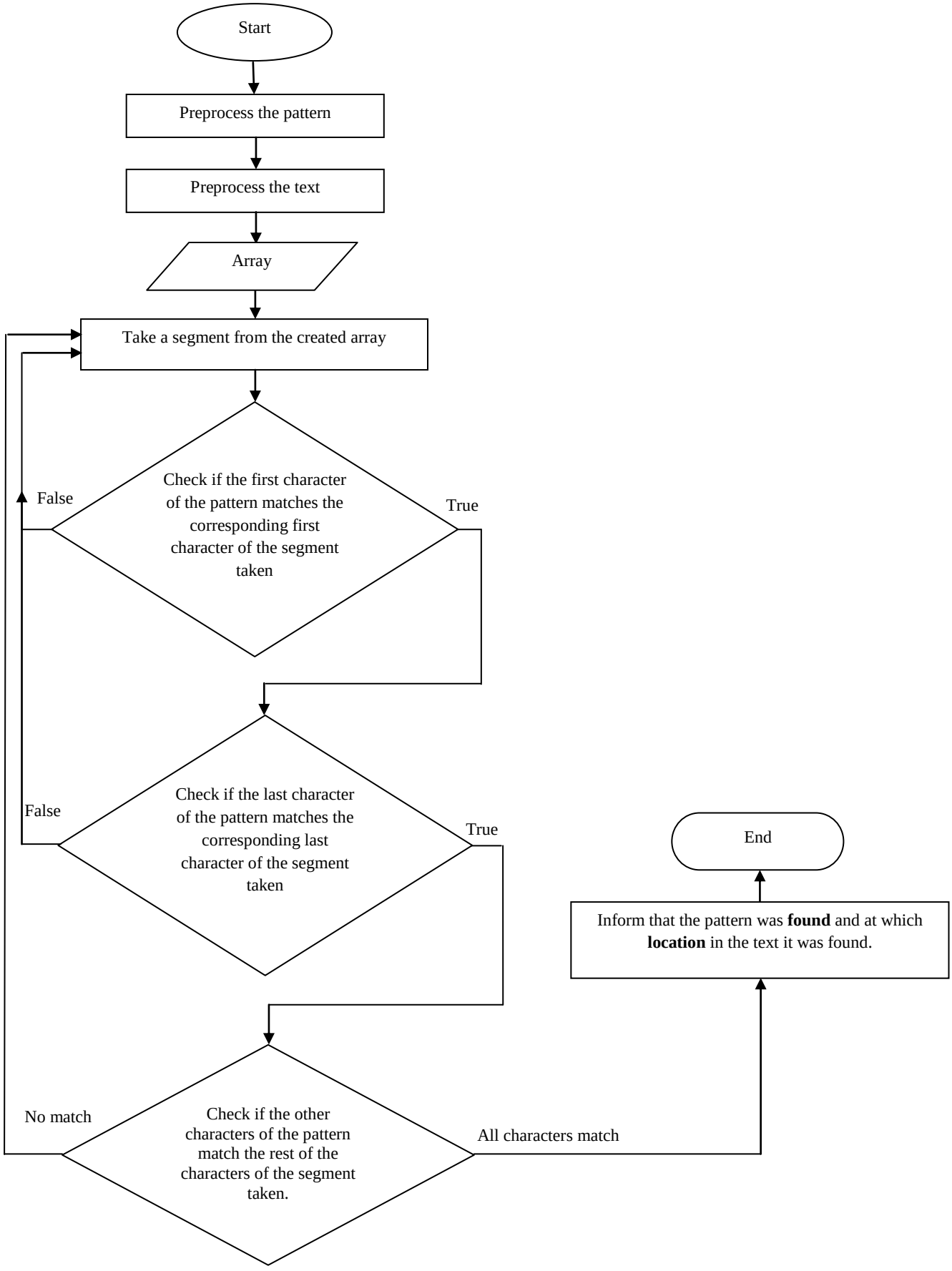

Figure 3: A flowchart for the proposed algorithm to find the first occurrence of the pattern within the text.

5. RESULTS

The proposed algorithm requires performing preprocessing for the pattern and for the text. It finds all the occurrences of the pattern in the text with the time complexity O(eleNum*m) in the worst case. The proposed algorithm has effectively reduced the time complexity of the brute-force algorithm. **Table 1** summaries the algorithms that has improved the brute-force algorithm with their time complexity and whether they need preprocessing or not. Note that, the length of the text is denoted as **n** and the pattern length is denoted as **m**. The time complexity for the proposed algorithm can be detailed as follows:

- If the first character of the pattern does not match the first character of all the segments in the created array, then the time complexity would be: **O(eleNum)**, where eleNum denotes the number of elements in the created array.
- If the first character of the pattern matches the first character of all the segments in the created array but the last character of the pattern does not match the last character of all the segments in the created array, then the time complexity would be: **O(eleNum*2)**.
- If the first character of the pattern matches the first character of all the segments in the created array and the last character of the pattern matches the last character of all the segments in the created array, then the time complexity would be: **O(eleNum*m)**.

Table 1: The Preprocessing needed and time complexity for the Brute-force algorithm and the algorithms that improved it

Algorithm	Preprocessing	Time Complexity in the worst case
Brute-Force Algorithm	No preprocessing	O(nm)
Rabin-Karp Algorithm	Preprocesses the pattern	O(nm)
Knuth-Morris-Pratt Algorithm	Preprocesses the pattern	O(n+m)
Boyer-Moore Algorithm	Preprocesses the pattern	O(nm)
Occurrences Algorithm	Preprocesses the pattern and the text	O(eleNum*m)
Start-to-End Algorithm	No preprocessing	O(nm)
Proposed Algorithm	Preprocesses the pattern and the text	O(eleNum*m)

Table 2: An Example to compare between the running time of the Brute-force algorithm and the Proposed algorithm

Text	Pattern	Algorithm	Time Complexity
zoom picture of the cat	zoom	Brute-Force	92
		Proposed	5

Table 3: An Example to compare between the running time of the Brute-force algorithm and the Proposed algorithm

Text	Pattern	Algorithm	Running time
zoom picture of the cat	groom	Brute-Force	115
		Proposed	2

Table 4: An Example to compare between the running time of the Brute-force algorithm and the Proposed algorithm

Text	Pattern	Algorithm	Running time
zoom picture of the cat	rat	Brute-Force	69
		Proposed	5

Tables 2, 3 and 4 include an example to find a specified pattern within a specified text and the running time required to perform the search by the Brute-force and the proposed algorithm. The observed running time of the proposed algorithm is much less than the running time of the Brute-force algorithm.

6. CONCLUSION

In this paper, a string searching algorithm has been proposed as an improvement for the brute-force algorithm. The proposed algorithm finds all of the occurrences of a pattern within a text. The pattern and text are to be preprocessed before the actual searching starts. The preprocessing processes aim to create an array that will include the indexes of the segments of the text that will be compared with the pattern. Those segments determined by the array are the only segments of the text that will be taken into consideration in the searching process without having to traverse all the segments of the text to find the pattern.

The searching process depends on the created array and also the matching of the first character of the pattern with the corresponding first character of the segment. If a match occurs then the corresponding last characters are to be compared before the character by character comparison is allowed to be done. Accordingly, the time required to perform the searching has been effectively reduced.

The proposed algorithm has reduced the time complexity of the brute-force algorithm. In the worst case, the time complexity of the brute-force algorithm is O(nm) while the time complexity of the proposed algorithm in the worst case is O(eleNum*m).

In a world heaving with a massive amount of data and applications that require extracting and detecting patterns, it is always required to create algorithms that can perform in a faster and more efficient manner than those existing or improving the already existed ones. For future, the proposed algorithm can be improved by modifying the way the search process matches the segments specified by the created array with the pattern. Furthermore, a study can be done to test the effectiveness of the proposed algorithm against other existing algorithms.

7. REFERENCES

[1] Thierry Lecroq, Experimental Results on String Matching Algorithms, SOFTWARE—PRACTICE AND EXPERIENCE, Vol. 25(7), pp. 727–765, 1995.

[2] G. Stephen, String Searching Algorithms, World Scientific, Singapore, 1994.

[3] Apostolico A. and Galil Z., Pattern Matching Algorithms, Oxford University Press, 1997.

[4] Robert Sedgewick and Kevin Wayne, Algorithms, 4th edition, ISBN-13: 978-0-321-57351-3, ISBN-10: 0-321-57351-X, Princeton University, Addison-Wesley, 2011.

[5] Liu Z., Du X. and Ishii N., An improved adaptive string searching algorithm, Software Practice and Experience, 28(2), pp. 191–198, 1988.

[6] Sunday D., A very fast substring search algorithm, Communications of the ACM, 33(8), pp. 132–142, 1990.

[7] Bruce W. and Watson E., A Boyer-Moore-style Algorithm for Regular Expression Pattern Matching, Science of Computer Programming, Vol. 48, pp. 99-117, 2003.

[8] Fenwick P., Fast string matching for multiple searches, Software, Practice and Experience, 31(9), pp. 815–833, 2001.

[9] Ohdan Masanori, Takeuchi Ryo and Satou Tadamasa, An Evaluation of String Search Algorithms at Users Standing, Proceedings of the 3rd WSES International Conference on Mathematics and Computers in Mechanical Engineering (MCME), pp. 4231-4236, ISBN: 960-8052-35-1, 2001.

[10] Softpanorama, Searching Algorithms, 2001, Available: http://www.softpanorama.org/Algorithms/searching.shtml. [Accessed: 8 July, 2019].

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.

[12] G.L. Prajapati, Mohd. Sharique, Piyush Nagani, Adarsh V., Study of Selected Shifting based String Matching Algorithms, International Journal of Computer Applications (0975 – 8887), Volume 140, No. 9, April 2016.

[13] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt, Fast pattern matching in strings, SIAM journal on computing, Vol. 6, No. 2, pp. 323-350, June 1977.

[14] Hume and Sunday, Fast String Searching, Software-Practice and Experience, Vol. 21(11), pp. 1221–1248, 1991.

[15] Michael T. Goodrich and Roberto Tamassia, Algorithm Design and Applications, John Wiley and Sons, 2015.

[16] M. Crochemore and D. Perrin, 1991, Two Way String Matching, Journal of the Association for Computing Machinery, Vol. 38, No. 3, pp.651-675.

[17] Robert S. Boyer and J. Strother Moore, A Fast String Searching Algorithm, Association for Computing Machinery, Vol. 20, No. 10, pp. 762-772, Oct. 1977.

[18] Mohammad Ababneh, Saleh Oqeili and Rawan A. Abdeen, Occurrences Algorithm for String Searching Based on Brute-force Algorithm, Journal of Computer Science, ISSN 1549-3636, 2(1), pp. 82-85, 2006.

[19] Rawan A. Abdeen, Start-to-End Algorithm for String Searching, IJCSNS International Journal of Computer Science and Network Security, Vol. 11, No. 2, pp. 179-182, February 2011.