A Comparative Study on Brute-Force String Matching and Boyer-Moore String Matching

Laet Laet Lin[#1], Myat Thuzar Soe[#2]

[#]Faculty of Computer Science, University of Information Technology
Yangon, Myanmar

[1]laetlaetlin@uit.edu.mm

[2]myatthuzarsoe@ uit.edu.mm

Abstract – **String matching is a very important issue in computer science, and it is widely used in most computer applications such as text editing, word processing, and image pattern-matching, etc. To solve the string-matching problem, there are distinctive string-matching algorithms. Such algorithms can significantly lessen the reaction time of the computer applications which utilize string matching. In this paper, a comparative analysis of Brute-Force (BF) and Boyer-Moore (BM) for string-matching algorithms has been presented. These algorithms are implemented in the java programming language and their performance efficiencies are compared based on the number of shifts, the number of comparisons, and the runtime in nanoseconds. Through a series of trials, the better performance of the BM algorithm is demonstrated in comparison to the BF algorithm. It has been discovered that the BM algorithm is efficient and has the least time complexity.**

Keywords **– Brute-Force (BF), Boyer-Moore (BM), Bad-symbol shift, Good-suffix shift, String matching.**

I. INTRODUCTION

String matching or string searching is the process of checking the presence of a given pattern string P of m characters in a longer text string T of n characters ($m \leqslant n$) and finding a substring of T that matches P [1]–[3]. The problem of finding one or all occurrences of a pattern string inside a given text string can be solved using different string-matching techniques. In solving vital tasks such as text editing, signal processing, speech and pattern recognition, library system, web search engine, information retrieval, and so on, string matching has played an important role.

Contingent on the text's size, the pattern's size and the number of the patterns to be matched, different string matching techniques have been advanced to reduce the number of comparisons and to keep away from a huge number of mismatches in the string-matching technique. The simplest approach to find whether the specified pattern happens inside the given text T is to check T at successive situations from left to right. This approach is called BF string matching. It is completely effective for short texts and patterns, but the efficiency decreases for enormous data [4]. To increase the effectiveness of the string-matching approach, some other algorithms are there which invoke preprocessing and distinctive order of comparison. A typical exact string-matching algorithm, the BM algorithm uses preprocessing and reverse order search logic that means the algorithm compares the pattern P with the text T within a sliding window in the right-to-left order [5].

String matching is one of the essential issues in information technology. Moreover, the most fundamental time-consuming query in most of the applications is string matching. Better string-matching algorithms can improve the applications' efficiencies more effectively. In this way, a fast string matching algorithm is a significant area of research.

In this paper, two exact string-matching algorithms are compared based on their comparative analysis. These two algorithms that are covered in this paper are the BF algorithm and the BM algorithm. This paper is organized as follows. Section II incorporates a little depiction of some string-matching algorithms alongside the BF algorithm and the BM algorithm in detail. Section III highlights the comparative analysis and evaluation results between BF and BM algorithms. Section IV is devoted to the outline of the embodiment of the string-matching algorithm in the current problems.

II. BACKGROUND THEORY

In the area of text handling, string matching acts as a significant job. A string-matching algorithm is viewed as a proficient string-matching algorithm if it is complete and can discover a pattern string in a text string by the least number of comparisons inside the effective time limit. Some algorithms utilized to perform string-matching activity are BF, Karp-Rabin (KR), BM, and Knuth-Morris-Pratt (KMP) algorithms [6].

Here, let's think the lengths of text and pattern are n and m, respectively. The BF algorithm is an extremely essential string-matching algorithm with O(mn) time complexity [4]. An examination utilizes a hashing function with expected execution time O(n + m) whose preprocessing stage is in O(m) time complexity and the searching stage is in O(mn) time unpredictability with consistent space. Such an algorithm can be utilized to match different patterns [7]. Because of various pattern matching abilities, such an algorithm can be utilized to distinguish counterfeiting in any event, for bigger phrases. KMP algorithm plays out the calculation by avoiding bunches of pointless comparisons with increment its effectiveness. It plays out the comparisons from left to right. The time and space complexity of the preprocessing stage is O(m) and the searching stage is in O(n + m) time complexity. KMP algorithm gives a preferable performance over the BF algorithm. Then again, the BM algorithm is faster than the KMP algorithm [6].

Though the BF algorithm is not as effective as KR, KMP, and BM string-matching algorithms, it has a few favorable circumstances over them. It does not need any preprocessing, so it does not need extra space. It may very well be powerful for short texts and patterns. On the other hand, the preprocessing stage time and space complexity of the BM algorithm is O(m + n) and it gives better performance while the size of the pattern increments [5]. To contemplate the veering practices of BF and BM

algorithms, a comparison between them is represented in this paper. Detailed depictions of BF and BM string-matching algorithms are as follows.

A. Brute-Force (BF) String-Matching Algorithm

A BF string-matching algorithm for solving the string-matching problem is completely the simplest algorithm. Firstly, the algorithm begins by aligning the pattern P with the very first characters in the text T and compares the corresponding character pairs from left to right until all the character pairs match or a mismatching character pair is found. For the mismatched situation, P is shifted to the right precisely by one position, and the character comparisons are resumed, beginning again with P's first character and its partner in T. In this way, the algorithm checks whether P's occurrence is there or not, at all positions of T from *0* to *n – m* [2], [4]. Past that position (*n – m)*, the characters are insufficient to match the whole P; so, the algorithm does not require any comparisons here [2].

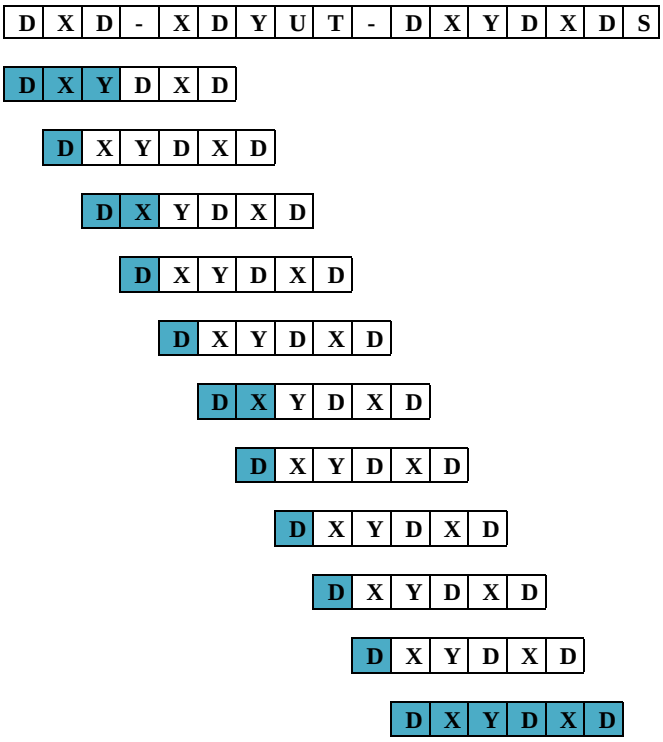

Fig. 1 An operation of the BF string-matching algorithm. The characters of the pattern which are compared with their partners in the text are described in blue colour. When a mismatch happens, the pattern P is moved to the right exactly by one position. Here, the total number of shifts is 10 and the total number of comparisons is 20.

The pseudocode of the BF algorithm is as follows.

Algorithm BFStringMatching (Pat[0...m–1], Txt[0...n–1])
//**Input:** An array Pat[0...m–1] of m characters representing
//a pattern string and an array Txt[0...n–1] of n characters
//representing a text string
//**Output:** The position of the text's first character that
//begins a matching substring if the search is successful, or
//–1

```
for j ← 0 to n – m do
        k ← 0
        while k < m and Pat[k] = Txt[j + k] do
                k ← k + 1
        if k = m return j
return –1
```

The sample of the BF algorithm is as follows.
Text string: DXD-XDYUT-DXYDXDS
The pattern string for searching: DXYDXD
Fig. 1 illustrates the searching process of this sample in BF algorithm.

The BF algorithm does not require preprocessing because due to its simplicity there is nothing to be prepared before the algorithm starts. But it is not optimal, as the pattern length increases, the number of comparisons increments individually. In the algorithm, the outer loop is executed at most (n – m + 1) times, and the inner loop is executed at most (m) times. The algorithm can make *m* comparisons prior to moving the pattern, and this can occur for every one of the (n – m + 1) attempts. Therefore, the execution time of the BF algorithm is O(m (n – m + 1)) that is O(mn) [5]. If m and n are the same, its worst-case execution time is quadratic [1]. If the problem to be solved does not complicate or if the speed to solve the problem is also not important, the BF algorithm can be utilized. Besides, the BF algorithm surpasses other string-matching algorithms for text and pattern which have a smaller size [8].

B. Boyer-Moore (BM) String-Matching Algorithm

BM algorithm is the most popular string-matching algorithm because of its efficient nature. Based on BM's concept, many string-matching algorithms were developed [9], [10]. BM algorithm compares a pattern's characters with their partners in the text by moving from right to left [1], [2], [11]. It determines the size of the shift by using a bad-symbol shift table and a good-suffix shift table.

1) *Bad-Symbol Shift*: The bad-symbol shift is directed by the character *C* in the text T that does not match with its partner in the pattern P. If the text's C does not include in P, P is shifted to pass this text's C. This shift size is calculated by the following equation:

$$txt(C) - r$$

where *txt(C)* is an entry in the bad-symbol shift table, and r is the number of matched characters.

This table is a list of possible characters that can be found in the text. Texts may contain punctuation marks, space, and other special characters. The *txt(C)* is computed by the following formula:

$$txt(C) = \begin{cases} \text{Length m of the pattern P, if character } C \text{ in the text is not among P's first (m – 1) characters;} \\ \text{Distance from the rightmost character } C \text{ amidst P's first (m – 1) characters to its final character, otherwise.} \end{cases}$$

Let us build a bad-symbol shift table for searching a pattern DBPDGP in some text. It is shown in Table 1, all entries of the table are equal to 6, except for B, D, G, and P, which are 4, 2, 1, and 3, respectively.

TABLE I
BAD-SYMBOL SHIFT TABLE

C	B	D	G	P	Other Characters
txt(C)	4	2	1	3	6

An example of shifting the pattern using the above Table 1 is shown in Fig. 2. In this example, the pattern DBPDGP is searched in some text string. Before the failure of comparison on the text's character Y, the comparisons of the last two characters were matched. So, the pattern can be moved 4 positions to the right because the shift size is txt(Y) − 2 = 6 − 2 = 4.

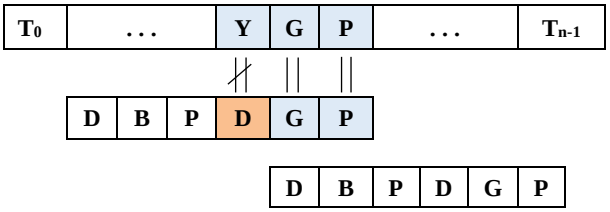

Fig. 2 Shifting the pattern using bad-symbol table

Bad-symbol shift d_1 is calculated by *txt(C) − r* if this value is greater than zero and by 1 if it is less than or equal to zero. This is described as a formula:

$$d_1 = max\{txt(C) - r, 1\}.$$

2) *Good-Suffix Shift:* The good-suffix shift is directed by a match of the pattern's last r > 0 characters. The pattern's ending portion is referred to as its suffix of size r and denotes it *suf*(r). If other events (not preceded by the same character as in its rightmost event) of suf(r) contain in P, then P can be shifted according to the distance d_2 between like a second rightmost event of suf(r) and its rightmost event. In the case not including another event of suf(r) in P, the longest prefix of size k < r which matches the suffix of the same size k is needed to be found. If such a prefix exists, the shift size d_2 is calculated by the distance between the corresponding suffix and this prefix; d_2 is set to P's length m, otherwise.

Table 2 shows a sample of the good-suffix shift table for the pattern AFCFAF.

TABLE II
GOOD-SUFFIX SHIFT TABLE

r	Pattern	d_2
1	AFC$\bar{F}$AF	2
2	$\overline{AF}$CFAF	4
3	$\overline{AF}$CFAF	4
4	$\overline{AF}$CFAF	4
5	$\overline{AF}$CFAF	4

The steps in the BM string-matching algorithm are as follows:

Step 1: Build a bad-symbol shift table as discussed earlier for a specified pattern P and the alphabet used in both text T and P.

Step 2: Build a good-suffix shift table as discussed earlier by using P.

Step 3: Against the starting of T, align P.

Step 4: Until either a matched substring is found, or the P string comes to past the last character of T, repeat the following step. Beginning with P's last character, compare the corresponding characters of P and T until all *m* pairs of the characters match or a character pair that mismatch is found after r ≥ 0 pairs of the characters are matched. In the mismatched situation, fetch *txt(C)* from the bad-symbol shift table where *C* is T's mismatched character. When r > 0, additionally fetch the corresponding d_2 from the good-suffix shift table. Move P to the right according to the number of positions calculated by the formula:

$$\boldsymbol{d} = \begin{cases} d_1 & for\ r = 0, \\ \max\{d_1, d_2\} & for\ r > 0, \end{cases}$$

where $d_1 = \max\{txt(C) - r, 1\}$.

Here is a sample of the BM algorithm for searching the pattern string DXYDXD in the text string DXD-XDYUT-DXYDXDS.

First, construct the bad-symbol shift and the good-suffix shift tables. The bad-symbol shift table to find d_1 value and the good-suffix shift table with d_2 values are shown in Table 3 and Table 4, respectively.

TABLE III
BAD-SYMBOL SHIFT TABLE FOR ABOVE SAMPLE

C	D	X	Y	Other Characters
txt(C)	2	1	3	6

TABLE IV
GOOD-SUFFIX SHIFT TABLE FOR ABOVE SAMPLE

r	Pattern	d_2
1	DXY$\bar{D}$XD	2
2	$\overline{D}$XYDXD	5
3	$\overline{D}$XYDXD	5
4	$\overline{D}$XYDXD	5
5	$\overline{D}$XYDXD	5

As shown in Fig. 3, first, the pattern string is aligned with the starting characters of the text string.

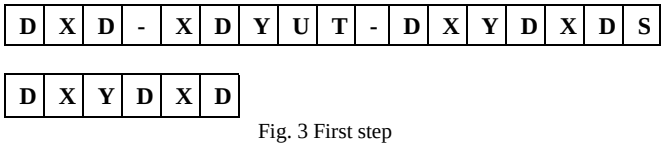

Fig. 3 First step

As shown in Fig. 4, after matching with two pairs of characters, the pattern's character 'D' fails to match its partner '-' in the text. So, the algorithm retrieves txt(-) = 6 from bad-symbol table to compute d_1 = txt(-) – 2 = 4 and also retrieves d_2 = 5 from good-suffix shift table. And then the pattern is moved to the right by max {d_1, d_2} = max {4, 5} = 5.

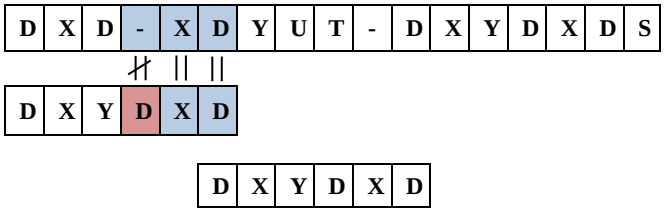

Fig. 4 Second step

As shown in Fig. 5, after matching one pair of D's and failing the next comparison on the text's '-' character, the algorithm fetches txt(-) = 6 from bad-symbol table to compute $d_1 = 6 - 1 = 5$ and also fetches $d_2 = 2$ from good-suffix shift table. And then the pattern is moved to the right by max $\{d_1, d_2\}$ = max {5, 2} = 5.

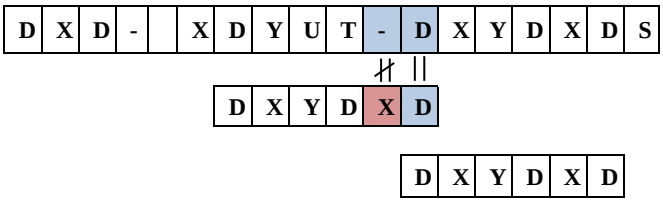

Fig. 5 Third step

Lastly, after matching all the pattern's characters with their partners in the text, a matching substring is found in the text string. Here, the total number of shifts is 2 and the total number of comparisons is 11.

The worst-case time complexity of the BM algorithm is linear if only the very first occurrence of the pattern is searched [2]. In general, BM algorithm runs faster if the pattern is longer. It can make one or more right-shifts by using the bad-symbol shift rule and the good-suffix shift rule. So, the algorithm does not have to check each character of the text string. Besides, the algorithm takes O(m) comparisons when the pattern string is absent in the text string. The best-case time efficiency of the BM algorithm is O(n/m) [4], [12].

III. Comparative Analysis and Evaluation Results

The primary downside of the BM algorithm is the preprocessing time and the space needed, which relies upon the pattern size and the alphabet size [13]. BF algorithm does not need preprocessing for the pattern or the text strings; however, its problem is the process slowness, and it produces effective outcomes rarely [14]. On the other hand, the BM algorithm is amazingly quick, especially on large alphabets, and it evades many unnecessary comparisons according to the pattern relative to the text [14]. By the study of the researchers [15], the BM algorithm is the best for the typical searching process. The behaviours of BF and BM algorithms are presented in Table 5.

An example of the comparative analysis between the BF algorithm and the BM algorithm is shown in Table 6. In this example, the different pattern strings like "*Force*", "*between*", "*algorithm", "boyer-moore*" and "*sample result*" are searched in a given text string of 102 characters: *"An example of the comparative analysis between the Brute-Force algorithm and the Boyer-Moore algorithm"*.

In Table 6, the observed result of the BM algorithm is very efficient than the BF algorithm.

TABLE V
Behaviours of BF and BM Algorithms

Behaviours	Comparison Order	Rules of Shifting	Preprocessing Stage
BF Algorithm	Left to right order	One by one character shift	no
BM Algorithm	Right to left order	Both bad-symbol shift rule and good-suffix shift rule	yes

TABLE VI
Example of Comparison Between BF Algorithm and BM Algorithm

BF Algorithm			**BM algorithm**		
No. of Shifts	No. of Comparisons	Runtime in Nanoseconds	No of Shifts	No. of Comparisons	Runtime in Nanoseconds
57	62	496200	13	19	260300
39	46	676000	6	13	325300
63	78	706200	11	20	327800
91	93	656000	8	14	313600
89	92	664200	9	11	314500

This paper compares the evaluation results between BF and BM algorithms based on the number of shifts, the number of comparisons, and the runtime in nanoseconds. These two algorithms are implemented in the java language and are compared by searching the patterns of different sizes like 3, 5, 7, 9, 11, 13, and 15 respectively in a text string of 1021 characters.

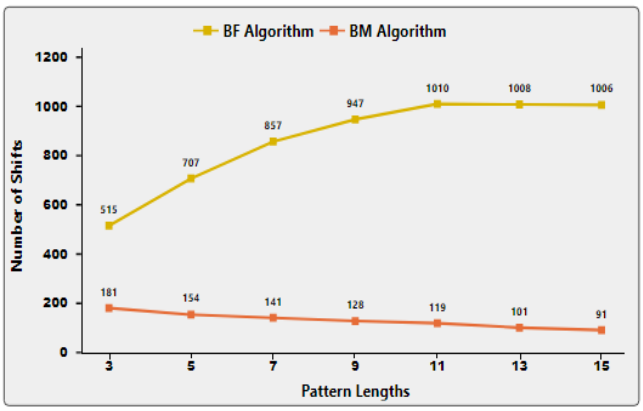

Fig. 6 Evaluation based on the number of shifts

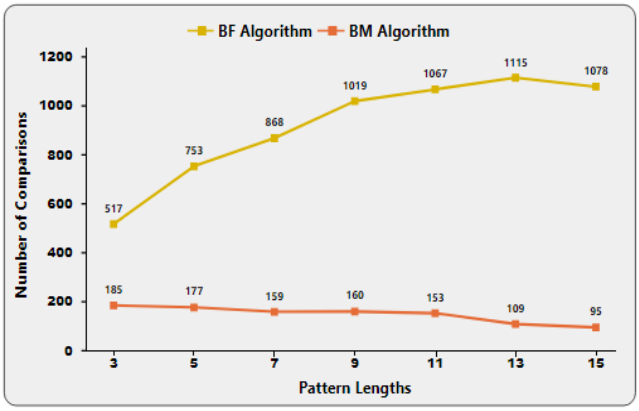

Fig. 7 Evaluation based on the number of comparisons

1) Evaluation Based on the Number of Shifts: Using different pattern lengths, the total numbers of shifts made by each algorithm are shown in a graph form in Fig. 6. As the results in the graph shows, the BM algorithm took minimum shifts as compared to the BF algorithm. In the case not including the pattern string of length m in the text string of length n, the total number of shifts for the BF algorithm is always (n – m). But the number of shifts for

the BM algorithm can vary based on bad-symbol shift and good-suffix shift. As the pattern's length increases, the BM algorithm becomes more and more efficient.

2) Evaluation Based on the Number of Comparisons: Using different pattern lengths, the total number of comparisons by each algorithm is shown in the graph form as in Fig. 7. The number of comparisons for both algorithms can vary based on the pattern string. According to the results in the graph, the BM algorithm produces a more efficient result than the BF algorithm.

3) Evaluation Based on the Runtime in Nanoseconds: Using different pattern lengths, the runtime in nanoseconds by each algorithm is shown in the graph form as in Fig. 8. According to the results in the graph, the BM algorithm is very faster than the BF algorithm.

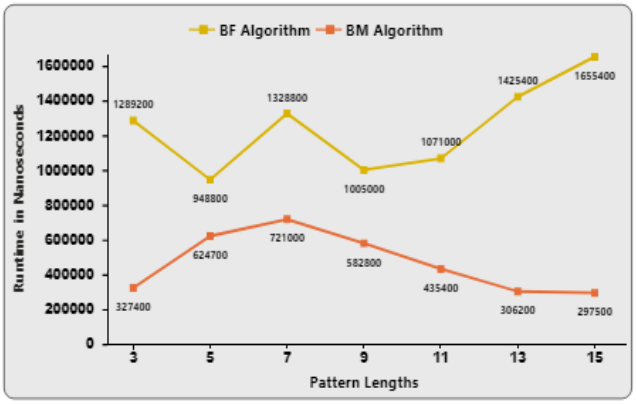

Fig. 8 Evaluation based on the runtime in nanoseconds

IV. Conclusions

In this paper, the evaluation results based on the number of shifts, the number of comparisons, and the runtime in nanoseconds show that the BM algorithm is much more efficient than the BF algorithm. In the current field, to find the specific content in the least time is the most basic factor. String-matching algorithms play a critical role in such a manner. A great deal of research is going on at the level of software and hardware to make pattern searching quicker. By the application of various algorithms in various fields, the approximate best algorithm can be learned. It has been discovered that most applications utilize the BM algorithm for its attractive and proficient work, and most of the other applications utilize the basics of this algorithm as it has the least time complexity.

Acknowledgment

We are appreciative of the advisors from the University of Information Technology who gave us accommodating remarks and suggestions all through this project.

References

[1] R. A. Abdeen, "An Algorithm for String Searching", *International Journal of Computer Applications* (0975 – 8887), Vol. 177 – No. 10, October 2019.

[2] A. Levitin, *Introduction to the design and analysis of algorithms*, 3rd edition, ISBN 10: 0-13-231681-1, ISBN 13: 978-0-13-231681-1, Villanova University, Pearson, 2012.

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

[4] A. A. Manaf, A. Zeki, M. Zamani, S. Chuprat, E. El-Qawasmeh, *Informatics Engineering and Information Science*, Part II, Springer Heidelberg Dordrecht, New York, 2011.

[5] R. S. Boyer and J. S. Moore, "A fast string searching algorithm", *Communications of the ACM*, vol. 20, (10), pp. 762, 1977.

[6] P. Gupta, V. Agarwal and M. Varshney, *Design and analysis of algorithms*, PHI Learning Pvt. Ltd., 2012.

[7] G. A. Stephen, *String Searching Algorithms*, World Scientific Publishing Co Pte Ltd, Singapore, 2000.

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

[9] R. Rahim, A. S. Ahmar, A. P. Ardyanti, and D. Nofriansyah, "Visual Approach of Searching Process using Boyer-Moore Algorithm.", *Journal of Physics*, vol. 930, 2017.

[10] I. Anggraeni and Mulyati, "Searching Process using Boyer Moore Algorithm in Medical Information Media", *International Journal of Recent Technology and Engineering* (IJRTE), ISSN: 2277-3878,Vol. 8 Issue-2S7, July 2019.

[11] M. T. Goodrich and R. Tamassia, *Algorithm Design and Applications*, John Wiley and Sons, 2015.

[12] Y. Liao, "A Survey of Software-Based String matching Algorithms for Forensic Analysis", *Annual ADFSL Conference on Digital Forensics, Security and Law. 2.*, 2015.

[13] R. A. Baeza-Yates (2002), String Searching Algorithms. [Online]. Available: http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap10.htm

[14] Pandiselvam. P, Marimuthu. T, Lawrance. R., "A Comparative Study on String Matching Algorithm of Biological Sequences", *International Conference on Intelligent Computing*, 2014.

[15] A. Kumar, V. Sharma, S. Kumar, "A comparative analysis of various exact string matching algorithms for Virus-Signature Detection", *First International Conference on Emerging Trends in Soft Computing and ICT* (SCICT-2011), 2011.