Pattern Matching based on Boyer Moore Algorithm, A New Method

Julio C. Rivera , Paul M. Di Gangi , James L. Worrell , Samuel C. Thompson , Allen C. Johnston
Management, Information Systems, and Quantitative Methods Department
The University of Alabama at Birmingham
Birmingham, AL 35294, USA
(205) 934-8890
jrivera@uab.edu

ABSTRACT: *Pattern matching actually is consisting of searching a pattern inside of a text. It is a fundamental challenge in the field of computer science. It is important to any string matching algorithm to be able to locate quickly some or all occurrences of a user-specified pattern in a text. Although There had been presented many algorithms in this field. Boyer More Algorithm (BM), among them, is one of the most efficient algorithms that has used in many applications. In this study, we had proposed a new pattern matching technique rely on utilizing Boyer More Algorithm, based on its second order. The proposed method uses matched partition in the previous step in order to avoid unnecessary comparisons. Afterwards, the proposed algorithm has been implemented and also has been compared with existing algorithms. The comparison results show that our proposed method has less time complexity than other existing techniques, especially than Boyer Moore algorithm.*

Keywords: Pattern matching, Knuth-Morris-Pratt Algorithm, Boyer Moore Algorithm, Naive Algorithm

Received: 2 June 2015, Revised 30 June 2015, Accepted 5 July 2015

1. Introduction

Pattern matching is a fundamental challenging problem in computer science. It has been extensively studied and many techniques and algorithms have been designed to solve this problem. These algorithms are mostly used in information retrieval, bibliographic search, computational biology and question answering applications [10, 11].

A string-matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then, it checks if the pattern occurs in the window, otherwise it shifts the window to the right. This procedure repeats again until a matching occurs or the right end of the window goes beyond the right end of the text (Amintoosi *et al*., 2006). The primary way to decrease the time needed is tio decrease the number of comparisons made by each algorithm.

It is consisting of searching a predefined pattern P of length m inside of a text T of length n. A pattern matching algorithm aligns the pattern with the beginning of the text and keeps shifting the pattern forward until the end of the text or a match is reached. It seems to be a very simple problem but it is not.

Finding a pattern inside of a long text at least possible time is implicated to utilize algorithms that have least possible time and space complexity. Therefore, there have been various algorithms presented to minimize time and space complexity in different text pattern. Some exact string matching algorithms are Brute force algorithm, Naïve algorithm, Boyer-Moore algorithm [3], Knuth-Morris-Pratt (KMP) Algorithm [7]. In this paper, we present a brief literature review of these algorithms.

The rest of the paper is organized as follows. We briefly present the Background and related work in section 2. Section 3 deals with the proposed model. We make some concluding remarks from the experiments in Section 4.

2. Background and Related Work

To check whether the given pattern is present in the sequence or not we need an efficient algorithm with less comparison time and low complexity.

Brute Force Algorithm

The worst algorithm for string matching is brute force algorithm. It enumerates all match character of the pattern with the text at the same position, and we succeed if and only if its value is equal to the pattern size. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The C++ code for brute force method:

```
for (i = 0; i < n - m; i ++)
{
       num = 0;
       for (j = 0; j < m; j ++) ;
       if (t [i+j] == p [j])
       num++;
       if (num = = m) return i;
}
return -1;
```

Where a text string *t* of length *n*, and a pattern string *p* of length *m* are given as inputs. Therefore, the running time of this algorithm is *O (mn)*, where m is the length of the pattern and n is the length of the text and the expected number of text character comparisons is 2*n*. We could easily improve it by sliding the pattern over one character when a mismatched occur- naïve algorithm.

Naïve Algorithm

The naïve algorithm searches for a pattern in a text by matching the first character of the text with the first character of the pattern, and if we succeed, try to match the second character, and so on[1]; if we hit a mismatch, slide the pattern over one character and try again. When we find a match, return its starting location. It always shifts the window by exactly one position to the right.

The C++ code for naïve method:

```
for  (i = 0; i < n-m; i ++)
     {

     for (j = 0; && j < m && T [i + j] = = P [j]; j ++) ;
     if  (j == m)   return i; //  found a match
     }
return -1    // not found a match
```

In practice this works better than brute force algorithm- not usually as bad as O(mn) at the worst case. This is because the inner loop usually finds a mismatch and shifts the pattern over the text as long as one character without going through all m steps. But this algorithm still take O(mn) at the worst case analysis.

The advantage of aforementioned methods is that they don't need of pre-processing stage and they don't require extra space. But their important weakness is their huge time complexity. So that when we are facing with a long text, they lose their performance. Therefore, we have to use some methods that can afford of huge time complexity in facing with a long text, that we are going to search a pattern inside it.

Knuth-Morris-Pratt

Knuth-Morris-Pratt (KMP) [2] algorithm is proposed in 1977 to speed up the procedure of exact pattern matching by improving the lengths of the shifts (the amount of jumping along the text in one comparison level).
It compares the characters from left to right of the pattern as same as naïve algorithm.

Before starting
When a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin(the amount of jumping level the pattern along the text).

The key to not examining every character in the text is to use information learned in failed match attempts to decide what to do next. This is done in Knuth- Morris-Pratt algorithm [2], as we will see shortly. Although, the Knuth-Morris-Pratt algorithm has better worst-case running time than the other efficient algorithm such as BM, the latter is known to be extremely efficient in practice (Watson and Watson, 2003).

The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions *T[i]...T[i+j-1]*. You can use this knowledge to save work as you can skip some iterations for which no match is possible. The value of jumping characters (j) is just a function of the value of matched characters (i) and does not depend on other information.

The entire KMP algorithm consists of overlap computation followed by the main part of the algorithm in which we scan the text (using the overlap values to speed up the scan). The first part takes O(m) and the second part takes *O(n)* time. Therefore, the whole KMP algorithm runs in time *O(n + m)*, which is much better than the simple quadratic time algorithm.

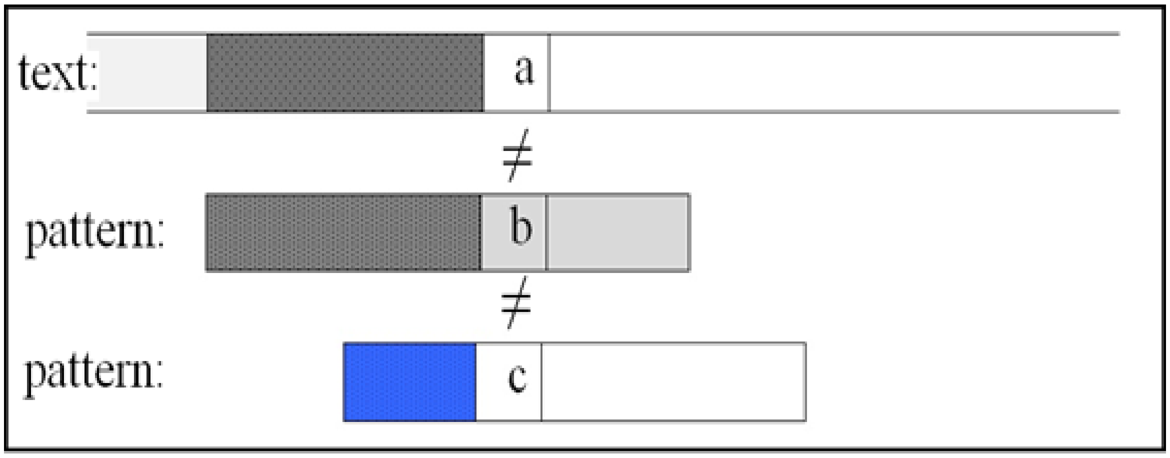


Figure 1. Knuth-Morris-Pratt Matching Method

The KMP algorithm works by turning the patterns given into a machine, and then running the machine. It takes
O(m) space and time complexity in pre-processing phase, and *O(n+m)* time complexity in searching phase (independent of the alphabet size). KMP is a linear time string matching algorithm. [14]

Boyer-Moore Algorithm

Since 1977, with the publication of the Boyer-Moore algorithm, there have been many papers published that deal with exact pattern matching and in particular discuss and/or introduce variants of Boyer-Moore algorithm. It performed character comparisons in reverse order from right to the left of the pattern and did not require the whole pattern to be searched in case of a mismatch.

Boyer-Moore algorithm holds a window containing pattern over the text, much as the Knuth-Morris-Pratt algorithm does except that searching process is from right of the pattern instead of left to right. This window moves along the text, however, its improved performance is based around two clever ideas:

• Inspect the window from right to left Perhaps the most surprising feature of this algorithm is that its checks to see if we have a successful match of p at a particular location in t work backwards. So if we are checking to see if we have a match starting at t [i], we start by checking to see if p[m] matches t [$i + m$], and so on.

• Recognize the possibility of large shifts in the window without missing a match. Actually, it is not necessary to examine every character in the text in order to locate the pattern. The key is to use information learned in failed matched attempts to decide what to do next. This is done with the use of precomputed tables.

Although Knuth, Morris, and Pratt were able to achieve a much better algorithm than naïve algorithm, they were still unable to achieve a sub linear algorithm in the average case.

The idea behind the Boyer-Moore algorithm is information gain. All of the previous approaches attempted to solve the problem by examining the first characters in the pattern. Boyer and Moore believed that more information was actually gained by beginning the comparison from the end of the pattern instead of the beginning. The Boyer-Moore algorithm was successful at performing the string searching task in sub linear time in the average case, something that no other algorithm at the time could accomplish. The algorithm has stood the test of time and is still used as a comparison/benchmark when new algorithms are introduced trying to improve on the running time.

In this algorithm, as like as KMP algorithm, we have two steps: the preprocessing step and the searching step. At the processing step, we determine the length of jump for any possible mismatch position. In order to maximize the length of jump, this approach uses the match characters as shown as Figure2. As we can see, the shift amount for a mismatch calculate so that the position of match characters before jumping (are shown as grey level) have to equal to their corresponding character after jumping. Afterwards, at the searching step, upon the occurrence of the first mismatch at a position of the pattern, the pattern shifts somewhat forward along the text. The magnitude of this shift is based on the position of mismatch in the pattern, that is calculated in the preprocessing step.

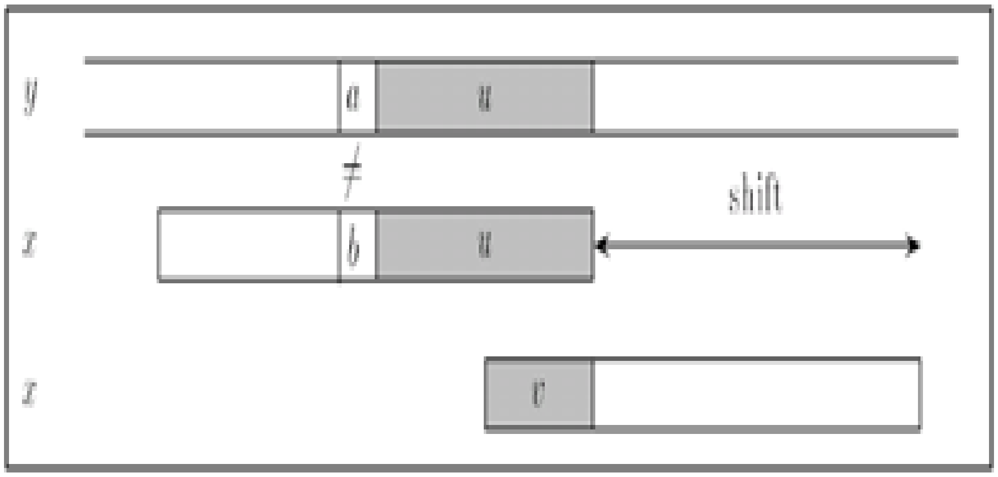
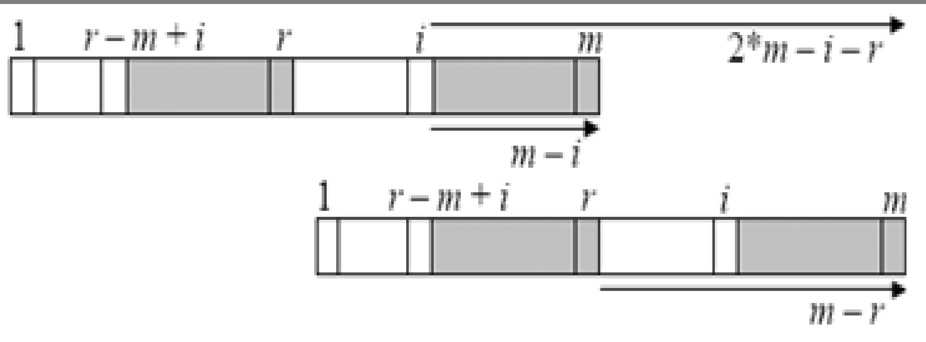


Figure 2. Boyer Moore Method



$$
\begin{aligned}
Match(p, i) = max\{r \mid & (m - i < r < m \wedge p[i] \neq p[r - m + 1] \\
& \wedge \forall j.(i + 1 \leq j \leq m \Rightarrow p[j] = p[r - m + j])) \\
& \vee (0 \leq r \leq m - i \\
& \wedge \forall j.(1 \leq j \leq r \Rightarrow p[j] = p[m - r + j]))\}
\end{aligned}
$$

Figure 3. Computation of jumping amount in Boyer-Moore Algorithm for ith mismatch position.

As we can see from Figure3, Match (*p, i*) is assigned to compute the jumping length of the pattern along the text, where i is position of mismatch along the pattern. As it is figure out in Figure3, assume that we have had (*m-i*) matches from the right, the (i-th) next character when compared is a mismatch. We can shift down our pattern to the next occurrence such that p[r-m+i] isn't equal to *x[i]* and also *x[i], ..., x[m]* are as same as *p[r-m+i]*,..., *p[r]* respectively.

Usually the numbers of matched characters in every matching search attempt are very small. In other words, the mismatch position probably is in the last of the pattern. Therefore, the amount of jump in Boyer- Moore Algorithm is usually very small. In this paper, we have solved this problem. Our proposed method uses two last searching attempt of Boyer-Moore algorithm instead of one in Boyer-Moore algorithm. Actually, we not only use the last searching attempt as Boyer-Moore algorithm, but we also use the searching attempt before that. The implementation of the proposed method has described in the following section.

3. The Proposed Method

In order to reduce the processing time of the BM algorithm,

The proposed algorithm improves the length of the shifts of the BM algorithm. The extensive testing of the proposed algorithms yields to speeding up the BM algorithm.This algorithm performs best when preprocessing of the text is not possible or not desired.

In this algorithm, firstly, we determine the length of jump (Long_Jump[i]) for every possible mismatch along the pattern, by using of Boyer-Moore algorithm. Then, we compute the length of two step jump (Long_Jump2[mba, mna]), where mna is the position of a mismatch at the current searching attempt and also mba is the positin of a mismatch at before that- the one before current searching attempt. In order to develop our algorithm, we consider three possible observations for to consequence searching attempts:

Observation 1:
All of the matched characters in two consequences searching attempts are aligned the pattern (see Figure 4). The next searching attempt must consider these two consequences in order to figure out the jumping length of the following shift.

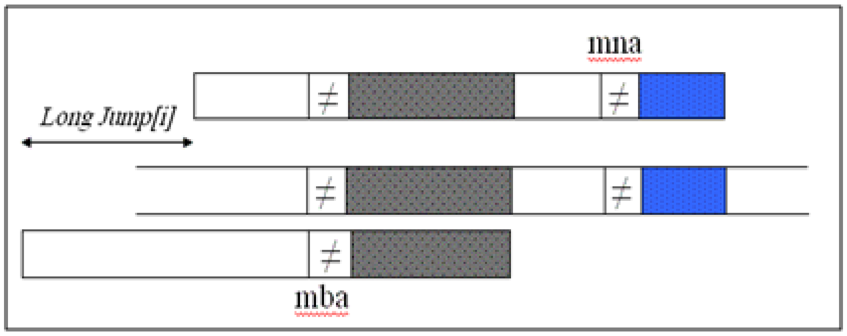


Figure 4. matched characters in two consequences searching attempts

Observation 2:
Just some part of the matched characters in before searching attempt and all of the matched characters in this searching attempts has aligned the pattern (see Figure 5). The next searching attempt must consider these two consequences in order to figure out the jumping length of the following shift.

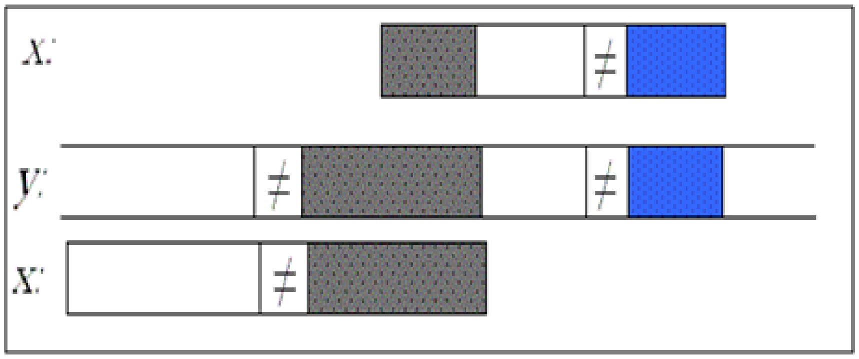


Figure 5. Just some part of the before searching attempt and are aligned the current searching attempt

Observation 3:
None of the matched characters in before searching attempt and all of the matched characters in this searching attempts has aligned the pattern (see Figure 6). The next searching attempt must consider just last attempt in order to figure out the jumping length of the following shift.

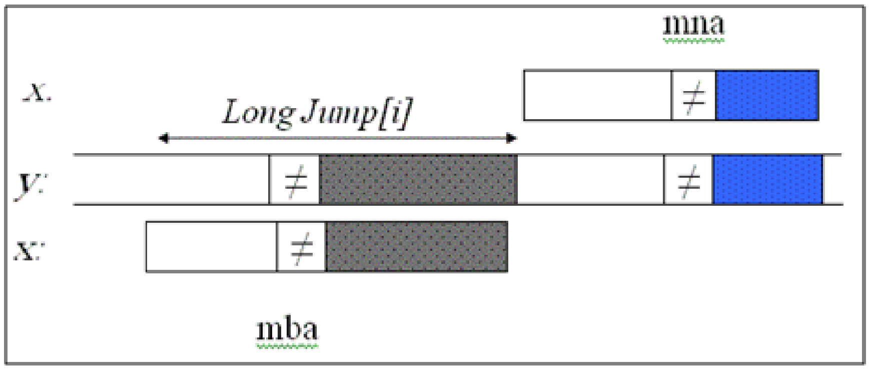


Figure 6. None of the before searching attempt and all of the matched characters in this searching attempts are aligned the pattern. mba is the before matching and mna is the current matching attempt

Computation of jumping amount for every mismatch position in the pattern has shown in Figure 7.

For simplicity, assume that we have the following text and pattern and we want to find all occurrences of the pattern in the text:

```
Text is   :  = “cdamdpbbcabd?????????”
Pattern is:=”bbc b bcabc”
```

Where (?) indicates an arbitrary character. In the searching phase, the BM algorithm will make (3 matching attempts to search for the pattern in the text, as follows:

First attempt (3 character comparisons, 2 matches and 1 Mismatch):

```
Text is   :  = “cdamdpbbcabd?????????”
Pattern is:= “bbc b bcabc”
```

In this paper, the matched characters have determined by blue color and the mismatched characters have demonstrated by red color.

By using BM algorithm, the jumping length(Long-Jump) is 3 characters(locations) as follows:

Second attempt (1 character comparisons, 0 matches and 1 Mismatch):

```
Text:= “ cdamdpbbcabd?????????”
Pattern is:=”b bcbbcabc”
```

The jumping length after these two matching attempts is 6 characters as follows:

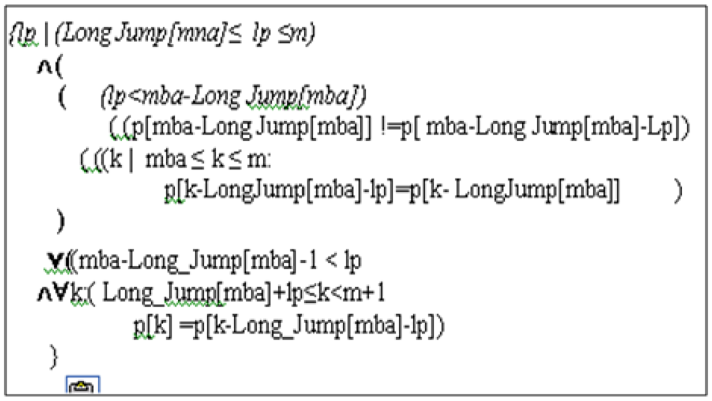
```
Text:=” cdamdpbbcabd?????????”
Pattern is:=     “bbcbbcabc”
```

Whereas by using proposed method, after one matching attempts, the jumping length is 9 characters as follows:

```
Text: = “ cdamdpbbcabd?????????”
Pattern is: =  “bbcbbcabc”
```

As we can see from this example, the proposed method improves significantly the jumping length is comparison with Boyer-Moore algorithm.

Computation of jumping length of our proposed method for every mismatch position along the pattern has demonstrated in Figure 7. In this table, P.Lenght is the length of pattern characters that has varied between 15 to 100.



```
{lp | (Long Jump[mna]≤ lp ≤m)
  ∧(
    (    (lp<mba-Long Jump[mba])
          ((p[mba-Long Jump[mba]] !=p[ mba-Long Jump[mba]-Lp])
       (((k |  mba ≤ k ≤ m:
                 p[k-LongJump[mba]-lp]=p[k- LongJump[mba]]       )
    )
   ∨((mba-Long_Jump[mba]-1 < lp
  ∧∀k:( Long_Jump[mba]+lp≤k<m+1
              p[k] =p[k-Long_Jump[mba]-lp])
   }
```

Figure 7. Computation of jumping amount for every mismatch position in the pattern. mba is the before matching attempt and mna is the current matching attempt

Pattern Length / Algorithm	P.Lenght=15	P.Lenght=20	P.Lenght=30	P.Lenght=40	P.Lenght=50	P.Lenght=100
Boyer Moore	39.3	313.6	171	322	167.3	269
Kunth-Morris- Pt	277	352	274	274.5	315	297
Proposed Algorithm	285	322	165	189	175	262

Tabel 1. Running Time evaluation of BM, KMP, and proposed method

We run our experiments using randomly generated patterns and text over a four characters alphabet. We slightly modified the algorithms to find all matches of the pattern.

Figure 8 shows a comparison between execution times of the KMP, BM, and proposed algorithms for each patterns sample of each pattern length from 15-100.

As we can see, It is clear that our proposed algorithm enhance the execution time of string matching as compared to the BM and KMP algorithms. This enhancement is calculated by considering the differences in execution times of the algorithms to search for several patterns samples as recorded in Table 1.

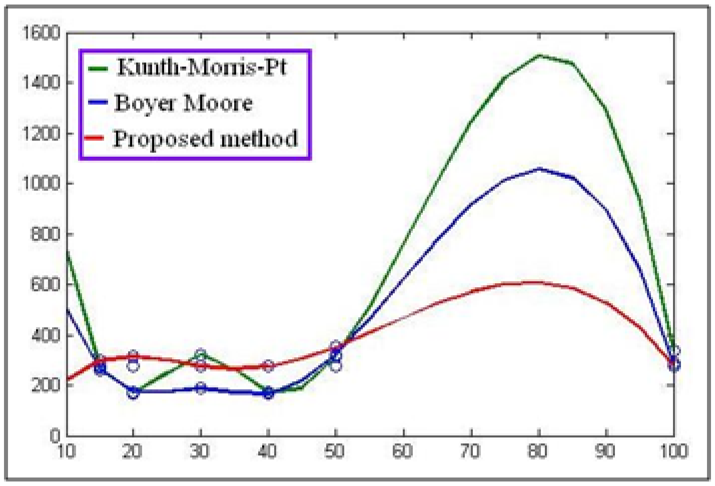


Figure 8. Comparison of different algorithms with the proposed method

4. Conclusion

Pattern Matching is a fundamental challenge in the field of computer science. Innovation and creativity in string matching can play an immense role for getting time efficient performance in various domains of computer science.

This study introduced a new exact pattern-matching algorithm based on the Boyer Moore algorithm. The proposed method uses matched partition in the previous step in order to avoid unnecessary comparisons. Afterwards, the proposed algorithm has been implemented and also has been compared with existing algorithms.

By analyzing string-matching algorithms, it can be concluded that Boyer-Moore string matching algorithms is more efficient than other proposed method such as KMP. Practice shows that BM Algorithm is fast in the case of larger alphabet. KMP decreases the time of searching compared to the Brute Force algorithm.

Since, the amount of jump in Boyer- Moore Algorithm is usually very small, In this paper, we have tried to solve this problem.

Our proposed method uses two last searching attempt of Boyer-Moore algorithm instead of one in Boyer-Moore algorithm. Altough time complexity of our proposed method in preprocessing step (computation of jumping length) is more than Boyer-Moore algorithm. But the seaching time of our proposed method is less then other method, in overal.

From the experimental results had shown in Table 1, it can be seen that our proposed algorithm gives better performance compared with some of the other popular methods like, KMP and Boyer Moore string search techniques. Also, as we can see from Figure. 7, the proposed algorithm gives very good performance related to the other popular methods.

References

[1] Besta, M. (2002). Mechanization of String-Preprocessing in Boyer Moore pattern Maching Algoritm, Detroit.

[2] Sheik, S. S. et al, (2004). A FAST Pattern Matching Algorithm, *J. Chem. Inf. Comput. Sci.*, 44 (4) 1251–1256.

[3] Lecroq. T. (1995). Experimental Results on String Matching Algorithms, SOFTWARE-PRACTICE AND EXPERIENCE, 727–765.

[4] Lecroq, T. (2000). New experimental results on exact string-matching, Rapport LIFAR 2000.03, Université de Rouen.

[5] Knuth, D. E., Morris, J. H. Pratt, ,V. R. (1974). Fast pattern matching in strings. TR CSp74-440, Stanford University, Calif.

[6] Boyer, R. S., Moore, J. S. (1977). A Fast String Searching Algorithm, Association for Computing Machinery.

[7] Cantone, D., Faro, S. (2003). Fast–search: a new efficient variant of the boyer– moore string matching algorithm. *Lecture Notes in Computer Science*, 2647:47–58.

[8] Tarhio, J. (1996). A sublinear algorithm for two-dimensional string matching. Pattern Recognition Letters, 17:833–838, July.

[9] Baeza-Yates, R. A. (1989) . String searching algorithms revisited. Lecture Notes in Computer Science, 382. 75–96.

[10] Wu, Y. C., Yang, J. C., Lee, Y. S. (2007). A weighted string pattern matching-based passage rankinjg algorithm for video question answering, *J. Expert Systems*, 2588-2600.

[11] Rami Mansi, H., Jehad Odeh, Q. (2009). On Improving the Naïve String Matching Algorithm, *Asian Journal of Information Technology,* 8. 14-23.

[12] Amintoosi, M, Fathy, M., Monsefi, R. (2006). Using pattern matching for tiling and packing problems, *Eur. J. Operat. Res.*, 183 (3): 950-960.

[13] Devaki-Paul. (2011). Novel Devaki-Paul Algorithm for Multiple Pattern Matching , *International Journal of Computer Applications (0975 – 8887) 13(3) , January* .

[14] A- Ning Du., Bin- Xingfang., Xiao-Chun Yun., Ming-Zenghu., Xiu- R ong Zheng. (2003). Comparision of String Matching Algorithms: An Aid To Information Content Security *In* : Proceedings of the Second International Conference on Mache Learmng and Cybernetics, xi, 2-5 November, 2996-3001.