Fastest Approach to Exact Pattern Matching

Iftikhar Hussain, Imran Ali, Muhammad Zubair and Nazarat Bibi
Department of Computing and Technology
Iqra University, Islamabad Campus
Islamabad, Pakistan
Iftikhar.iftikhar786@gmail.com, imran.ali837@gmail.com, mzubair_sher@yahoo.com and nazarat.zubair84@gmail.com

Abstract— **This research, presents an improved version of Bidirectional (BD) [20] algorithm to solve the problem of exact pattern matching. Fastest-Bidirectional (FBD) exact pattern matching algorithm introduced a new idea of scanning partial text window (PTW) as well with the pattern by taking Berry-Ravindran (BR) consecutive characters to take decision of moving pattern to the right of PTW. FBD algorithm compares the characters of pattern to selected text window from both sides simultaneously as BD. The time complexity of preprocessing phase of FBD algorithm is *O(m+ |∑|)* and searching phase takes *O(mn/2)*.**

Keywords— Scanning text window, exact pattern matching, Fastest-Bidirectional, Berry Ravindran.

I. INTRODUCTION

Exact string matching algorithms are an important class of the string searching algorithms that try to find all possible occurrences of pattern *P* of size *m* from the text string *T* of size *n*. Researchers have been focused this area of research and many technique and algorithms have been proposed and designed to solve this problem. Exact String matching algorithms are widely used in bibliographic search, question answering application, DNA pattern matching, operating systems, voice recognition, imaging processing, search engine, text processing applications and information retrieval from databases [3, 4].

This research presents an improved version of Bidirectional (BD) [20] exact pattern matching algorithm based on window sliding method. Proposed Fastest-Bidirectional (FBD) exact pattern matching algorithm used Berry Ravindran's first two consecutive characters right to PTW to scan pattern from right to left. In addition FBD uses first two consecutive characters of pattern to scan PTW from left to right.

Literature review of previous exact string matching algorithms used to complete this research. After the publications of Knuth-Morris-Pratt and Boyer-Moore exact pattern matching algorithms, so for there have hundreds of papers published related to exact pattern matching [19]. According to literature survey, all the authors focus to reduce the **number of character comparisons** as [7] and **processing time** as [13, 8, 9] in both worst/average cases. In this paper we compare Fastest-Bidirectional algorithm's results with Bad Character Boyer-Moore, BM Horspool, Quick Search, Berry Ravindran, BM Smith, Raita and Bidirectional exact pattern matching algorithms which considered efficient in terms of number of character comparisons and attempts take to complete the processing of selected text.

This paper presents a brief literature review of some existing exact string matching algorithms in Section II. Section III describes the basic concept and working of Fastest-Bidirectional algorithm with brief example. Implementation of FBD algorithm presented in Section IV. Then we compare the FBD algorithm with some existing algorithms in terms of number of character comparisons and attempts take to achieve the task of searching. In Section V, we present experiments that compare the FBD algorithm with existing algorithms. Finally, in Section VI, we draw conclusions from the experiments.

II. LITERATURE SURVEY

Brute force (BF) [10] or Naïve algorithm is the logical place to begin the review of exact string matching algorithms. It compares a given pattern with all substrings of the given text in any case of a complete match or a mismatch. It has no preprocessing phase and did not require extra space. The time complexity of the searching phase of brute force algorithm is *O(mn)*.

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. It compares the characters from left to right of the pattern. In case of match or mismatch it uses the previous knowledge of comparisons to compute the next position of the pattern with the text. The time complexity of preprocessing phase is *O(m)* and of searching phase is *O(nm)*.

Boyer-Moore (BM) [1] algorithm published in 1977 and at that time it considered as the most efficient string matching 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. In case of a match or mismatch, it used two shifting rules to shift the pattern right. The time and space complexity of preprocessing phase is *O(m+|∑|)* and the worst case running time of searching phase is *O(nm + |∑|)*. The best case of Boyer-Moore algorithm is *O(n/m)*.

Boyer-Moore Horspool (BMH) [8] did not use the shifting heuristics as Boyer-Moore algorithm used. It used only the occurrence heuristic to maximize the length of the shifts for text characters corresponding to right most character of the pattern. It's preprocessing time complexity is *O(m+|∑|)* and searching time complexity is *O(mn)*.

Quick Search (QS) [9] algorithm perform comparisons from left to right order, it's shifting criteria is by looking at one character right to the pattern and by applying bad character

978-1-4244-8003-6/10/$26.00 ©2010 IEEE

shifting rule. The worst case time complexity of QS is same as Horspool algorithm but it can take more steps in practice.

Boyer-Moore Smith (MBS) [18] noticed that by computing the BMH shift; sometimes maximize the shifts than QS shifts. It uses bad character shifting rule of BMH and QS bad character rule to shift pattern. It's time-complexity in preprocessing phase is $O(m+|\sum|)$ and in searching is $O(mn)$.

Turbo Boyer Moore (TBM) [13] is variation of the Boyer-Moore algorithm, which remembers the substring of the text string which matched with suffix of pattern during last comparisons. It does not compare the matched substring again; it just compares other characters of the pattern with text string.

Two Way algorithm (TW) [16] proposed by Crochemore and Rytter in 2002. The Two Way algorithm uses an idea related to the short maximal suffix of the pattern to calculate the shifting lengths of pattern in text string. The Two Way algorithm's time complexity with short maximal suffix is $O(n)$.

Berry Ravindran (BR) [17] algorithm proposed by Berry and Ravindran in 1999, it performs shifts by using bad character shifting rule for two consecutive characters to the right of the partial text window of text string. The preprocessing time complexity is $O(m+(|\sum|)^2)$ and the searching time complexity is $O(mn)$.

In Reverse Colussi (RC) [14] algorithm comparisons are done in specific order given by the preprocessed phase. The time complexity of preprocessing phase is $O(m^2)$ and searching phase is $O(n)$.

III. FASTEST-BIDIRECTIONAL EXACT PATTERN MATCHING ALGORITHM

Fastest-Bidirectional exact pattern matching algorithm compares a given pattern character wise from both sides simultaneously as Bidirectional algorithm. The difference BD and FBD is that, in case of mismatch or a complete match of the pattern, FBD used for first two consecutive characters right to PTW to scan pattern from right to left. In addition FBD uses first two consecutive characters of pattern to scan PTW from left to right. If both pairs of consecutive characters found on equal shifts then align pattern to the position where the pairs of characters found otherwise, shift whole pattern to the right of the partial text window with maximum jump $m+2$. A complete match will be found when left pointer cross right pointer at middle of the pattern.

Fastest-Bidirectional algorithm has four cases to take decision to shift pattern to right of partial text window. Suppose $T[1…n]$ is the text string and $P[1…m]$ is the pattern and we compare pattern $P[1…m]$ with the selected text window $T[i…i+m-1]$ from both sides of the pattern simultaneously. If mismatch cause by any pointer either right or left at any position of the pattern then scan pattern $P[m…1]$ for first two consecutive characters $T[i+m]$ and $T[i+m+1]$ right to PTW and scan partial text window $T[i … i+m-1]$ for first two consecutive $P[1]$ and $P[2]$ characters of pattern;

Case 1: If $T[i+m]$ is equal to $P[m]$ and leftmost consecutive characters of pattern $P[0]$, $P[1]$ are found at $T[i+1]$ and $T[i+2]$ on equal shifts as in pattern as well as in partial text window of text string then align pattern to the position where the consecutive characters found on equal distance. The pattern is shifted by 1 position to the right of partial text window as shown in Fig. 1.

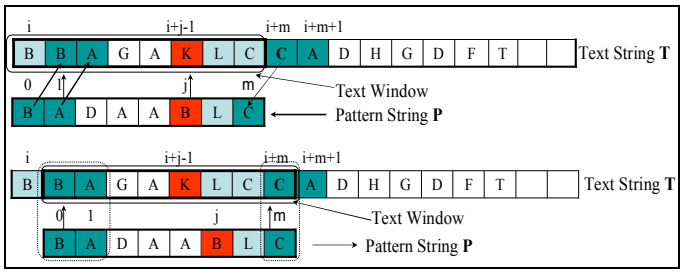


Figure 1: Pattern is shifted by 1 position to the right of PTW.

Case 2: If both pairs $P[0]$, $P[1]$ and $T[i+m]$, $T[i+m+1]$ of consecutive characters founds on equal shifts in pattern as well as in PTW then align pattern to the position where the consecutive characters found on equal distance as in Fig. 2.

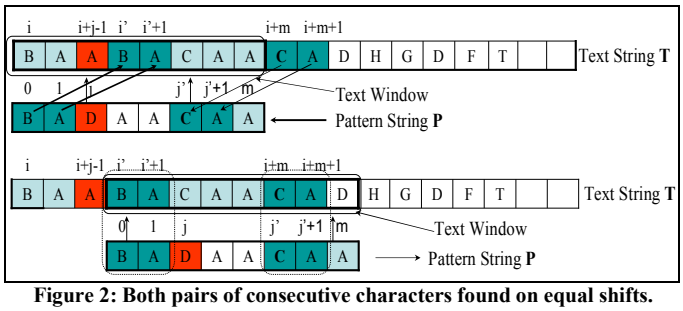


Figure 2: Both pairs of consecutive characters found on equal shifts.

Case 3: If character $T[i+m+1]$ of PTW is equal to the leftmost character of pattern $P[0]$ then align pattern to the position where both characters aligns. The pattern is shifted by $m+1$ positions to the right of PTW as shown in Fig. 3.

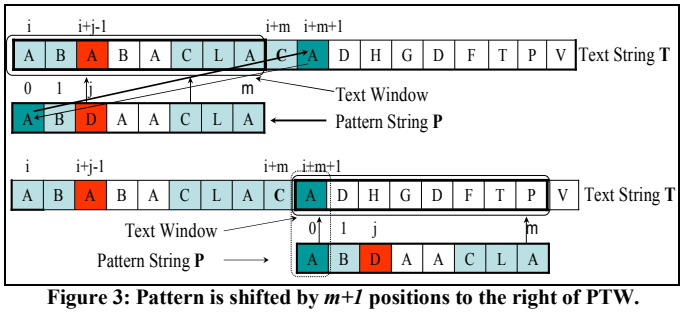


Figure 3: Pattern is shifted by *m+1* positions to the right of PTW.

Case 4: Otherwise; if both pairs of consecutive characters did not find on equal shifts or at least one character is not present then pattern is shifted by $m+2$ positions as shown in Fig. 4.

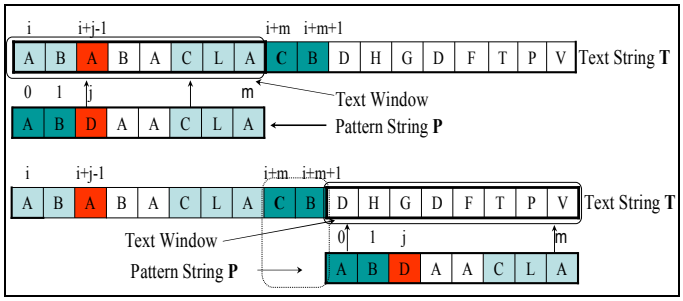
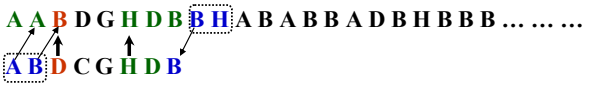


Figure 4: Pattern is shifted by *m+2* positions to the right of PTW.

Example of Fastest-Bidirectional Algorithm

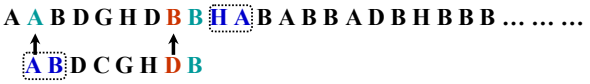
Here is a brief example of Fastest-Bidirectional exact pattern matching algorithm, where T="AABDGHDBBHAB ABBADBHBBBDABHABACGABAADBH" and P="ABDC GHDB".

Attempt 1:



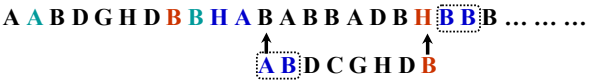
In first attempt FBD algorithm takes 6 comparisons, after mismatch the preprocessing phase scans for the characters *'BH'* of right to the partial text window in pattern and leftmost character *'AB'* of pattern in partial text window. Here case 1 applied because *'AB'* of pattern and *'B'* of right to PTW are found in PTW and pattern respectively on equal distance.

Attempt 2:



Mismatch occurred at third comparison by right pointer in second attempt, after scanning pattern and partial text window both pairs of consecutive characters did not find at same shifting positions. So, here Case 4 should be applied to perform shift by considering relevant characters.

Attempt 3:



Mismatch caused by right pointer at first comparison in third attempt and after preprocessing phase both characters did not find at same shifts so case 4 should be applied here.

Attempt 4:

Here Case 2 should be applied because both pairs of consecutive (*AB* of pattern and *CG* of PTW) characters were found on same shifts in preprocessing phase. In fourth attempt, it takes one comparison.

Attempt 5:

Here, in this attempt FBD takes one comparison.

This example takes overall 12 comparisons in 5 attempts by using FBD exact pattern matching algorithm.

IV. IMPLEMENTATION OF FBD ALGORITHM

1) *Preprocessing Phase*

Preprocessing phase of FBD algorithm finds occurrences of first two consecutive characters to the right of PTW in pattern and first two consecutive characters of pattern in PTW. This phase helps to take decision to move pattern to the right of the partial text window of text string. The pseudocode of preprocessing phase of FBD algorithm in Fig. 5 shows that the preprocess function pass a pattern *P[1 … m]*, partial text window *T[i … i+m-1]* , consecutive characters *T[i+m]* and *T[i+m+1]* right to PTW and starting index of partial text window *'i'* as input and return shifts's length to move pattern to the right of partial text window of text string. For loop, of preprocessing phase scans pattern from last character to leftmost character of pattern by decrementing the indexes of pattern. It also scan partial text window from second to two characters right to PTW by incrementing index *'i'*.

Inside for loop, first check for first character *T[i+m]* from two consecutive characters right to PTW in pattern and second character *P[1]* from consecutive characters of pattern in PTW. If both characters found in pattern and PTW respectively, then check for first character *P[0]* of pattern at first loop-counter then return *shift's length*. Then, it check for second character *T[i+m+1]* of consecutive characters of PTW and first character *P[0]* of first two consecutive characters of pattern. If both characters found then return *shift's length.* At last loop-counter, if second *T[i+m+1]* character of first two consecutive characters right to the PTW is found at leftmost position of the pattern then return *shift's length*.

```
Input:  Pattern, partial text window T[i…i+m+1] and
starting index i of partial text window of text string.
Output: Return shift's length.
1   k ← 1, shift ← P.length+2;
2   for j ← P.length-1 to 0  do
3       if P[j] = T[i+m] and T[i+2] = P[1] then
4           if j = P.length-1 and T[i+1] = P[0] then
5               shift ← k;        Break;
6           else if T[i+1]=P[0] and P[j+1]=T[i+m+1]
7               shift ← k;        Break;
8       else if j = 0 and P[j] = T[i+m+1] then
9           shift ← k+1;          Break;
10       k ← k+1, i ← i+1;
11  return shift;
```

Figure 5: Pseudocode of preprocessing phase of FBD.

2) *Searching Phase*

Character wise comparison will be performed between the pattern and the selected text window of the text string. Fig. 6 shows the pseudocode of searching phase of FBD in which it takes a text string of length 'n' and a pattern of length 'm' as input and display all occurrences of pattern from text string as output. The external while loop is used to shift the pattern to the right of partial text window.

Two pointers (left and right) are used to compare pattern with the selected text window within the second while loop as BD algorithm. A complete match will be found, if left pointer cross right pointer at middle of the pattern then preprocess function will be called to calculate the length of the shift. Else, if mismatch caused by left or right pointer, then preprocess function will be executed to calculate the shift's length to move pattern to right of partial text window where next attempt will be performed.

```
Input: Text string T of length n and Pattern P of length m.
Output: All occurrences of pattern in text string.
1   n ← T.length, m ← P.length;
2   i ← 0, j ← 0;
3   while i ≤ n-m          do
4       left ← 0, right ← m-1;
5       while j < (m+1)/2  do
6           if P[right]=T[i+right] and P[left]=T[i+left]
7               if j = (m+1)/2 -1 then
8               "Pattern matched at:", i;
9               i←i + preprocessing (P, T[i…i+m+1], i);
10          left ← left+1;
11          right ← right-1;
12          Else begin
13              i←i + preprocessing (P, T[i…i+m+1], i);
14              Break Inner-While;
```

Figure 6: Pseudocode of searching phase of FBD.

a. Analysis of Bidirectional Algorithm

1) Analysis of Preprocessing Phase

The worst case time complexity of preprocessing phase of FBD algorithm is *O(m)*, because only one loop is used to scan the pattern and partial text window to find the rightmost character of partial text window and leftmost character of pattern.

2) Analysis of Searching Phase

The inner while loop of searching phase of FBD algorithm runs at most '*m/2*' times so, its worst case time-complexity is *O(m/2)* because two pointers are used. The worst case time-complexity to shift pattern to right of the PTW is *O(n)* because the external while loop runs *'n'* times at most. The total time complexity of searching phase is *O(m/2).O(n)*, because the inner loop runs within external while loop. It can be written as *O(mn/2)*.

Fastest-Bidirectional algorithm requires *O(2m)* extra memory space in worst case to execute in addition with the text and pattern string.

V. RESULTS AND DISCUSSIONS

The efficiency of Fastest-Bidirectional exact pattern matching algorithm is measured and compared with existing; Bad Character Boyer-Moore, BM Horspool, Quick Search, Berry Ravindran, BM Smith, Raita and Bidirectional exact pattern matching algorithms. Fastest-Bidirectional algorithm is compared with existing algorithms by using characters **compare-base** and **attempts/shifts-base** comparison. A text string of length *60,000* of four *{A, C, G, T}* random characters and part of text of different lengths *{6, 12, 18, 24, 30, 36, 42, 48, 54, and 60}* is used as pattern for the experiments. The experiments calculate the number of characters comparison and attempts/shifts each algorithm takes to perform the task; the results are shown by using graphs in figs. 7 and 8.

a. Attempts Base Comparison

Total numbers of attempts taken by each algorithm using different pattern lengths are shown in graphical form in Fig. 6. As results in the graph shows that Fastest-Bidirectional algorithm takes minimum shifts as compare to other Bad Character Boyer-Moore, BM Horspool, Quick Search, Berry Ravindran, BM Smith, Raita and Bidirectional algorithms. Results also show that on short and long pattern's lengths Fastest-Bidirectional algorithm is efficient than existing algorithms.

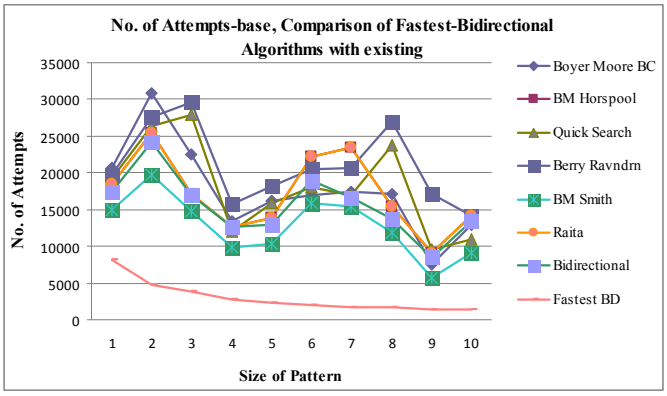


Figure 7: Shift Base Comparison.

b. No. of Characters compare base Comparison

Total numbers of characters comparisons taken by each algorithm using different pattern lengths are shown in graphical form in Fig. 8.

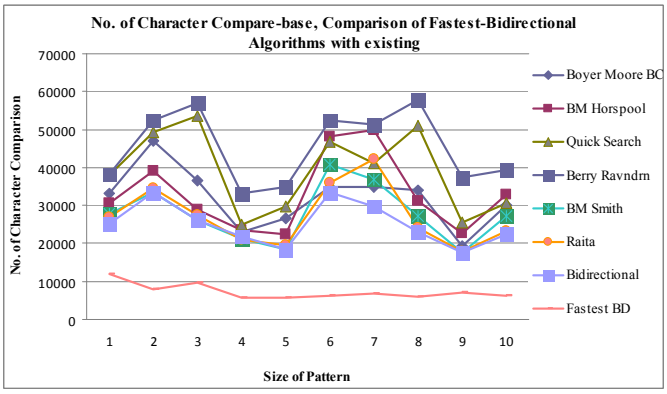


Figure 8: No. of characters compare base Comparison.

Results in the Figure shows that Fastest-Bidirectional algorithm takes less number of character comparisons than the existing algorithms when the pattern's length short as well as long.

According to both comparing criterion Fastest-Bidirectional exact pattern matching algorithm is quite efficient than the existing algorithms.

VI. CONCLUSION

This research presents a new version of Bidirectional exact pattern matching algorithm. Fastest-Bidirectional (FBD) exact pattern matching algorithm introduced a new idea of

scanning partial text window (PTW) as well with the pattern by taking Berry-Ravindran (BR) consecutive characters to take decision of moving pattern to the right of PTW. FBD algorithm compares the characters of pattern to selected text window from both sides simultaneously as BD. The worst case time-complexity of FBD is *O(mn/2)* in searching phase which asymptotically represented as *O(mn)* and *O(m)* in preprocessing phase. Comparison results show that the FBD algorithm is quite efficient than the existing algorithms, when the pattern length is short as well as on long pattern's lengths.

REFERENCES

[1] R.S. Boyer, J.S. Moore, "A fast string searching algorithm," *Communication of the ACM*, Vol. 20, No. 10, 1977, pp.762–772.

[2] Knuth, D., Morris, J. H., Pratt, V., "Fast pattern matching in strings," SIAM Journal on Computing, Vol. 6, No. 2, doi: 10.1137/0206024, 1977, pp.323–350.

[3] Rami H. Mansi, and Jehad Q. Odeh, "On Improving the Naïve String Matching Algorithm," *Asian Journal of Information Technology*, Vol. 8, No. 1, ISSN 1682-3915, 2009, pp. 14-23.

[4] Ziad A.A. Alqadi, Musbah Aqel, & Ibrahiem M. M. El Emary, "Multiple Skip Multiple Pattern Matching Algorithm," *IAENG International Journal of Computer Science*, Vol. 34, No. 2, IJCS_34_2_03, 2007.

[5] Ababneh Mohammad, Oqeili Saleh and Rawan A. Abdeen, "Occurrences Algorithm for String Searching Based on Brute-force Algorithm," *Journal of Computer Science*, Vol. 2, No. 1, ISSN 1549-3636, 2006, pp.82-85.

[6] A. Apostolic and R. Giancarlo, "The Boyer-Moore-Galil string searching strategies revisited," *SIAM J. Computer.* Vol. 15, No. l, 1986, pp.98–105.

[7] L. Colussi, Z. Galil, and R. Giancarlo, 'On the exact complexity of string matching,' *31st Symposium an Foundations of Computer Science* I, IEEE (October 22-24 1990), pp.135–143.

[8] R. N. Horspool, "Practical fast searching in strings," *Software—Practice and Experience,* Vol. 10, No. 3, 1980, 501–506.

[9] Sunday, D.M., "A very fast substring search algorithm," *Communications of the ACM*, Vol. 33, No. 8, 1990, pp. 132-142.

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, Chapter 34, MIT Press, 1990, pp 853-885.

[11] Karp, R.M., Rabin, M.O., "Efficient randomized pattern matching algorithms," *IBM Journal on Research Development*, Vol. 31, No. 2, 1987, pp. 249-260.

[12] Apostolico, A. Crochemore, M., "Optimal canonization of all substrings of a string," *Information and Computation*, Vol. 95, No. 1, 1991, pp.76-95.

[13] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski. W., Rytter, W., "Speeding up two string matching algorithms," *Algorithmica*, Vol. 12, No. 4/5, 1994, pp.247-267.

[14] Colussi, L., "Fastest pattern matching in strings," *Journal of Algorithms*, Vol. 16, No. 2, 1994, pp.163-189.

[15] Hume, A., Sunday, D. M., "Fast string searching," *Software Practice & Experience*, Vol. 21, No. 11, 1991, pp.1221-1248.

[16] Crochemore, M. and Rytter, W., "Jewels of Stringology," *World Scientific, Singapore*, 2002.

[17] Berry, T. Ravindran, S., "A fast string matching algorithm and experimental results, in proceeding of the Prague Stringology," *Club Workshop-99*, Collaborative report DC-99-5, Czech Technical University, Prague, Czech Republic, 1999, pp.16-26.

[18] Smith, P.D., "Experiments with a very fast substring search algorithm," *Software-Practice and Experience*, Vol. 21, No. 10, pp.1065-1074.

[19] Frantisek Franek, Christopher G. Jennings, W. F. Smyth, "A simple fast hybrid matching algorithm," *Journal of Discrete Algorithms*, Vol. 5, 2007, pp. 682-695.

[20] Iftikhar Hussain, Muhammad Zubair, Jamil Ahmed and Junaid Zaffar, "Bidirectional Exact Pattern Matching Algorithm," *TCSET'2010*, Feb. 2010, pp. 295(*Abstract*).