# An Application of Pattern Matching for Motif Identification

**K. K. Senapati** *kksenapati@bitmesra.ac.in*
*Department of Computer Science & Engineering.*
*Birla Institute of Technology, Mesra*
*Ranchi, 835215, India*

**D. R. Das Adhikary** *dibya21@gmail.com*
*Department of Computer Science & Engineering.*
*Birla Institute of Technology, Mesra*
*Ranchi, 835215, India*

**G. Sahoo** *gsahoo@bitmesra.ac.in*
*Department of Information Technology.*
*Birla Institute of Technology, Mesra*
*Ranchi, 835215, India*

**Abstract**

Pattern matching is one of the central and most widely studied problem in theoretical computer science. Solutions to the problem play an important role in many areas of science and information processing. Its performance has great impact on many applications including database query, text processing and DNA sequence analysis. In general Pattern matching algorithms are based on the shift value, the direction of the sliding window and the order in which comparisons are made. The performance of the algorithms can be enhanced to a great extent by a larger shift value and less number of comparison to get the shift value. In this paper we proposed an algorithm, for finding motif in DNA sequence. The algorithm is based on preprocessing of the pattern string(motif) by considering four consecutive nucleotides of the DNA that immediately follow the aligned pattern window in an event of mismatch between pattern(motif) and DNA sequence .Theoretically, we found the proposed algorithms work efficiently for motif identification.

**Key words:** Pattern Matching, Pattern String, Motif Finding, Motif Identification, Gene Identification, Preprocessing.

## 1. INTRODUCTION

Pattern matching consists in finding one, or more generally, all the occurrences of a given query string (pattern) from a possibly very large text is an old and fundamental problem in computer science. It emerges in applications ranging from text processing and music retrieval to bioinformatics. This task, collectively known as string matching, has several different variations. The most natural and important of these is exact string matching, in which, like the name suggests, one wish to find only occurrences that are exactly identical to the pattern string, which is the focus of our work. The field of approximate string matching, on the other hand find occurrences that are similar to the pattern string.

Given a text array, T [1 . . . n], of n character and a pattern array, P [1 . . . m], of m characters. The problem is to find an integer s, called valid shift where $0 \leq s \leq n - m$ and $T[s+1 \ldots s+m] = P[1 \ldots m]$. In other words, to find whether P in T i.e., where P is a substring of T. The elements of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B ... Z, a, b . . . z} [1] [2].

All pattern-matching algorithms scan the text with the help of a window, which is equal to the length of the pattern. The first process is to align the left ends of the window and the text, and then compare the corresponding characters of the window and the pattern. After a whole match or a mismatch of the pattern, the text window is shifted in the forward direction until the right end of the window reaches the end of the text. The algorithms vary in the order in which character comparisons are made and the distance by which the window is shifted on the text after each attempt. Many pattern matching algorithms

are available with their own merits and demerits based on the pattern length, periodicity and alphabet set. An efficient way is to move the pattern on the text using the best shift value. To this end, several algorithms have been proposed to get a better shift value, for example: Boyer–Moore [3], Quick Search [4], Karp-Rabin [5], Raita [6] and Berry–Ravindran [7]. The efficiency of an algorithm lies in two phases: pre-processing phase and the searching phase. Effective searching phase can be established by altering the order of comparison of characters in each attempt and by choosing an optimum shift value that allows a maximum skip on the text [8]. The difference between various algorithms is mainly due to the shifting procedure and the speed at which a mismatch is detected.

The rest of the paper is organized as follows. Section 2 gives the review of several efficient algorithms in practice. Section 3 describes the proposed algorithm in detail. Section 4 is about complexity analysis of the proposed algorithm. In section 5, the experimental results of comparison between proposed algorithm and other compared algorithm are given. And section 6 is the conclusion.

## 2. PREVIOUS WORK

Many promising data structures and algorithms discovered by theoretical community are never implemented or tested at all. This is because the actual performance of the algorithm is not analyzed as only the working of the algorithm in practice is taken care. There have been several pattern matching algorithms designed in the literature till date as discussed below.

In the naive or Brute Force technique [1] [2] the string to be searched is aligned to the left end of the text and each pattern character is compared with the corresponding text character. In this process if a mismatch occurs or the pattern character exhausts the pattern is shifted by one unit toward right. The search is again resumed from the start of the pattern until the text is exhausted or match is found. Naturally the number of comparisons being performed is very more as each time the pattern string is shifted right by only one unit towards right. The worst case comparison of the algorithm is O (mn). The number of comparisons can be reduced if we can move the pattern string by more than one unit. This was the idea of KMP algorithm [1][2][9][10]. The KMP algorithm compares the pattern from left to right with the text just as BF algorithm. When a mismatch occurs the KMP algorithm moves the pattern to the right by maintaining the longest overlap of a prefix of the pattern with a suffix of a part of the text that matched the pattern so far. The KMP algorithm does almost 2n text comparisons and the worst case complexity of the algorithm is O (m+n).

Boyer-Moore algorithm [2][3] differs from other algorithms in one feature. Instead of comparing the pattern characters from left to right the comparison is done from the right towards left by starting comparison from the rightmost character of the pattern. In case of a mismatch it uses two functions last occurrence function and good suffix function. If the text character does not exist in the pattern then the last occurrence function returns m where m is the length of the pattern string. So the maximum shift possible was m. The worst case running time of the algorithm is O (mn).

Quick Search (QS) [2][4] 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 shifting rule. The worst case time complexity of QS is same as Horspool algorithm but it can take more steps in practice.

Berry Ravindran (BR) [2][7][11] algorithm proposed by Berry and Ravindran, 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+ (|Σ|) 2) and the searching time complexity is O (mn).

In this paper the idea is to reduce the number of comparisons being performed by obtaining as much shift as possible. A shift value of more than length of the pattern is obtained when the pattern gets mismatch with the text for one instance.

## 3. PROPOSED ALGORITHM

The proposed algorithms works in two phases, the preprocessing phase and the searching phase.

### 3.1. Preprocessing phase

In Berry-Ravindran algorithm, two consecutive characters immediately to the right of the pattern window are considered in the function brBc [7]. But in the proposed algorithm, four consecutive characters immediately to the right of window are considered. Initially, the indexes of the four consecutive characters in the text string from the left are (m), (m+1), (m+2) and (m+3) for a, b, c and d respectively. The MBrBc (a, b, c, d) of the algorithm consists in computing for each four characters a, b, c, d for all a, b, c, d , in the pattern.

The MBrBc(a, b, c, d) is defined as follows:

$$\text{MBrBc[a,b,c,d]} = \min \begin{cases} 1 & \text{if } p[m-1] = a \\ 2 & \text{if } p[m-2]p[m-1] = ab \\ 3 & \text{if } p[m-3]p[m-2]p[m-1] = abc \\ m-i+1 & \text{if } p[i]p[i+1]p[i+2]p[i+3] = abcd \\ m+1 & \text{if } p[0]p[1]p[2] = bcd \\ m+2 & \text{if } p[0]p[1] = cd \\ m+3 & \text{if } p[0] = d \\ m+4 & \text{otherwise} \end{cases}$$

This function finds the position of the right most occurrence of 'abcd' in the pattern and computes shift value so that the 'abcd' in the pattern and the 'abcd' if any in the text immediately to the right of the window are aligned. If 'abcd' does not exist in the pattern or a portion i.e. 'a' 'ab' 'abc' 'bcd' 'cd''d' presents then other cases that are considered.

We improve the algorithm in programmatically by using a one dimensional array called shift array (SA). The shift array is use for storing the shift value for all four character permutation of the given pattern. For any four given character we generate a unique number and use this unique number as index to store the shift value for that four character. For any "abcd" the value is cal calculated like this:

index=(a*1000)+( b*100)+( c*10)+d
SA [index] =shift value

We use this technique to store the shift value, which is a very efficient than other possible technique.

### 3.2. Searching Phase

In this proposed algorithm, after each attempt the window is shifted to the right using the shift value computed for the four consecutive characters immediately right of the window. MBrBc function calculates the shift value based on the right most occurrences of four consecutive characters, say abcd, which is immediately to the right of the window. The probability occurrence of four consecutive characters, abcd, in the pattern as compared to that of ab is less. Thus MBrBc always provides a better shift than brBc (Berry-Ravindran Bad character function) [12].

The searching phase of the algorithm works in the following way.

**Step1**: Compare the characters of the windows with the corresponding text characters from left as well as right [13][14]. If there is a mismatch during comparison, the algorithm goes to step2, otherwise the comparison process continues until a complete match is found. The algorithm stops and displays the corresponding position of the Pattern on the text string. If we search for all the pattern occurrences in the text string, the algorithm continues to step2.

**Step2**: In this step, we use the shift values from the next arrays depending on the four text characters placed immediately after the pattern window. The window is shifted to the correct positions based on the shift value and the algorithm goes to step 1, this process continues till the end of the string. The searching phase of the algorithm works in the following way. Suppose we have a text string T[O . . . n-l] and pattern P[O . . . m-l] and starts searching of P in T. The algorithm compares the pattern with selected text window from both (right and left) sides simultaneously. In case of match or mismatch MBrBc shift value is used to

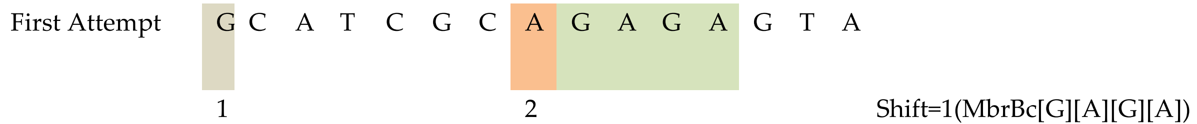
shift the window to the right. This procedure is repeated until the window is placed beyond n-m+1, that is the last character of the pattern placed beyond the last character of the text.

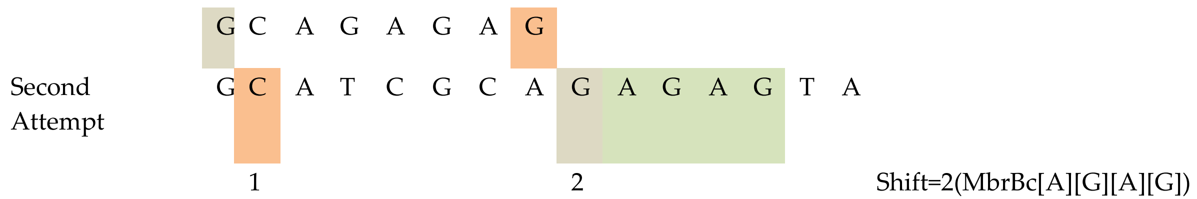
The Searching Phase of Motif Finding Algorithm

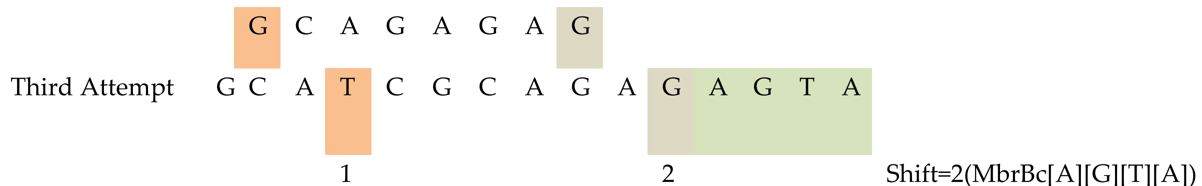
```
1.  Search_Motif(T,P)
2.       n←length[T]
3.       m←length[p]
4.       i←0
5.       s←n-m
6.       while( i <= s) do
7.            left←0
8.            right←m-1
9.            while(left<=right)do
10.                if (P[left]==T[i+left]&& P[right]==T[i+right])
11.                     left←left+1
12.                     right←right-1
13.                else
14.                     break
15.           end while
16.           if(left>right)
17.                print "pattern occurs at" T[i]to T[i+m-1]
18.           i=i+ MbrBc(T[i+m],T[i+m+1],T[i+m+2],T[i+m+3])
19.      end while
20. end procedur
```

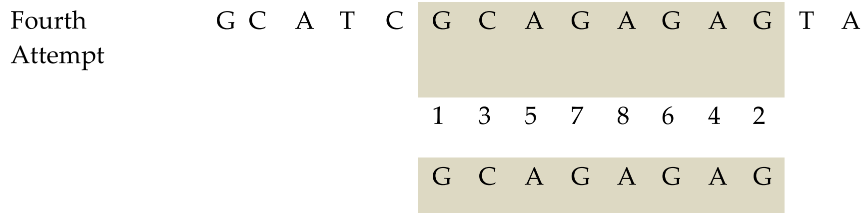
### 3.3 Working Example

Consider the text and pattern string as shown below where text length n=15 and pattern length m=8
Text (T) = G C A T C G C A G A G A G T A
Pattern (P) = G C A G A G A G, m = 8






**First Attempt:** In the first attempt, we align the sliding window with the text from the left. In this case, a match occurs between text character (G) and pattern character (G)in the left side of window and a mismatch occurs between text character (A) and pattern character (G)in the right side of window. Therefore we take the immediate four characters following the text as (G,A,G, and A). We find according to MBrBc function. That the shift=1, therefore the window is shifted to the right 1 step.

**Second Attempt:** In the second attempt, we align the sliding window with the text from the left. In this case, a mismatch occurs between text character (C) and pattern character (G)in the left side of window and a match occurs between text character (G) and pattern character (G)in the right side of window. Therefore we take the immediate four characters following the text as (A, G, A and G). We find according to MBrBc function. That the shift=2, therefore the window is shifted to the right 2 step.

**Third Attempt:** In the third attempt, we align the sliding window with the text from the left. In this case, a mismatch occurs between text character (T) and pattern character (G)in the left side of window and a match occurs between text character (G) and pattern character (G)in the right side of window. Therefore we take the immediate four characters following the text as (A, G, T and A). We find according to MBrBc function. That the shift=2, therefore the window is shifted to the right 2 step.

**Fourth Attempt:** In the fourth attempt, we align the sliding window with the text from the left after shift. In this case all the character of the pattern matches with the text. So match performed. Next the window is moved for the next pattern in the text string.

## 4. ANALYSIS

The space complexity is O((m/2+1)).where m is the pattern length..The pre-process time complexity is O(m+|∑|^4). The worst case time complexity is O(nm). The worst case occurs when at each attempt; all the compared characters and the text are matched and at the same time the shift value is equal to 1 i.e. Last character of the pattern is equal to the first character present next to the window.
Example:
Text (T): AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Pattern (P): AAAAA

The best case time complexity is O (n/ (m+2)).The best case occurs when at each attempt; in the first comparison a mismatch is found and at the same time the shift value is m+2.
Example:
Text (T): AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Pattern (P): SSSSS

## 5. EXPERIMENTAL EVALUATIONS

To assess the performance of my algorithm, we considered all the well-known algorithms stated before for comparison with the proposed algorithm. The algorithms are compared with test cases and their corresponding results are discussed.

### 5.1 Environment

In the experiments we used a PC with Intel(R) Core(TM) 2 Duo 2.10 and 1.96 GHz processor, with 2GB of RAM. The host operating system is windows Xp. The source codes were compiled using the “gcc” compiler without optimization. All the coding is done with the help of Dev C++ tool.

### 5.2 Results &Discussion

The plant genome (Arabidopsis thaliana) consists of 27,242 gene sequences distributed over five chromosomes(CHR_I to CHR_V) (NCBI site, ftp://ftp.ncbi.nih.gov/genomes/Arabidopsis_thaliana/CHR_I). Part of a nucleotide sequence of a gene from Chromosome I (CHR_I) has been used (see below for details) to test the proposed algorithm. Two types of data have been analyzed for comparisons; one with small alphabet size, i.e. ∑= 4 (nucleotide sequences) and another with big alphabet size, i.e. ∑ = 20 (amino acid sequences). The algorithms are tested on pattern size 4, 8, 12, 16 and 20.

To assess the performance of our algorithm, we considered two well-known algorithms (i.e. Brute force algorithm and Berry-Ravindran algorithm [5]), the improved version of these two well-known algorithms (i.e. Improved Brute force algorithm and Improved Berry-Ravindran algorithm) for comparison with the proposed algorithm. The Improved Brute Force algorithm is a variation of Brute force algorithm, in which we implement our method of searching in the searching phase. In the same way, we implement our method of searching in the searching phase of Berry-Ravindran [5] algorithm to get the Improved Berry-Ravindran algorithm. The algorithms are compared with test cases and their corresponding results are discussed.

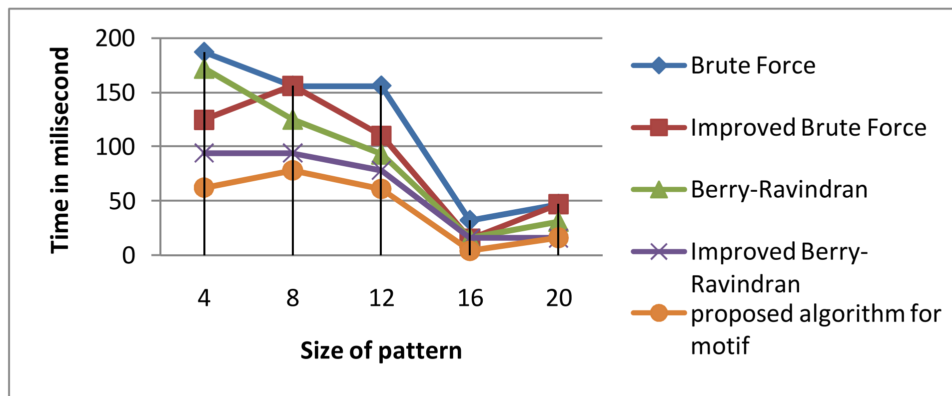



**Figure 1:** Finding Motif in Nucleotide Sequence (time)

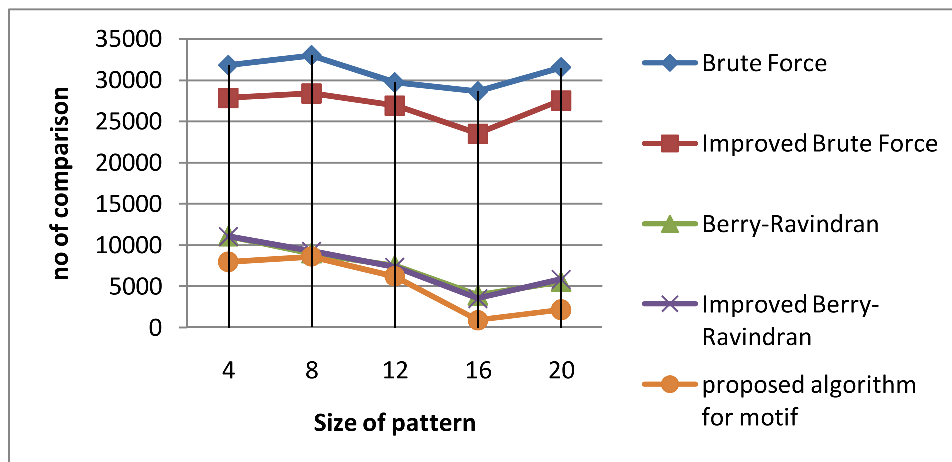



**Figure 2:** Finding Motif in Nucleotide Sequence (character comparison)

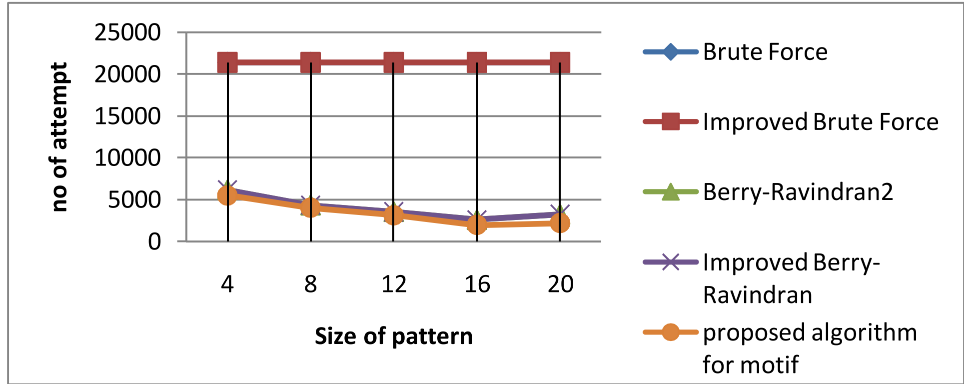



**Figure 3:** Finding Motif in Nucleotide Sequence (attempt)

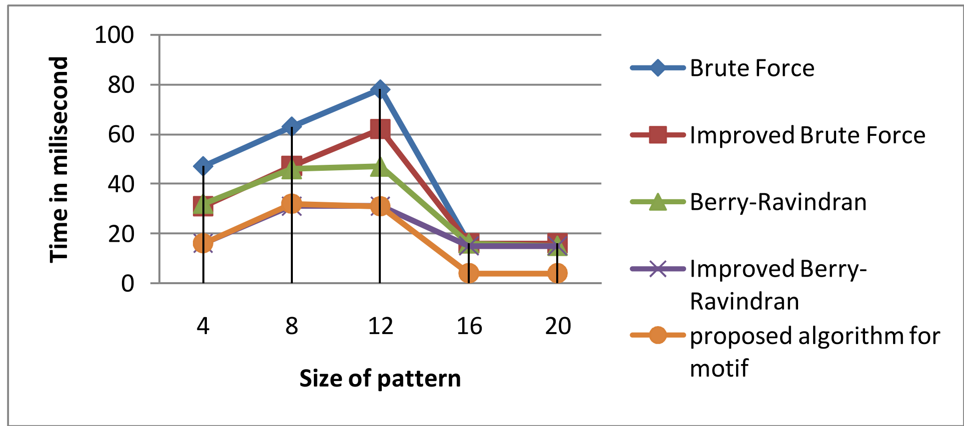



**Figure 4:** Finding Motif in Amino acid Sequence (time)

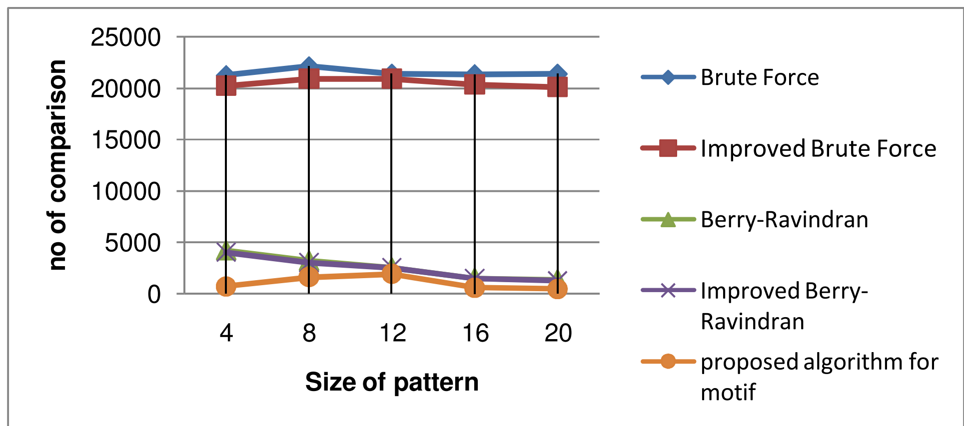



**Figure 5:** Finding Motif in Amino acid Sequence (character comparison)

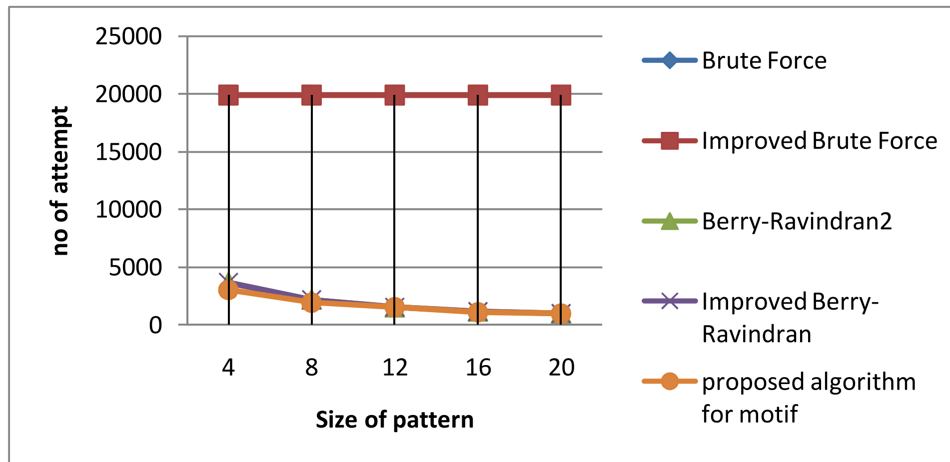



**Figure 6:** Finding Motif in Amino acid Sequence (attempt)

From the above figures it is clear that the proposed algorithm takes less number of time, character comparison and attempt in almost all cases, than other to find all the occurrences of the pattern.

As the experimental result shows the proposed algorithm is more efficient than other for small pattern and the motif normally ranges between 8 and 20. So it is clear that it is more efficient in finding motif in amino acid sequence as well as nucleotide sequence [15].

## 6. CONCLUSIONS

In this paper we present an efficient pattern matching algorithm based on preprocessing of the pattern string by considering four consecutive characters of the text. The idea of considering four consecutive characters is from the fact that occurrence of three successive characters is less frequent than the other possibilities because of which even the shift value obtained is also more compared to Berry-Ravindran and Brute Force algorithms. The concept of searching from both sides makes the algorithm efficient when a mismatch present at the end of the pattern with that of align text window. Theoretically, we prove that the proposed algorithms will shift the pattern faster than other compared algorithms. Experimentally, we show that the proposed algorithms indeed significantly outperform the compared algorithm in almost all cases.

## 7. REFERENCES

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

[2] C. Charras, T. Lecroq(1997). *Handbook of Exact String Matching Algorithms*. [online]. Available: http://www-igm.univ-mlv.fr/~lecroq/string/string.pdf [Oct 08, 2012].

[3] R.S. Boyer, J.S. Moore. *A Fast String Searching Algorithm*, Communications of the ACM, vol. 20, pp.762-772, 1977.

[4] D.M. Sunday. *A Very Fast Substring Search Algorithm*, Journal of Communication of the ACM, vol. 33, pp. 132-142, 1990.

[5] R.M. Karp, M.O. Rabin. *Efficient Randomized Pattern Matching Algorithms*, IBM J. Res. Dev, vol. 31, pp. 249-260, 1987.

[6] T.Raita."Tuning the Boyer-Moore-Horspool string-searching algorithm", Software – Practice Experience, 1992, pp. 879–884.

[7] T. Berry, S. Ravindran. "A Fast String Matching Algorithm and Experimental Results", Proceedings of the Stringology Club Workshop'99, 1999, pp. 16-26.

[8] R. Thathoo,A. Virmani, S. Lakshmi, N. Balakrishnan, K. Sekar. *TVSBS: A Fast Exact Pattern Matching Algorithm for Biological Sequences*, Current Science, vol. 91, pp. 47-53, Jul. 2006.

[9] D. E. Knuth, H. Morris, V. R. Pratt. *Fast Pattern Matching in Strings*, SIAM Journal of computing, vol. 6, pp. 323-350, 1977.

[10] J. H. Morris(Jr), V. R. Pratt. "A Linear Pattern Matching Algorithm", 40th Technical Report, University of California, Berkeley, 1970.

[11] Y.Huang, L. Ping, X. Pan, G. Cai. "A Fast Exact Pattern Matching Algorithm for Biological Sequences", International Conference on Biomedical Engineering and Informatics, IEEE computer Society, Feb. 2008, pp. 8-12.

[12] V. Radhakrishna, B. Phaneendra, V.S. Kumar. "A Two Way Pattern Matching Algorithm Using Sliding Patterns", 3rd International Conforence on Advanced Computer Theory and Engineering (ICACTE), 2010, vol. 2, pp. 666-670.

[13] Hussain, M. Zubair, J. Ahmed, J. Zaffar. "Bidirectional Exact Pattern Matching Algorithm", TCSET'2010, Feb. 2010, pp. 295.

[14] S. S.Sheik,S. K. Aggarwal, A. Poddar, N. Balakrishnan, K. Sekar.A *FAST Pattern Matching Algorithm*, Journal of Chemical Information and Computer Sciences, vol.44, pp. 1251–1256, 2004.

[15] M.Q. Zhang. "Computational prediction of eukaryotic protein-coding genes", Nature Reviews Genetics, vol. 3, Sep. 2002, pp. 698-709.