J. Patuakhali Sci. and Tech. Uni. 2016, 7 (1&2): 119-129 ISSN 1996-4501

INDEX BASED LEAST COUNT FORWARD BACKWARD MULTIPLE PATTERN MATCHING ALGORITHM

Chinmay Bepery[*1], Asma Akter Maria[2], Amina Nasrin Sumyea[2] & Sabina Yasmin[2]

Abstract

DNA related pattern searching is a common activity for molecular biologists. Executing pattern comparison of the DNA and protein sequence is a computationally intensive task. In this paper, a technique called Index Based Least Count Forward Backward Multiple Pattern Matching Algorithm (ILFBMPM) is proposed that eliminates unnecessary comparison with accurate retrieval of the matched pattern. Over exact matching, the proposed algorithm also improved index based search which can skip the avoidable comparisons in the DNA sequence. The number of comparisons of the proposed algorithm is very low relative to other existing popular methods. In 75% cases, our algorithm provides better result than the current state of the art.

Key words: DNA Sequence, Index Based Search, Pattern Matching, Sequence Analysis

Introduction

DNA is the basic blueprint of human life and the DNA sequence of each human is unique. Pattern matching in a DNA sequence or searching a pattern from a large sequence is one of the main research area in Bioinformatics. The DNA sequences of all human beings are 99.9% identical meaning that the difference is only 0.1%. The total amount of DNA extracted from the organism is increased exponentially. So pattern matching techniques plays a major role in various applications in computational biology. It focuses on finding the particular pattern in a given DNA sequence. Obtaining valid information from the sequence becomes more difficult because of the rapid growth of data size. Given a pattern $P = \{p_0, p_1, \ldots \ldots, p_{m-1}\}$ of length m=|P| and a text string $\boldsymbol{S} = \{\boldsymbol{s_0}, \boldsymbol{s_1}, \ldots \ldots, \boldsymbol{s_{n-1}}\}$ of length n i.e., n=|S| over an alphabet set $\sum$ where $\sum = \{\boldsymbol{A, C, G, T}\}$ and m $\leq$ n. There are two categories of existing pattern matching algorithms.

- Exact pattern matching algorithms
- Inexact/approximate pattern matching algorithms

Exact pattern matching algorithm finds one or all exact occurrences of a string in a given sequence. Some exact matching algorithms are Naive Brute force algorithm (Mansi & Odeh, 2009)Boyer-Moore algorithm (Chen, Lu, & Ram, 2004), KMP Algorithm (Knuth, Morris, & Pratt, 1977). A helpful utility command "grep" used in the Unix environment (Wu & Manber, 1992) allows user to search globally for lines matching the regular expression.

[1] Department of Computer Science & Information Technology, [2]Faculty of Computer Science & Engineering, Patuakhali Science & Technology University [*]Corresponding author: chinmay.cse@pstu.ac.bd

Inexact pattern matching algorithm is sometimes referred to as approximate pattern matching algorithm. Inexact/approximate pattern matching matches some portions of given sequence with *k* mismatches/differences. The inexact sequence data arise in several fields and applications such as computational biology, signal processing and text processing. Some Inexact pattern matching algorithms are Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. As a consequence of mutation activity within DNA sequence, the biological inference does not expect an identical match, rather a high sequence similarity usually implies significant functionality or structural functionality.

The field of bioinformatics has many applications such as text editors, search engine, molecular medicine, industry, agriculture and comparative biology. In search engines or many other information retrieval systems, it is necessary to find one or more patterns rapidly. The main purposes of pattern matching algorithms are to shorten the number of character comparisons and reduce the required time in worst and average case analysis. The most recent methods are proposed for the problem by (Bhukya & Somayajulu, 2010), (Bhukya & Somayajulu, 2011a, 2011b).To the best of our knowledge, Index based Forward Backward Multiple Pattern Matching algorithm IFBMPM (Bhukya & Somayajulu, 2010), is the current state-of-the-art. In this paper, we propose Index based Least count Forward Backward Multiple Pattern Matching algorithm (ILFBMPM). By using the ILFBMPM algorithm, the number of comparisons decreases when compared with Multiple Skip Multiple Pattern Matching Algorithm MSMPMA (Alqadi, Aqel, & El Emary, 2007) and IFBMPM.

The rest of this paper is arranged as follows. We briefly present the notations and related work in Section II. Section III deals with Materials and Methods, i.e., ILFBMPM algorithm for DNA sequence. Results and discussion are presented in Section IV and finally, this article is concluded in Section V.

Notations and Related Work

In our proposed algorithm we use an alphabet set of characters $\sum = \{A, C, G, T\}$ for DNA sequence. The notations of this paper are given below:

DNA sequence characters for given string and predefined pattern $\sum = \{A, C, G, T\}$. m=|P| denotes the length of the pattern P. The length of string S is denoted by n=|S|. An empty string is denoted by φ. Least counted character of the pattern is defined as *pminchar*. Number of occurrences of *pminchar* in P and S is denoted by c_p and c_s respectively. T_{p1} denotes the index of 1st occurrence of *pminchar* in P and T_{sj} denotes the index of *jth* occurrence of *pminchar* in S.

In this section, we briefly discuss different types of exact pattern matching algorithms. The brute force algorithm (Mansi & Odeh, 2009) checks at all positions in the text between 0 and *n-m*, whether the pattern occurs or not. It shifts the pattern by exactly one position to the right after each attempt. During the searching phase, the comparisons can be done in any order. The time complexity of this searching phase is *0(m*n)* (Chen et al.,

2004).The Knuth-Morris-Pratt algorithm (Knuth et al., 1977) performs the comparisons from left to right of the input string, but shifts the pattern more intelligently than the brute-force algorithm. It preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself. In preprocessing phase the space and time complexity is *0*(m*n). In searching phase the time complexity is *0(m*m)* Wu, Manber, & Myers, (1992) proposed an algorithm for approximate limited expression matching. Wu and Manber (1992) proposed algorithm for fast text searching allowing errors. Boyer-Moore (BM) algorithm (Boyer & Moore, 1977) performs character comparisons in reverse order from right to the left of the pattern and does 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 worst case complexity is *0(m+n)* and the average case complexity is *0(n/m)* (Chen et al., 2004).

Automation method for finding approximate patterns in strings is proposed by Ukkonen, (1985). The author proposed the idea of using a DFA for solving the inexact matching problem. The IFBMPM (Bhukya & Somayajulu, 2010) algorithm checks for each first occurrence of the first character of the pattern and compares one character from the left and one character from the right until all characters are compared. If all characters match to the pattern, it prints the starting index of the sequence. If any character mismatches, it skips the test and continues to check the next occurrence of the first character of the pattern available in the index table. In Index Based Sequential Multiple Pattern Matching with Least Count (ISMPMC) Algorithm (Bhukya & Somayajulu, 2011a), when the pattern is aligned with string, it will match the characters sequentially one by one from the starting character of the given pattern. We analyze the previous algorithms and find that the concept of exact multiple pattern matching algorithms are more accurate than that of inexact pattern matching algorithms.

Materials and Methods

In this research, we use the index of the DNA sequence of character set Σ to search a pattern in a string. Let $S = \{S_0S_1S_2 \ldots \ldots \ldots S_{n-1}\}$ be a string of length n and the pattern $P = \{P_0P_1P_2 \ldots \ldots \ldots P_{m-1}\}$ of length m. It is assumed that both the string and the pattern are constructed with four $\{A,C,G,T\}$ characters and $n>>m$. Our proposed algorithm consists of two phases, pre-processing (the given string S based on the pattern) and searching. In the preprocessing phase, two tables are used to hold the indices of characters {A,C,G,T} of S and P respectively. We collect all indices into tables Stab and Ptab for the occurrences of four characters of Σ from S and P into different buckets with total number of times. Next, we find out that character over $\sum$ whose frequency is minimum in S and P respectively from different buckets. Then we find out the least counted character (*pminchar*) from pattern P. This *pminchar* is the driving parameter of our algorithm. The pattern does not exist in the string if $c_p > c_s$ where, c_p = number of occurrences (*pminchar*) in P and c_s = number of occurrences (*pminchar*) in S. According to the first index of *pminchar*, the pattern is aligned with the string. Now, forward-backward matching is started. If there is any mismatch, we skip and consider the next index of

pminchar from the string and repeat forward-backward matching. If the pattern is completely matched with the string, we consider the next index of *pminchar* from the string and repeat the above procedure.

Proposed ILFBMPM Algorithm

Input: String *S* of *n* characters and a pattern *P* of *m* characters, where *S, P* $\in \sum$ = {*A, C, G, T*}

Output: The no. of occurrences and the positions of *P* in *S.*

Step1. Set flag: = 0, num_occ: = 0;

Step2. Create index table for P and S using two integer arrays Stab[4][m] and Ptab[4][n];

Step 3. Create ascending ordered sorted array psort[4] as count table from Ptab[4][n];

Step 4. Find least counted value from psort[1]and least counted character ($pminchar$) from Ptab[4][n] based on psort[1];

Step 5. If $c_p > c_s$ *;[* c_p = *number of occurrenc* (*pminchar*) *in pattern P and* c_s = *number of occurrence* (*pminchar*) *in string S.]*

PRINT "Pattern does not exist" then
Exit;

Step 6. Repeat steps 7 to 9 for j: =1, 2, 3... c_s ;

Step 7. Set L: = T_{p1} , q: = T_{sj} −L, K: =|P| − T_{p1}, r: = T_{sj} + K, u: =0, v: =|P|−1*; [*T_{p1} = *index of 1st occurrence of pminchar in P and* T_{sj} = *index of jth occurrence of pminchar in S, q = Starting index for comparison from S, r = Terminal index for comparison from S, u = Starting index of P and v = = Terminal index of P]*

Step 8. Repeat for u: =1, 2…….v while u<v;
If S[q]!= P[u] or S[r]!= P[v] then
Set flag: = flag+1, j: =j+1 and break;
Else
(a) Set q: =q+1 and r: = r−1;
(b) Set u: =u+1 and v: =v−1;
(c) Set flag: =0;
[End if]
[End for]

Step 9. If flag = 0 then
(a) PRIRT "Pattern found at position j";
(b) Set num_occ: = num_occ+1;
(c) Set j: =j +1.
[End if]
[End for]

Step 10. PRINT "Occurrence no is: num_occ";

Step 11. Exit.

The algorithm takes the given string and checks the occurrences of each pattern within the string. If the pattern is found in the string, it prints the number of matched pattern and their starting index positions. It first creates index tables, then creates count tables for *S* and *P*. After sorting pattern count table the least count character is aligned with the corresponding string. Then it compares forward-backward with respect to the string.

Computational Analysis

Let,the given input string $S = \{S_0S_1S_2 \dots \dots \dots S_{n-1}\}$ over the alphabet $\sum = \{A, C, G, T\}$ and $P = \{P_0P_1P_2 \dots \dots \dots P_{m-1}\}$ be a pattern where the length of string is denoted by $|S|=n$ and length of the pattern is denoted by $|P|=m$ where $|S|>>|P|$. If the Index Table (which hold an index of S) is either one of the characters having count 0 and that same character has a non-zero value in the pattern index table, the pattern cannot be found in S. Other traditional cases are as follows:

Case 1: If $S = \varphi$i.e. $|S| = 0$and for any $|P| \geq 0$ then the number of occurrences of *P*in *S*is *0*.

Case 2: If $S = \varphi$i.e., $|S| = 0$and $P = \varphi$i.e., $|P| = 0$ then the number of occurrences of *P*in *S*is *0*.

Case 3: If $S = \varphi$i.e., $|S| = 0$and $P \neq \varphi$i.e., $|P| = 0$then the number of occurrences of *P*in *S*is *0*.

Case 4: If $S = \varphi$i.e., $|S| = 0$and $P = \varphi$i.e., $|P| \neq 0$then the number of occurrences of *P*in *S*is *0*.

Case 5: If $S \neq \varphi$i.e., $|S| \neq 0$and for any of $|P| = 0$then the number of occurrences of *P*in *S*is *0*.

Case 6: If $S \neq \varphi$i.e., $|S| \neq 0$, $P \neq \varphi$i.e., $|P| \neq 0$and $|S| \leq |P|$then the number of occurrences of *P*in *S*is *0*.

Example 1

Let S= TCAACTGACTAGGTCATGACTGACTAGCA be a string of 29 characters and P= ACTGACTA be a pattern of 8 characters. The following index table stores all the index of each character A, C, G and T in its corresponding row. The 0^{th} row stores the indexes of occurrences of the character A, 1^{st} row for C, 2^{nd} row for G and 3^{rd} row stores indexes for T.

Table 1: Index table for string S

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**					**Count**
A	2	3	7	10	15	18	22	25	28	--	--	--	--	9
C	1	4	8	14	19	23	27	--	--	--	--	--	--	7
G	6	11	12	17	21	26	--	--	--	--	--	--	--	6
T	0	5	9	13	16	20	24	--	--	--	--	--	--	7

Now, we create a count table for the predefined pattern P= ACTGACTA from which we pick the least count character to align in the string.

Table 2: Index table for pattern P

	1	2	3				Count
A	0	4	7	--	--	--	3
C	1	5	--	--	--	--	2
G	2	--	--	--	--	--	1
T	3	6	--	--	--	--	2

Here G is occurring least no of times in the pattern so G is used as the initial alignment and once if there is a match of G with the string, rest of the characters will be compared in a forward-backward order for the pattern matching process. The algorithm maps to the first occurrence of G according to the table and then start comparing the first element of pattern with the possible match in the string relative to that G. Pattern *P* go least count index 6.

S=T C A A C T **G** A C T A G G T C A T G A C T G A C T A G C A

P=A C T **G** A C T A

The least count character G is matched and then it starts forward-backward comparison.

S=T C A **A** C T **G** A C T**A** G G T C A T G A C T G A C T A G C A

P=**A** C T **G** A C T **A**

The first position (forward) and last position (backward) character of pattern P are matched above, then it proceeds next similar forward and backward comparison shown below:

S=T C A **AC** T **G** A C **TA** G G T C A T G A C T G A C T A G C A

P=**AC** T **G** A C **TA**

Here another match is found then it precedes next forward-backward matching that is shown in below:

S=T C A **ACTG** A **CTA** G G T C A T G A C T G A C T A G C A

P=**ACTG** A **CTA**

Above comparing matches and it goes for next comparing and complete forward-backward comparison shown in below:

S=T C A **ACTGACTA** G G T C A T G A C T G A C T A G C A

P=**ACTGACTA**

All the characters are matched, so the pattern is found at position 6 of G. Now we go to the second index in index table for G and align G to it.

S=T C A A C T GA C T A **G** G T C A T G A C T G A C T A G C A

P=A C T **G** A C T A

After aligning G we do forward-backward comparison that are given below:

S=T C A A C T GA C T A **G** G T C **A** T G A C T G A C T A G C A

P=A C T **G** A C T **A**

In forward-backward comparison, the first (forward) character is not matched. So we jump the next index of G.

S=T C A A C T GA C T A G **G** T C A T G A C T G A C T A G C A

P=A C T **G** A C T A

After aligning G we do forward-backward comparison that are given below:

S=T C A A C T GA C T A G **G** T C A T G A C T G A C T A G C A

P=A C T **G** A C T A

In forward-backward comparison, the first (forward) and last (backward) character is not matched. So we jump the next index of G.

S=T C A A C T GA C T A G G T C A T **G** A C T G A C T A G C A

P=A C T **G** A C T A

After aligning G we do forward-backward comparison that are given below:

S=T C A A C T GA C T A G G T C A T **G** A C T G A C T A G C A

P=A C T **G** A C T A

In forward-backward comparison, the first (forward) and last (backward) character is not matched. So we jump the next index of G.

S=T C A A C T GA C T A G G T C A T G A C T **G** A C T A G C A

P=A C T **G** A C T A

After aligning G we do forward-backward comparison that are given below:

S=T C A A C T GA C T A G G T C A T G **A** C T **G** A C T **A** G C A

P=**A** C T **G** A C T **A**

The first position (forward) and last position (backward) character of pattern P are matched above, then it proceeds next similar forward and backward comparison shown below:

S=T C A A C T GA C T A G G T C A T G **AC** T **G** A C **TA** G C A

P=**AC** T **G** A C **TA**

Here another match is found then it precedes next forward-backward matching that is shown in below:

S=T C A A C T GA C T A G G T C A T G **ACTG** A **CTA** G C A

P=**ACTG** A **CTA**

Above comparing matches and it goes for next comparing and complete forward-backward comparison shown in below:

S=T C A A C T GA C T A G G T C A T G **ACTGACTA** G C A

P=**ACTGACTA**

All the characters are matched, so the pattern is found at position 21 of G. The last index of G is stored in the position 26, there are two characters after the 26^{th} position in the string, but the pattern has 4 characters after the least count character G. It exceeds the corresponding length of string. Finally, the search for P in S is completed, *P* occurred two times in the string *S* at index position 3 and 18.

Example 2

The sequence has been taken from the Index based Forward Backward Multiple Pattern Matching (IFBMPM) algorithm (Bhukya & Somayajulu, 2010) which is firstly used in MSMPMA(Alqadi et al., 2007) for testing the ILFBMPM algorithm. It explains large sequence data by taking a DNA biological sequence $S \in \sum$ of size n=1024 and pattern $P \in \sum$. Let S be the following DNA sequence.

AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAGTGTTACCAACTCGGG
TGCCTATTGGCCTCCAAAAAAGGCTGTTCAACGCTCCAAGCTCGTGACCTCGTC
ACTACGACGGCGAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGACGCGTG
GTGAATCCTATGGGTTAGGATCGTGTCTACCCCAAATTCTTAATAAAAAACCTA
GGACCCCCTTCGACCTAGACTATCGTATTATGGACAAGCTTTAACTGTCGTACT
GTGGAGGCTTCAAAACGGAGGGACCAAAAAATTTGCTTCTAGCGTCAATGAAA
AGAAGTCGGGTGTATGCCCCAATTCCTTGCTGCCCGGACGGCCAGGCTTATGTA
CAATCCACGCGGTACTACATCTTGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAA
TCATAGCGGTACTTCATAATGGGACACAACGAATCGCGGCCGGATATCACATCT
GCTCCTGTGATGGAATTGCTGAATGCGCAGGTGTGAATACTGCGGCTCCATTCG
TTTTGCCGTGTTGATCGGGAATGCACCTCGGGGACTGTTCGATACGACCTGGGA
TTTGGCTATACTCCATTCCTCGCGAGTTTTCGATTGCTCATTAGGCTTTGCGGTA
AGTAAGTTCTGGCCACCCACTTCGAGAAGTGAATGGCTGGCTCCTGAGCGCGTC
CTCCGTACAATGAAGACCGGTCTCGCGCTAAATTTCCCCCAGCTTGTACAATAG
TCCAGTTTATTATCAAAGATGCGACAAATAAATTGATCAGCATAATCGAAGATT
GCGGAGCATAAGTTTGGAAAACTGGGAGGTTGCCAGAAAACTCCGCGCCTACT
TTCGTCAGGATGATTAAGAGTATCGAGGCCCCGCCGTCAATACCGATGTTCTTC
GAGCGAATAAGTACTGCTATTTTGCAGACCCTTTGCCAGGCCTTGTCTAAAGGT
ATGTTACTTAATATTGACAATACATGCGTATGGCCTTTTCCGGTTAACTCCCTG.

The index table for S is very large to show here. The number of occurrences and the number of comparisons are shown in the following table 3 by taking different patterns form 1 to 8 of different cases.

Result and Discussion

To compare with MSMPMA (Alqadi et al., 2007) and IFBMPM (Bhukya & Somayajulu, 2010), same dataset of size n=1024 is used for our experiment. For eight different patterns, the number of occurrences, the number of comparisons and CPC (comparison per character) ratio of three algorithms MSMPMA, IFBMPM, and ILFBMPM are shown in the Table 3.A contrast has been done on the basis of number of comparisons and CPC ratio with the other algorithms. The proposed algorithm gives better performance and lower CPC ratio than the current algorithm.

Table 3: Comparison of different algorithms with ILFBMM

Pattern (P's)	No. of characters	No. of occurrences	MSMPMA	IFBMPM	ILFBMPM Model	CPC improvement in ILFBMPM (%)				
			No. of Com.	CPC Ratio	No. of Com.	CPC Ratio	No. of Com.	CPC Ratio	$\frac{a-c}{a}\times 100$	$\frac{b-c}{b}\times 100$
				a		b		c		
A	1	259	1024	1.00	518	0.51	518	0.51	49.41%	0.00%
AG	2	53	1230	1.20	624	0.61	494	0.48	59.84%	20.83%
CAT	3	11	1298	1.27	567	0.55	672	0.66	48.23%	-18.52%
AACG	4	5	1359	1.33	614	0.60	624	0.61	54.08%	-1.63%
AAGAA	5	2	1375	1.34	616	0.60	534	0.52	61.16%	13.31%
AAAAAAGG	8	1	1394	1.36	634	0.62	554	0.54	60.26%	12.62%
TTCTTAATAAAA	12	1	1390	1.36	651	0.64	546	0.53	60.72%	16.13%
GGCTGTTCAACGCTC	15	1	1349	1.32	598	0.58	570	0.56	57.75%	4.68%

The number of observations come out from the experimental results: 1)Applicable only for DNA related biological data and provides good performance. 2) Reduction in number of comparisons. 3) The ratio of comparisons per character has gradually reduced. 4) For each pattern, our algorithm starts from the first least counted matching character of the pattern which avoids the unnecessary comparisons.

Many algorithms are introduced in literature to solve the pattern matching problem with less comparisons and in less time but each has some limitations. The proposed Index based least count Forward-Backward algorithm is one simple solution for such needs. Here, ILFBMPM provides better results in terms of comparisons than IFBMPM in 6 cases out of 8 cases. It also provides better results in terms of comparison than MSMPMA in all cases.

This algorithm can be appreciated for decreasing the number of comparisons as compared with the other algorithms as shown in the following figure 1.

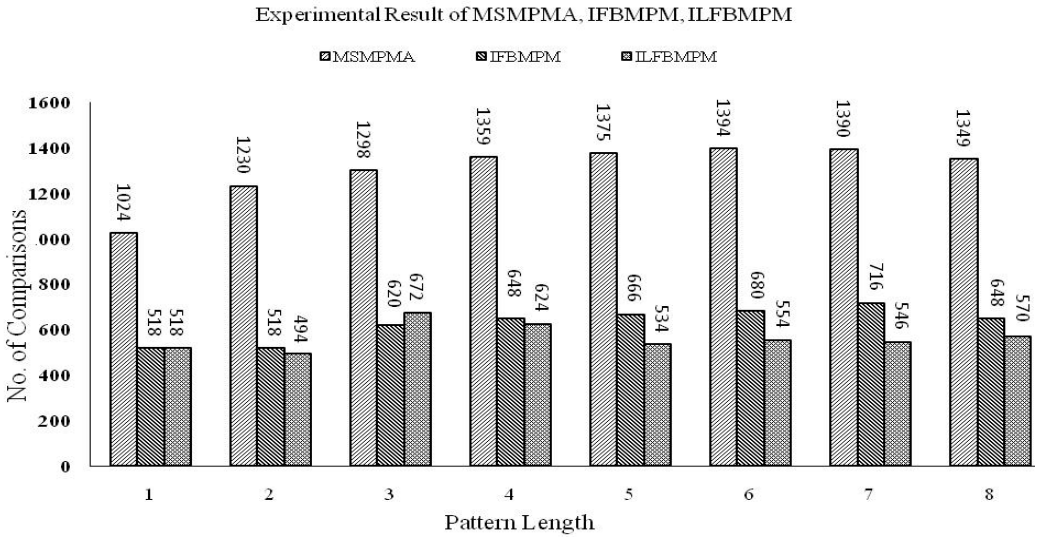

Figure 1. Comparison of different algorithms

Figure 1shows the number of comparisons for ILFBMPM is less than the MSMPMA algorithm. Figure 2 shows the comparison among MSMPMA, IFBMPM, and ILFBMPM based on CPC ratio. In 6 out of 8 cases, proposed algorithm provides lower CPC ratio than the current state of the art.

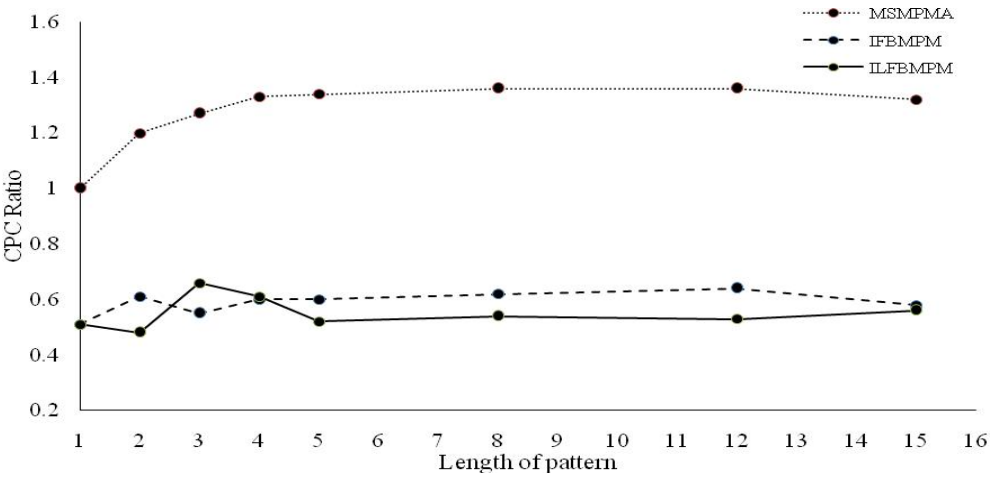

Figure 2. Comparison of different algorithms for CPC ratio

By using the current technique, it is clear that the proposed algorithm performs well because of less number of comparisons than MSMPMA and IFBMPM algorithm. The proposed ILFBMPM algorithm uses least counted character of pattern for initial alignment and proceed forward-backward matching. The proposed algorithm might be improved, if the gaps of least counted character in pattern are calculated and unnecessary comparisons can be avoided (if gap mismatch occur in string and pattern).

Conclusion

The proposed algorithm ILFBMPM gives efficient results for searching patterns. We have compared the number of cases with another popular algorithm MSMPMA and IFBMPM. Our algorithm showed 100% better result than MSMPMA and 75% better result than IFBMPM in 8 experimental cases. It is a very simple approach for finding multiple patterns from a given text. We have implemented this algorithm with DNA sequence further it can be extended to protein or other sequence.

References

Alqadi, Z. A., Aqel, M., & El Emary, I. M. 2007. Multiple skip Multiple pattern matching algorithm (MSMPMA). *IAENG International Journal of Computer Science, 34*(2): 14-20.

Bhukya, R., & Somayajulu, D. 2010. An Index Based Forward Backward Multiple Pattern Matching Algorithm. *algorithms, 1*, 2.

Bhukya, R., & Somayajulu, D. 2011a. *An Index Based Sequential Multiple Pattern Matching Algorithm Using Least Count.* Paper presented at the International Conference on Life Science and Technology IPCBEE.

Bhukya, R., & Somayajulu, D. 2011b. *Index Based Sequential Multiple Pattern Matching Algorithm Using Pair Indexing.* Paper presented at the International Conference on Life Science and Technology IPCBEE.

Boyer, R. S., & Moore, J. S. 1977. A fast string searching algorithm. *Communications of the ACM, 20*(10): 762-772.

Chen, L., Lu, S., & Ram, J. 2004. *Compressed pattern matching in DNA sequences.* Paper presented at the Computational Systems Bioinformatics Conference, 2004. CSB 2004. Proceedings. 2004 IEEE.

Knuth, D. E., Morris, J., James H, & Pratt, V. R. 1977. Fast pattern matching in strings. *SIAM journal on computing, 6*(2): 323-350.

Mansi, R. H., & Odeh, J. Q. 2009. On Improving the Naïve String Matching Algorithm. *Asian Journal of Information Technology, 8*(1): 14-23.

Ukkonen, E. 1985. Finding approximate patterns in strings. *Journal of algorithms, 6*(1): 132-137.

Wu, S., & Manber, U. 1992. *Agrep–a fast approximate pattern-matching tool.* Paper presented at the Proc. of USENIX Technical Conference.

Wu, S., Manber, U., & Myers, G. 1992. *A sub-quadratic algorithm for approximate limited expression matching*: University of Arizona, Department of Computer Science.