2011 International Conference on Bioscience, Biochemistry and Bioinformatics
IPCBEE vol.5 (2011) © (2011) IACSIT Press, Singapore

An Index Based Sequential Multiple Pattern Matching Algorithm

Raju Bhukya
Assistant Professor
Department of Computer Science and Engineering,
National Institute of Technology, Warangal.A.P.India
E-mail: rajubhukya@gmail.com, raju@nitw.ac.in

DVLN Somayajulu
Professor
Department of Computer Science and Engineering,
National Institute of Technology, Warangal.A.P.India
E-mail: somadvlns@gmail.com, soma@nitw.ac.in

***Abstract*—Pattern matching is well known and fundamental problem in bioinformatics. It is one of the emerging areas in computational biology. Searching DNA related data is a common activity for molecular biologists for finding the structural and functional behavior in protein and genes. In this research we propose a new pattern matching technique called an index based sequential multiple pattern matching algorithm. Many algorithms have been proposed but more efficient and robust methods are needed for the multiple pattern matching algorithms. In the current approach we explore a new technique which avoids unnecessary comparisons in the DNA sequence. Due to this, the number of comparisons gradually decreases and comparison per character ratio of the proposed algorithm reduces accordingly when compared to the some of the existing popular methods. The experimental results show that there is considerable amount of performance improvement due to this the overall performance increases.**

Keywords- DNA Sequence; Index; Partition; Pattern.

I. INTRODUCTION

DNA is the basic blue print of life and it can be viewed as a long sequence over the four alphabets *A, C, G* and *T*. DNA contains genetic instructions of an organism. It is composed of nucleotides of four types. Adenine(A), Cytosine(C), Guanine (G), and Thymine (T). The amount of DNA extracted from the organism is increasing exponentially. So pattern matching techniques plays a vital role in various applications in computational biology for data analysis related to protein and gene in structural as well as the functional analysis. It focuses on finding the particular pattern in a given DNA sequence. The biologists often queries new discoveries against a collection of sequence databases such as GENBANK, EMBL and DDBJ to find the similarity sequences. DNA pattern matching is an a fundamental and upcoming areas in computational molecular biology. The field of bioinformatics has many applications in the modern world which includes text editors, search engines, molecular medicines, industry, agriculture and Comparative biology. As the size of the data grows it becomes more difficult for users to retrieve necessary information from the sequences. Hence more efficient and robust methods are needed for fast pattern matching techniques.

Let $P = \{p_1, p_2, p_3,...,p_m\}$ be a set of patterns which are strings of nucleotide sequence characters from a fixed alphabet set called $\sum = \{A, C, G, T\}$. Let T be a large text consisting of characters in $\sum$. In other words T is an element of $\sum^*$. The problem is to find all the occurrences of pattern P in text T. It is an important application widely used in data filtering to find selected patterns, in security applications, and is also used for DNA searching. Many existing pattern matching algorithms are reviewed and classified in two categories

- Exact string matching algorithm
- Inexact/approximate string matching algorithms

Exact string matching algorithm is for finding one or all exact occurrences of a string in a sequence. The problem can be stated as: Given a pattern p of length m and a string /Text T of length n ($m \leq n$). Find all the occurrences of p in T. The matching needs to be exact, which means that the exact word or pattern is found. Some exact matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm [2], Knuth-Morris-Pratt Algorithm [5].

Inexact /Approximate pattern matching is sometimes referred as approximate pattern matching or matches with k mismatches/ differences. This problem in general can be stated as: Given a pattern P of length m and string/text T of length n. ($m \leq n$). Find all the occurrences of sub string X in T that are similar to P, allowing a limited number, say k different characters in similar matches. The Edit/transformation operations are insertion, deletion and substitution. Inexact/ Approximate string matching algorithms are classified into: Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. Inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing. Pattern matching algorithms have two main objectives.

- Reduce the number of character comparisons required in the worst and average case analysis.
- Reducing the time requirement in the worst and average case analysis.

The rest of the paper is organized as follows. We briefly present the background and related work in section II. Section III deals with proposed model *i.e.*, ISMPM algorithm for DNA sequence. Results and discussion are presented in Section IV and we make some concluding remarks in Section V.

II. BACKGROUND AND RELATED WORK

This section reviews some work related to DNA sequences. An alphabet set $\sum = \{A, C, G, T\}$ is the set of

characters for DNA sequence which are used in this algorithm.

The following notations are used in this paper:

DNA sequence characters $\sum = \{A, C, G, T\}$.

ϕ denotes the empty string.

$|P|$ denotes the length of the string *P*.

S[n] denotes that a text which is a string of length *n*.

P[m] denotes a pattern of length *m*.

CPC-Character per comparison ratio.

String matching mainly deals with problem of finding all occurrences of a string in a given text. In most of the applications it is necessary for the user and the developer to be able to locate the occurrences of specific pattern in a sequence. Some of the exact string matching algorithms are available, such as Naïve string search, Brute-force algorithm, Bayer-Moore algorithm [2], Knuth-Morris-Pratt algorithms [5] . In Brute-force algorithm the first character of the pattern *P* is compared with the first character of the string *T*. If it matches, then pattern *P* and string *T* are matched character by character until a mismatch is found or the end of the pattern *P* is detected. If mismatch is found, the pattern *P* is shifted one character to the right and the process continues. The complexity of this algorithm is *O(mn).* The Bayer-Moore algorithm[2] applies larger shift-increment for each mismatch detection. The main difference the Naïve algorithm had is the matching of pattern *P* in string *T* is done from right to left *i.e.,* after aligning *P* and string *T* the last character of *P* will matched to the first of *T* . If a mismatch is detected, say *C* in *T* is not in *P* then *P* is shifted right so that *C* is aligned with the right most occurrence of *C* in *P*. The worst case complexity of this algorithm is $O(m+n)$ and the average case complexity is *O(n/m).*

In Index based forward backward multiple pattern matching IFBMPMA [8] the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches The Knuth-Morris-Pratt algorithm [5] is based on the finite state machine automation. The pattern *P* is pre-processed to create a finite state machine *M* that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is *O(m+n).* In approximate pattern matching method the oldest and most commonly used approach is dynamic programming. In 1996 Kurtz[6] proposed another way to reduce the space requirements of almost *O(mn).* The idea was to build only the states and transitions which are actually reached in the processing of the text. The automaton starts at just one state and transitions are built as they are needed. The transitions those were not necessary will not be build. By using dynamic programming approach especially in DNA sequencing Needleman-Wunsch algorithm [7] and Smith-waterman[9] algorithms are more complex in finding exact pattern matching algorithm. By this method the worst case complexity is *O(mn).* In the MSMPMA [11] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. By using this index as the starting point of matching, it compares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges from 1 to m-1). Harspool [4] does not use the good suffix function, instead it uses the bad character shift with right most character .The time complexity of the algorithm is *O(mn).* Berry-Ravindran [1] calculates the shift value based on the bad character shift for two consecutive text characters in the text immediately to the right of the window. This will reduce the number of comparisons in the searching phase. The time complexity of the algorithm is *O(nm)* .Sunday [3] designed an algorithm quick search which scans the character of the window in any order and computes its shift with the occurrence shift of the character *T* immediately after the right end of the window.

Ukkonen [10] proposed automation method for finding approximate patterns in strings. He proposed the idea using a DFA for solving the inexact matching problem. Though automata approach doesn't offer time advantage over Boyer-Moore algorithm [2] for exact pattern matching. The complexity of this algorithm in worst and average case is *O(m+n).*In this every row denotes number of errors and column represents matching a pattern prefix. Deterministic automata approach exhibits *O(n)* worst case time complexity. The main difficulty with this approach is construction of the DFA from NFA which takes exponential time and space. The first bit-parallel method is known as "*shift-or*" which searches a pattern in a text by parallelizing operation of non deterministic finite automation. This automation has *m+1* states and can be simulated in its non deterministic form in *O(mn)* time. The filtering approach was started in 1990. This approach is based upon the fact it may be much easier to tell that a text position doesn't match. It is used to discard large areas of text that cannot contain a match. The advantage in this approach is the potential for algorithms that do not inspect all text characters.

III. INDEX BASED SEQUENTIAL MULTIPLE PATTERN MATCHING METHOD(ISMPM)

In the proposed work we use the indexes for the DNA sequence of character set $\sum$ to search a pattern in a string. Let *S* be a string of length *n* and the pattern *P* of length *m*. Let $\sum^*$ be the set of all possible strings with the alphabet set $\sum$. Then $S, P \in \sum^*$, $|S| = n$ and $|P| = m$. Generally $|P| \leq |S|$ *i.e.,* $m \leq n$.

Initially it needs to build up a table called index table, which is useful to reduce the number of comparisons. Once the index table is created for a given string it is used for all types of input patterns. After building the index table for each pattern *P* it need to check whether the pattern *P* occurs in the sequence or not. It check for the pattern in the DNA sequence in a sequential order by checking character by character. If the pattern occurs in the sequence it prints the starting index of the pattern else if the pattern doesn't match then it will stop searching and then go to the next index stored in the index table of the same row. For each pattern we start checking from the first character of the pattern by using the index table.

A. ISMPM Algorithm

Input: String S of n characters and a pattern P of m characters, where $S, P \in \sum^*$.

Output: The no. of occurrence and the positions of pattern P in the string S.

```
Algorithm: Integer arrays indexTab[4][n], charIndex[4]
Integer found:=1, n_occ:=0,n_cmp:=1
   FOR i:=0;i<n;i:=i+1
indexTab[(S[i]-64)%5][charIndex[(S[i]- 64)%5]++]:=i;
  End FOR
      FOR i:=0;i<chatIndex[(P[0]-64)%5];i:=i+1
            found:=1
            IF i+m-1 > n-1
                  found:=0
            End IF
            FOR j:=0;j < m;j:=j+1
                  n_cmp:=n_cmp+1
                  IF P[j] ≠ S[i+j]
                        found:=1
            End FOR
      IF found:=1
            n_occ:=n_occ+1
PRINT  "Pattern Found At Location i, Occurrence no is: n_occ"
       End IF
       End FOR
```

The index based algorithm for multiple pattern matching uses a table (2D vector) called index table. The basic idea used here is to store all the indexes of each character in its corresponding row in the 2D vector. Here it uses a new technique called ASCII value based indexing technique, which is used to reduce the pre-processing time and number of comparisons. The subscript [*(S[i]-64)%5*] always returns a subscript value in the range 0,1,2,3 which is needed for subscripting 2D array of size *[4][n]*. The subscript values 0,1,2,3 represent the characters T, A, G, C respectively.

TABLE I. ARRAY SUBSCRIPTION VALUES FOR DNA

S.No	DNA	ASCII Value	ASCII Value-64	(ASCIIValue - 64)%5	Array Subscript
1	A	65	1	1	1
2	C	67	3	3	3
3	G	71	7	2	2
4	T	84	20	0	0

So for each character in the string of the function *(S[i]-64)%5)* directly references to its corresponding row in the *indexTable[ ][ ]*. The vector *charIndex[4]* stores the counter value of each occurrence of each character with reference to [*(S[i]-64)%5*]. For each occurrence of the first character of the pattern, the proposed algorithm compares all the characters one by one from left to right, if all characters match with the pattern it prints pattern from its starting index. If any character mismatches it skips the search and continues to search from the next index by using index table. Let *S* be the string of length *n*, *P* be the pattern of length *m* and *S, P*$\in\sum$*, and *i* be the index of the first character of the pattern *P* in the string *S*. Let X_i denote the character at the i^{th} location in the string *X*. Now for each value of *i*, we check whether

$$P_r = S_{i+r} \text{ for } r=0, 1, 2, \ldots, m-1.$$

If it is true for all values of r then we will print the pattern found at the location i and continues the search for next value of i.

B. *Trivial cases in comparison*

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

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

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

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

C. *This section describes different examples using our proposed approach (ISMPM) for the DNA sequences.*

Let *S=ACTTCAGGTCAACGATGTCA* be a string of 20 characters and *P=TCAA* be a pattern of 4 characters. The following table stores all the indexes of each character *A, C, G* and *T* in its corresponding row. The 0^{th} row stores the indexes of occurrences of the character *T*, 1^{st} row for *A*, 2^{nd} row for *G* and 3^{rd} row stores for *C*.

TABLE II. INDEX TABLE FOR DNA SEQUENCE

	DNA Sequence Indexes					
T 0	2	3	8	15	17	
A 1	0	5	10	11	14	19
G 2	6	7	13	16		
C 3	1	4	9	12	18	

As the first character starts from 0^{th} index of the sequence *S.* The first character in the pattern *P* is *T* so start search for *P* from the 0^{th} row (which stores the indexes of characters *T*). So the first index stored in 0^{th} row is 2 so the algorithm starts from 2^{nd} character in the string .It compares the first character in the pattern with the character of first index of the 0^{th} row in table.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The first character in the pattern matches to the starting index 2 of the 0^{th} row in *indexTab[4][20]*, so continue the match for second character.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The second character doesn't match from the 2^{nd} index so we stop the search for *P* in *S* from the index 2. The next index stored in the 0^{th} row of *indexTab[4][20]* is 3.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The first character is matched so it continues the search.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The second character of the pattern *P* also matches with the corresponding character of *S* from the starting index 3.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The third character too matches, so it continues the search.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The 4th character of *P* mismatches with the corresponding character of *S* from the starting index 3, so stop the search and go for the next index for another match. The next index stored in the 0th row of *indexTab[4][20]* which is the index 8.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The first character from the starting index 8 of the string *S* is matched with the first character of the pattern *P*.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The second character also matches and we continue our searching the next character.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The third character as well as the fourth character also matches with the pattern.

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

Finally all the characters match with the pattern *P* in *S* from the starting index of 8, so we print the pattern from the location 8. The next index stored in 0th row of *indexTab[4][20]* is 15.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The first character matches with the pattern *P*, so continue the searching for *P* in *S*.

S=A C T T C A G G T C A A C G A T G T C A
P=T C A A

The second character in *P* is not matching with the corresponding character from the index 15, so stop the search from the index 15.

S=A C T T CA G G T C A A CG A T G T C A
P=T C A A

The last index stored in the 0th row of *indexTab[4][20]* is 17 we need to start the search from the index 17, but it is skipped because the remaining characters from index 17 are only 3 characters. But the pattern has 4 characters, so it is not possible to find the pattern starting from the index 17, so stop the search from the index 17. Finally the search for *P* in *S* is completed, *P* occurred only one time in the string *S.*

D. The DNA sequence data has been taken from the Multiple Skip Multiple Pattern Matching algorithm MSMPMA [11] for testing the ISMPM algorithm. It explains large sequence data by taking a DNA biological sequence S∈ ∑ of size n=1024 and pattern P∈ ∑*. Let S be the following DNA sequence.*

AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAA
TAGTGTTACCAACTCGGGTGCCTATTGGCCTCCAAAAA
AGGCTGTTCAACGCTCCAAGCTCGTGACCTCGTCACTA
CGACGGCGAGTAAGAACGCCGAGAAGGTAAGGGAACT
AATGACGCGTGGTGAATCCTATGGGTTAGGATCGTGTC
TACCCCAAATTCTTAATAAAAAACCTAGGACCCCCTTCG
ACCTAGACTATCGTATTATGGACAAGCTTTAACTGTCGT
ACTGTGGAGGCTTCAAAACGGAGGGACCAAAAAATTTG
CTTCTAGCGTCAATGAAAAGAAGTCGGGTGTATGCCCC
AATTCCTTGCTGCCCGGACGGCCAGGCTTATGTACAAT
CCACGCGGTACTACATCTTGTCTCTTATGTAGGGTTCA
GTTCTTCGCGCAATCATAGCGGTACTTCATAATGGGAC
ACAACGAATCGCGGCCGGATATCACATCTGCTCCTGTG
ATGGAATTGCTGAATGCGCAGGTGTGAATACTGCGGCT
CCATTCGTTTTGCCGTGTTGATCGGGAATGCACCTCGG
GGACTGTTCGATACGACCTGGGATTTGGCTATACTCCA
TTCCTCGCGAGTTTTCGATTGCTCATTAGGCTTTGCGG
TAAGTAAGTTCTGGCCACCCACTTCGAGAAGTGAATGG
CTGGCTCCTGAGCGCGTCCTCCGTACAATGAAGACCG
GTCTCGCGCTAAATTTCCCCCAGCTTGTACAATAGTCC
AGTTTATTATCAAAGATGCGACAAATAAATTGATCAGCA
TAATCGAAGATTGCGGAGCATAAGTTTGGAAAACTGGG
AGGTTGCCAGAAAACTCCGCGCCTACTTTCGTCAGGAT
GATTAAGAGTATCGAGGCCCCGCCGTCAATACCGATGT
TCTTCGAGCGAATAAGTACTGCTATTTTGCAGACCCTTT
GCCAGGCCTTGTCTAAAGGTATGTTACTTAATATTGACA
ATACATGCGTATGGCCTTTTCCGGTTAACTCCCTG.

The Index Table (*indexTab[4][1024]*) for sequence *S* is very large to show here. For different patterns the number of occurrences and the number of comparisons is shown in the Table.III. For different patterns *P*'s the number of occurrences and the number of comparisons is shown in the below. To check whether the given pattern presents in the sequence or not we need an efficient algorithm with less comparison time and complexity. In general algorithms like Brute force or other conventional algorithms will take much time to do the searching process. The proposed ISMPM technique is one simple solution which gives better performance and less complexity. This algorithm can be appreciated for decreasing the number of comparisons as well as CPC ratio compared with some of the popular algorithms as shown in the next section.

IV. RESULTS AND DISCUSSIONS

Patterns of length from 1 to 20 are randomly selected from the input text and searched by the proposed method. 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. In Table.III we have compared and analyzed the results with different pattern sizes ranging from 1 to 20. Experimental results indicate that index based sequential multiple pattern matching algorithm gives very good performance related to the some of the existing methods. As we move towards the higher length of the pattern the number of comparisons decreases as shown in the below table.

TABLE III. EXPERIMENTAL RESULTS OF ISMPM ALGORITHM

S.No	Pattern(*P*'s)	Pattern Length	No.of Occr	ISMPM Comp	CPC Ratio
1	*A*	1	259	259	0.252
2	*AG*	2	53	518	0.505
3	*CAT*	3	11	542	0.529
4	*AACG*	4	5	614	0.599
5	*AAGAA*	5	2	627	0.612
6	*AAAAAA*	6	3	648	0.632
7	*AGAACGC*	7	2	596	0.582
8	*AAAAAAGG*	8	1	651	0.635
9	*GCTCATTAG*	9	1	584	0.570

10	*CCTTTTCCGG*	10	1	574	0.560
11	*TTTTGCCGTGT*	11	1	641	0.625
12	*TTCTTAATAAAA*	12	1	645	0.629
13	*GGGACCAAAAAAT*	13	1	581	0.567
14	*TTTTGCCGTGTTGA*	14	1	641	0.625
15	*CCTCCAAAAAAGGCT*	15	1	577	0.563
16	*GGCTGTTCAACGCTCC*	16	1	584	0.570
17	*TTTTCGATTGCTCATTA*	17	1	641	0.625
18	*GGGATTTGGCTATACTCC*	18	1	581	0.567
19	*GGCCTTGTCTAAAGGTATG*	19	1	589	0.575
20	*CCTGAGCGCGTCCTCCGTAC*	20	1	576	0.562

Table IV shows experimental results of different algorithms like MSMPMA, Brute-Force, Trie-Match, naïve string matching with the ISMPM algorithm. The performance of these algorithms are observed with two parameters namely number of comparisons and comparisons per character ratio (CPC).The results shows that ISMPM provides best performance.

TABLE IV. COMPARISON OF DIFFERENT ALGORITHMS

Pattern	No. of Occ	ISMPM		MSMPMA		Brute-Force		Tri-Match		Naïve String	
		No.of Com	*CPC*	*No.of Comp*	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*
A	259	259	0.2	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	53	518	0.5	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	11	542	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	5	614	0.5	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	2	627	0.5	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	1	651	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	1	645	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	1	584	0.5	1349	1.3	1349	1.3	1365	1.3	1349	1.3

Fig.1.is the graph for the comparison of different pattern matching algorithms with ISMPM. The current technique gives very good performance in reducing the number of comparisons when compared with some of the popular algorithms like MSMPMA, Brute-force, Trie-match and Naïve string search algorithms. The dotted line represents the ISMPM model where as MSMPMA, Brute-Force, Trie-matching and Naïve searching are shown by solid lines. As the size of the pattern increases the number of occurrences decreases and is equal to 1 in many cases shown in the above table III .From the experimental results shown above of pattern size 8 to 20 the occurrences is almost 1 in all the cases.

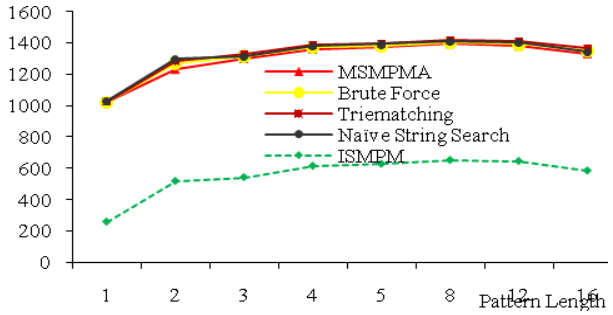

Figure 1. Comparison of different algorithms with ISMPM

The following are observed from the experimental results.

- Reduction in number of comparisons.
- The ratio of comparisons per character has gradually reduced and is less than 1.
- Suitable for unlimited size of the input file.
- Once the indexes are created for input sequence we need not create them again.
- It gives the good performance for DNA related sequences.

V. CONCLUSION

This study introduced a new ISMPM algorithm for pattern searching algorithm. This paper gives the most efficient method for determining DNA pattern matching sequences. It is a simple approach for finding the multiple patterns from a given sequence file. The proposed algorithm gives better performance compared with other algorithms. Based on the experimental work carried out with DNA sequence data, ISMPM approach gives good performance.

REFERENCES

[1] Berry, T. and S. Ravindran, 1999. A fast string matching algorithm and experimental results. In: Proceedings of the Prague Stringology Club Workshop '99, Liverpool John Moores University, pp: 16-28.

[2] Boyer R. S., and J. S. Moore, ''A fast string searching algorithm' Communications of the ACM 20, 762- 772, 1977.

[3] D.M. Sunday, A very fast substring search algorithm, Comm. ACM 33 (8) (1990) 132–142.

[4] Horspool, R.N., 1980. Practical fast searching in strings. Software practice experience, 10:501-506

[5] Knuth D., Morris. J Pratt. V Fast pattern matching in strings, SIAM Journal on Computing, Vol 6(1), 323-350, 1977.

[6] Kurtz. S, Approximate string searching under weighted edit distance. In proceedings of the 3rd South American workshop on string processing. Carleton Univ Press, pp. 156-170, 1996

[7] Needleman, S.B Wunsch, C.D(1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins." J.Mol.Biol.48,443-453.

[8] Raju Bhukya, DVLN Somayajulu,''An Index Based Forward backward Multiple Pattern Matching Algorithm, 'World Academy of Science and Technology..June 2010, pp347-355.

[9] Smith,T.F and waterman, M (1981). Identification of common molecular subsequences T.mol.Biol.147,195-197.

[10] Ukkonen,E., Finding approximate patterns in strings J.Algor. 6, 1985, 132-137.

[11] Ziad A.A Alqadi, Musbah Aqel & Ibrahiem M.M.EI Emary, Multiple Skip Multiple Pattern Matching algorithms. IAENG International. Vol 34(2), 200