2011 International Conference on Life Science and Technology
IPCBEE vol.3 (2011) © (2011) IACSIT Press, Singapore

Index Based Sequential Multiple Pattern Matching Algorithm Using Pair Indexing

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

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

***Abstract*—The current research in life science area is producing large amount of genetic data. DNA related pattern matching in a given sequence is one of the fundamental problems in computational biology. Biologists often search DNA sequences to identify use full information available from protein and genes for the functional and structural behavior. Many algorithms have been proposed but more efficient and robust methods are needed for the exact pattern matching algorithms. In the proposed work an index based sequential multiple pattern matching using pair indexing algorithm gives very good performance when compared with some of the existing algorithms. The current technique avoids unnecessary DNA comparisons as a result the number of comparisons and CPC ratio gradually decreases and overall performance increases.**

Keywords- Characters; matching; patterns; sequence.

I. INTRODUCTION

Pattern matching can be defined as finding the occurrences of a particular pattern of characters in a large text/sequence. An exact pattern matching involves identification of all the occurrences of a given pattern of *m* characters (*p=p1,p2,p3…pm*) in a text of *n* characters *(t=t1,t2,t3…tn)* built over a finite alphabet set ∑ of size σ.

Let *P={p1,p2,p3..pm}* be a set of patterns which are strings of nucleotide sequence characters from a fixed alphabet set ∑*={A,C,G,T}*. Let *T* be a large text consisting of characters in ∑. In other words *T* is an element of ∑*. 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 to the user and the developer to locate the occurrences of specific pattern in a sequence. So a new Indexing technique has been developed. In many cases most of the algorithm operates in two stages.

- Pre-processing phase or study of the pattern
- Processing phase or searching phase.

The pre-processing phase collects the full information and is used to optimize the number of comparisons. Whereas searching phase finds the pattern by the information collected in preprocessing. Depending upon the algorithm some of the algorithm uses pre-processing phase and some algorithm will search without it. Many pattern matching algorithms are available with their own merits and demerits based upon the pattern length and the technique they use. These pattern matching algorithms have been extensively used in various applications like information retrieval, information security, searching nucleotide sequences, amino acid sequences, and pattern matching in biological databases. String matching algorithms are classified into 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. 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]. Applications of exact pattern matching algorithms includes parsers, spam filters, digital libraries, screen scrapers, word processors, web search engines, natural language processing and computational biology.

Inexact pattern matching is sometimes referred as approximate pattern matching or matches with k mismatches/ differences. 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. Some Inexact pattern matching algorithms are Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. Applications of inexact pattern matching algorithms includes signal processing, computational biology and text processing.

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., IBMPMP algorithm for DNA sequence. Results and discussion are presented in Section IV and we make some concluding remarks in Section V.

BACKGROUND AND RELATED WORK

This section reviews some work related to DNA sequences. An alphabet set ∑ = {A, C, G, T} is the set of characters for DNA sequence which are used in the algorithm.

The following notations are used in this paper:
DNA sequence characters ∑= *{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.

Some of the pattern matching algorithm concentrates on pattern itself without using the preprocessing phase. Other algorithm compares the pattern and the text from left to the right. Some other algorithms perform comparison from right to left. The performance of the algorithm depends upon the order in which the comparison is done.

Several pattern matching algorithms have been developed to minimize the number of comparisons. Matching process is usually divided into two phases. The pre-processing phase and searching phase. Brute-force algorithm compares the pattern with the text from left to right. After each attempt it shifts the pattern by exactly one position to right. The time complexity is O(mn) in worst case and the number of text character comparison is 2n. Boyer-Moore algorithm [2] improves the performance by pre-processing the pattern by using two shift functions. The bad character shift and the good suffix shift. During the searching phase the pattern is aligned with the text and it is scanned from right to left. If a mismatch occurs the algorithm shifts the pattern with the maximum value taken between the two shift functions. The worst case complexity is O(mn).

In Index based forward backward multiple pattern matching technique[7] 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. In the MSMPMA [10] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. Index is used 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 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.

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. By using dynamic programming approach especially in DNA sequencing Needleman-Wunsch [6] algorithm and Smith-waterman algorithms [8] are more complex in finding exact pattern matching algorithm. By this method the worst case complexity is O(mn). The major advantage of this method is flexibility in adapting to different edit distance functions. 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.

Ukkonen [9] 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 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).

II. Index based Sequential Multiple Pattern Matching Algorithm Using Pair Indexing

In the proposed work it uses the indexes for the DNA sequence. It scans the DNA sequence from left to right and indexes are filled in their corresponding cells in an increasing order as they appear in the sequence. It has to search a pattern in a string whose alphabet set $\sum$ = {A, C, G, T}. Let the string be S of n characters represent in the DNA sequence and pattern P of m characters to be searched in string S. Instead of creating indexes on individual characters it creates indexes on pair of characters. There are 4 characters in our alphabet in set $\sum$. So there are 16 possible pairs. Let us call this $\sum p$ = {AA, AC, AT, AG, CA, CC, CT, CG, TA, TC, TT, TG, GA, GC, GT, GG}. It also maintains an array which stores the frequency of each pair p where $p \in \sum p$. After creating the index the algorithm searches for the pattern in the string using the pair pi in P with least count value. Using pairs is having an advantage which is the probability of finding a pair at a particular position is 1/16, whereas for individual characters it is 1/4, therefore when we use pair of characters as index which we are able to avoid more comparisons. Let $\sum^*$ be set of all possible strings with pair set $\sum p$. Then $S \in \sum^*$, |S| = n-1 where n is number of characters in S and |S| is number of pairs in S and |P| = m-1 where m is number of characters in P. Number of pairs is one less than number of characters.

A. 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 P in DNA.

Step 1: Integer arrays stab[16][n],sidx[16],pidx[16], sort[16]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Integer found:=1, ncmp:=0, npat:=0
Short integer pointers sp:=S, pp:=P;

Step 2: IF (m%2 == 1)

odd = 1;
Step 3: FOR i := 0; i < n; i++
stab[(*sp & 1536)>>9 | (*sp & 6)<<1][sidx[(*sp & 1536)>>9|(*sp & 6) << 1]++] = i;
increment sp by one byte
END FOR
Step 4: Sort array **sort** in ascending order according to values in **sidx**
Step 5: Store In **pidx** the frequency of character pairs found in p
Step 6: WHILE pidx[sort[k++]] = 0 && k < 16
DO
END DO
y:=sort[k-1]
Here **y** stores the least occurring pair in the string
Step 7: Store in **dif** the index of first occurrence of **y** in the array **p.**
Step 8: FOR i := 0; i <=sidx[y]; i++
found = 1;
k = stab[y][i] - dif;
sp := &S[k]
FOR j = 0; j< m/2; j++
ncmp+=2;
IF *sp != p[j]
found = 0;
break;
END IF
END FOR
IF odd = 1 && S[k+m] != P[m]
found := 0
IF found = 1
npat++;
PRINT "Pattern Found at location k occurrence no is : npat "
END IF
END FOR

The algorithm takes a string representing a DNA sequence as an input, and for given pattern it checks whether the pattern occurs in the string or not. As it first builds the table called *stab[16][n]*. In this table of indexes *stab[16][n]* it store the indexes of 16 possible pairs $\{p \mid p \in \sum_p \Lambda\, p \in S\}$. It also stores the count/occurrences of each pair p in the array *sidx[16]*. It uses pair $\{p_i \mid p_i \in P\}$ which has least count value *(sidx[i])* for searching to minimize the number of comparisons. The algorithm is suitable for DNA pattern matching because the numbers of possible character are less and the possible pairs are in $\sum_p$ = *{AA, AC, AT, AG, CA, CC, CT, CG, TA, TC, TT, TG, GA, GC, GT, GG}*.An indexing technique is used to reduce the pre-processing time and comparisons. For each pair it computes the array subscript value in *stab* by using binary operation on the pair stored at pointer *sp*.

((*sp) & 1536)>>9 | ((*sp) & 6)<<1

If pointer *sp* is pointing to the pair *CG* then 16 bit representation of value *sp i.e.*, (*sp) will be '01000111 01000011'. Here the first 8-bits *i.e.*, '01000111' are for letter *G* with ASCII value 71 and the last 8-bits '01000011' are for character *C* with ASCII value 67. The letters are in reverse order because in memory they are stored in reverse order (Little Endian form). Now when it applies the operation ((*sp)&1536)>>9 it gets 3 and by applying ((*sp)&6)<<1 it gets 4. It uses operator '|' on these two values i.e., '3 | 4' to get Array subscript 7, to store the index in *stab*. Here & is 'Binary and operator', << is 'Left shift operator', >> is 'right shift operator' and | is 'Binary or operator'.

TABLE I. ARRAY SUBSCRIPT VALUES FOR DNA CHARACTER

S.No	(*sp)	Binaryform (Little Endian form)	(*sp) & 1536)>>9	(*sp) & 6)<<1	Array Subscript
1	*AA*	01000001 01000001	0	0	0
2	*AC*	01000011 01000001	1	0	1
3	*AT*	01010100 01000001	2	0	2
4	*AG*	01000111 01000001	3	0	3
5	*CA*	01000001 01000011	0	4	4
6	*CC*	01000011 01000011	1	4	5
7	*CT*	01010100 01000011	2	4	6
8	*CG*	01000111 01000011	3	4	7
9	*TA*	01000001 01010100	0	8	8
10	*TC*	01000011 01010100	1	8	9
11	*TT*	01010100 01010100	2	8	10
12	*TG*	01000111 01010100	3	8	11
13	*GA*	01000001 01000111	0	12	12
14	*GC*	01000011 01000111	1	12	13
15	*GT*	01010100 01000111	2	12	14
16	*GG*	01000111 01000111	3	12	15

((*sp) & 1536)>>9 | ((*sp) & 6)<<1 always returns a unique subscript value in the range 0-15 for each pair $\{p \mid p \in \sum_p\}$ which is needed for subscripting 2D array of size *[16][n]*. The subscript values represents different pairs of characters as shown in the table II. So for each pair of character of string for the function (((*sp)&1536)>>9|((*sp) & 6)<<1)) directly references to its corresponding index in the 2D table *stab[16][n]*. The array *sidx[16]* stores the count of each character in the string.

B. Example

Initially let us take a DNA sequence *S=ACTTAGGCTCAATTCGATAGTTAGCATTCA* of 30 characters and *P = TTAG* of 4 characters. The following index table *stab* stores the indexes for each possible pair of characters. It stores the total number of occurrences of each pair in separate array *sidx*. It then uses a pair p_i in pattern *P* which occurs least number of times in the string *S* for searching. For above example the count value for pairs *TT* and *AG* are 4 and 3 respectively shown in table II. Count shows the number of times a pair has occurred in the given DNA sequence. Here *AG* occurs least number of times, so p_i = '*AG'* which is used for initial alignment. After that rest of the characters are compared in a sequential order for the pattern matching process.

TABLE II. INDEX VALUES AND COUNT FOR EACH PAIR

Pair	INDEXES				Count/Occ
AA	10				1

AC	**0**				**1**
AT	11	16	25		3
AG	4	18	22		3
CA	9	24	28		3
CC					0
CT	1	7			2
CG	14				1
TA	3	17	21		3
TC	8	13	27		3
TT	2	12	20	26	4
TG					0
GA	15				1
GC	6	23			2
GT	19				1
GG	5				1

In this technique it first aligns the pair '*AG'* in pattern *P* with '*AG'* in the DNA sequence *S*. After the alignment is completed it matches the characters sequentially. It stores the position of *AG* in the pattern *i.e.,*2. Go to the first occurrence of *AG* using the above table *i.e.,* 4 and align the string with pattern by subtracting 2 from the table value to find the starting pair *i.e.,* 4-2=2.

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

The first pair of character matches then it and compares the second pair in the pattern.

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

The second pair also of pattern *P* also matches with the sequence *S*. All the characters are matched, so the pattern is found at position 2. Now check the second index for *AG i.e.,* 18 and align *AG* in pattern to it 18-2 = 16.

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

The pair doesn't match, so go to the third index in table for *AG i.e.,* 22 and align *AG* in pattern to it 22 – 2 = 20.

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

The first pair of character matches then it compares the second pair in pattern.

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

The second pair also matches with the sequence .All the characters are matched, so the pattern is found at position 20. No more index values for *AG* are there so algorithm stops. The number of occurrences found is 2.

C. Trivial Cases in Comparisons

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.

D. The DNA sequence data has been taken from the Multiple Skip Multiple Pattern Matching algorithm MSMPMA[10] for testing the IBMPMP 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.*

"AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAGTG
TTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGCTGTTC
AACGCTCCAAGCTCGTGACCTCGTCACTACGACGGCGAGTA
AGAACGCCGAGAAGGTAAGGGAACTAATGACGCGTGGTGAA
TCCTATGGGTTAGGATCGTGTCTACCCCAAATTCTTAATAAAA
AACCTAGGACCCCCTTCGACCTAGACTATCGTATTATGGACA
AGCTTTAACTGTCGTACTGTGGAGGCTTCAAAACGGAGGGAC
CAAAAAATTTGCTTCTAGCGTCAATGAAAAGAAGTCGGGTGT
ATGCCCCAATTCCTTGCTGCCCGGACGGCCAGTTCATAATGG
GACACAACGAATCGCGGCCGGATATCACATCTGCTCCTGTG
ATGGAATTGCTGAATGCGCAGGTGTGCTTATGTACAATCCAC
GCGGTACTACATCTTGTCTCTTATGTAGGGTTCAGTTCTTCGC
GCAATCATAGCGGTACGAATACTGCGGCTCCATTCGTTTTGC
CGTGTTGATCGGGAATGCACCTCGGGGACTGTTCGATACGA
CCTGGGATTTGGCTATACTCCATTCCTCGCGAGTTTTCGATT
GCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCACCCACTT
CGAGAAGTGAATGGCTGGCTCCTGAGCGCGTCCTCCGTACA
ATGAAGACCGGTCTCGCGCTAAATTTCCCCCAGCTTGTACAA
TAGTCCAGTTTATTATCAAAGATGCGACAAATAAATTGATCAG
CATAATCGAAGATTGCGGAGCATAAGTTTGGAAAACTGGGAG
GTTGCCAGAAAACTCCGCGCCTACTTTCGTCAGGATGATTAA
GAGTATCGAGGCCCCGCCGTCAATACCGATGTTCTTCGAGC
GAATAAGTACTGCTATTTTGCAGACCCTTTGCCAGGCCTTGT
CTAAAGGTATGTTACTTAATATTGACAATACATGCGTATGGCC
TTTTCCGGTTAACTCCCTG" of a string of size 1024 and contains 1024 characters $c \in \sum$ and 1023 pairs of characters $p \in \sum_p$. For different patterns *P* the number of occurrence, the number of comparisons made and *CPC* ratio are shown in the following table III.

III. Results and Discussions

For different patterns *P*'s the number of occurrences and the number of comparisons of the proposed algorithms IBMPMP is shown in the Table.III. To check whether the given pattern is present in the sequence or not it needs an efficient algorithm with less comparison time and low complexity. By the current technique different patterns are analyzed and the graph is plotted by using these results. From the below experimental results, improvement can be seen that IBMPMP algorithm gives good performance compared to the some of the popular methods. It has taken different pattern sizes ranging from 2 to 20 of DNA sequences, as the pattern size increases the number of comparisons decreases in case of IBMPMP algorithm shown

in the Table III. Here the number of comparisons and CPC ratio decreases and is less than 1.

TABLE III. EXPERIMENTAL RESULTS OF IBMPMP ALGORITHM

S.No	Pattern(P's)	No of char	No. of occur	No. of comp	CPC Ratio
1	*AG*	2	53	106	0.10
2	*CAT*	3	11	100	0.09
3	*AACG*	4	5	140	0.14
4	*AAGAA*	5	2	136	0.13
5	*AAAAAA*	6	3	362	0.35
6	*AGAACGC*	7	2	224	0.22
7	*AAAAAAGG*	8	1	146	0.14
8	*GCTCATTAG*	9	1	132	0.13
10	*TTTTGCCGTGT*	11	1	142	0.14
11	*TTCTTAATAAAA*	12	1	142	0.14
12	*GGGACCAAAAAAT*	13	1	260	0.25
13	*TTTTGCCGTGTTGA*	14	1	144	0.14
14	*CCTCCAAAAAAGGCT*	15	1	116	0.11
15	*GGCTGTTCAACGCTCC*	16	1	264	0.26
16	*TTTTCGATTGCTCATTA*	17	1	124	0.12
17	*GGGATTTGGCTATACTCC*	18	1	266	0.26
18	*GGCCTTGTCTAAAGGTATG*	19	1	130	0.13
19	*CCTGAGCGCGTCCTCCGTAC*	20	1	126	0.12

It has been observed the following in terms of relative performance of our algorithm with the existing algorithms. From the experimental results shown in table IV performance is checked with the existing algorithms. The proposed algorithm gives good performance in two parameters like CPC ratio and number of comparisons with the algorithms like MSMPMA, Brute-force, Tri-Match, Naïve string matching algorithms.

TABLE IV. COMPARISON OF DIFFERENT ALGORITHMS

Pattern	**IBMPMP**		**MSMPMA**		**Brute-Force**		**Tri-Match**		**Naïve String**	
	No.of Com	***CPC***	***No.of Comp***	***CPC***	***No.of Comp***	***CPC***	***No.of Com***	***CPC***	***No.of Com***	***CPC***
AG	106	0.1	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	100	0.1	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	140	0.1	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	136	0.1	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	362	0.1	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	142	0.1	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	264	0.2	1349	1.3	1349	1.3	1365	1.3	1349	1.3

Fig.1. Shows the experimental results of the IBMPMP with MSMPMA, Brute-Force, Trie-Match and Naïve String search algorithms. The comparison is done between different algorithms in case of number of comparison of different pattern matching algorithms with IBMPMP. By using the current technique it is clear that proposed algorithm outperforms when compared with some of the existing techniques and reduces the number of comparisons. The dash line shows the IBMPMP model and the MSMPMA, Brute Force, Trie-matching and Naïve searching are shown by solid lines.

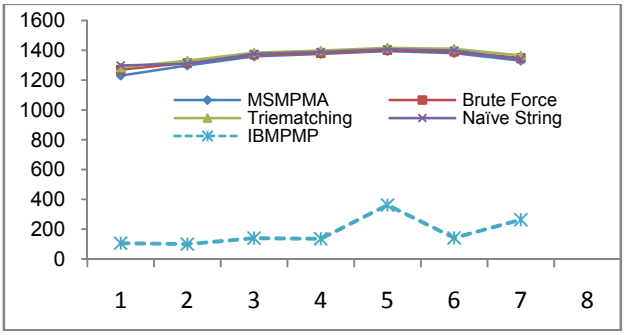

Fig 1.Comparison of different algorithms with IBMPMP

From the above results the following are observed from the experimental results. The number of comparison gradually reduces and the CPC ratio decreases nearly to half when compared with the available techniques. The current algorithm works for the input file of unlimited size. It gives good performance for DNA related sequence applications due to less number of characters.

IV. CONCLUSION

We have proposed a new pattern matching technique called IBMPMP algorithm for DNA sequences. Here we introduced the concept of indexing which gives the efficient method for solving DNA pattern matching problems. It is simple and straight forward approach for finding the multiple patterns from a given sequence file. The proposed algorithm gives very good performance with the other algorithms. Based on the experimental work our approach provides good performance related to the DNA sequence data.

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, pp.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] 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.

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

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

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

[10] Ziad A.A Alqadi, Musbah Aqel & Ibrahiem M.M.EI Emary, Multiple Skip Multiple Pattern Matching algorithms. IAENG International Journal of Computer Science, Vol 34(2), 2007.