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

An Index Based Sequential Multiple Pattern Matching Algorithm Using Least Count

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*- Pattern matching is one of the emerging areas in computational biology. Comparative biology is a central research in many sequence studies. Searching DNA related data is a common activity for molecular biologists to retrieve necessary and use full information from the sequences. But most of the techniques do not provide straight forward methods for searching DNA sequences. In this paper we introduce a new pattern matching technique called index based sequential multiple pattern matching algorithm using least count. The present technique avoids unnecessary DNA comparisons as a result the number of comparisons and CPC ratio gradually decreases and overall performance increases.**

Keywords*- *Characters; patterns; sequence.

I. INTRODUCTION

Biologists often interested in searching DNA related sequences to identify well defined information from the proteins and genes. As the size of the data increases it is more difficult for the users to retrieve the important information from the DNA sequence within less amounts of time and complexity. Many different pattern matching techniques has been developed but more efficient and robust methods are needed.

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. These algorithms are the most extensive problems in computer technologies during past two decades. 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 .Modern world of bioinformatics has large applications which include text editors, search engines, molecular medicine, industry, agriculture and comparative biology. String matching algorithms are mainly classified into two categories.

Exact and Inexact 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 [1], Knuth-Morris-Pratt Algorithm [2]. 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.*, IBMPMC 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 ∑ = *{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 ∑= *{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.

In the MSMPMA [7] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. Here 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 from 1 to m-1). 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 Knuth-Morris-Pratt algorithm [2] 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 [3] algorithm and Smith-waterman algorithms [5] 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.

The Bayer-Moore algorithm [1] applies larger shift-increment for each mismatch detection. The main difference in 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 be 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)*.While 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.

Ukkonen [6] proposed automation method in 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[1] 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.

III. INDEX BASED SEQUENTIAL MULTIPLE PATTERN MATCHING ALGORITHM USING LEAST COUNT.

The most common approach is to improve efficiency which involves the idea of indexing method where the number of comparisons are reduced when compared with different other algorithms. So a new index based algorithm is proposed. In such approach the characters are indexed according to their indexes as they occur in the text/sequence. The efficiency and performance highly depends upon the character size. The objective of the work is to find the patterns from the sequence file of large size. Many different solutions have been proposed to bring the optimal results in exact matching sequence data but gets inaccurate and slow results. Latest computational technology uses fast algorithms made relatively easy in bringing accurate results.

In the proposed work indexes has been used for the DNA 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 and the pattern *P* of *m* characters. After creating the index the algorithm will search for the pattern in the string using the index of least occurring character in the string. The index based algorithm uses a table called stab[4][n] which stores all the indexes of each character in its corresponding vector.

A. Algorithm

```
Input:   String S of n characters and a pattern P of m characters,
where S, P ∈ ∑*.
Output: The no. of occurrence and the positions of P in S.
Step 1:  Integer arrays stab[4][n], sidx[4], pidx[4], sort[4] = {0,
1, 2, 3}
         Integer found:=1, ncmp:=0, npat:=0
Step2:   FOR i := 0; i < n; i++
        stab[(DNA[i] - 64) % 5][sidx[(DNA[i]-64)%5]++] = i;
         END FOR
Step 3:Sort array sort in ascending order according to values in
sidx
Step 4:  WHILE pidx[sort[k++]]  =  0 && k < 4
         DO
         END DO
         y:=sort[k-1]
Step 5:Store the index of first occurrence of y in the array pin dif
Step 6:  FOR i := 0; i <=sidx[y];  i++
                 found = 1;
                 k = stab[y][i] - dif;
                 x = k;
                 FOR j = 0;  j< m;  j++
                         ncmp++;
                         IF DNA[k]  !=  p[j]
                         found = 0;
                         break;
                         END IF
                         k++;
                 END FOR
                 IF found  = 1
                         npat++;
PRINT  "Pattern Found at location x occurrence  is : npat "
                 END IF
         END FOR
```

Basically IFBMPM[4] checks for each first occurrence of the first character of pattern in the algorithm and compares one character from left and one character from right until all characters are compared, if all characters matches to the pattern then 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. This process continues till the two pointers from left as well as right pointer exchanges their position in opposite direction.

In the current algorithm an extra field has been added to the IFBMPM [4] which is used to search the pattern in a sequential order to reduce the number of comparisons and CPC ratio when compared with IFBMPM [4]. The count value always increments as it scans from the DNA sequence for the characters *A,C,G* and *T* .So the count value will be indicating the least count which is present either in *A,C,G* or *T*. The least count value will be taken into the consideration for the pattern comparison. If the least count values character is not present in the pattern then it considers for the next least count value available in the index table.

The algorithm is suitable for DNA pattern matching because the numbers of possible character are less. The characters are in $\sum = \{A, C, G, T\}$.The ASCII indexing technique is used to reduce the pre-processing time and comparisons. For each character in $\sum$ it computes array subscript value by using the following technique *[(S[i]-64)%5]*.

Table.I. array subscript values for DNA characters.

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 *[(S[i]-64)%5]* always returns a subscript value in the range 0, 1, 2, 3 which is needed for subscripting 2D vector of size [4][n]. The subscript values 0, 1, 2, 3 represents characters *T, A, G, C* respectively. So for each character of string in the function *[(S[i]-64)%5]* directly references to its corresponding vector in the 2D vector table *stab[4][n]*. The vector *sidx[4]* stores the count of each character of the string. Here it will use the character which has least count to find the pattern in the string from the index table. This will reduce the number of comparisons needed to find and occurrences of the pattern in the string.

A. B. 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.

C. Example

Take a string S=ACTTAGGCTCAACGATGTTAGCATC of 25 characters and *P = TTAG* of 4 characters. The index table stores all the indexes of each character *A,C,G* and *T* in its corresponding row as it occurs in the sequence. The 0th row stores the indexes of occurrences of the character *T*, 1st row for *A*, 2nd row for *G* and 3rd row for *C*. It stores the total number of occurrences of each character in separate array. The comparisons will start from the character in pattern *P* which is occurring least number of times in the string. For the sequence *S* the element *T, A, G* and *C* are occurring 7, 7, 5, 6 times respectively. Here *G* is occurring least no of times so *G* is used as the initial alignment and once if there is a match of pattern *G* with the sequence character *G* then rest of the characters will be compared in a sequential order for the pattern matching process.

Table. II. INDEX values for A,C,T and G

	DNA Text/Sequence Indexes							**Count**
T 0	2	3	8	15	17	18	23	7
A 1	0	4	10	11	14	19	22	7
G 2	**5**	**6**	**13**	**16**	**20**			**5**
C 3	1	7	9	12	21	24		6

In this technique the character *G* in pattern *P* is aligned with *G* in the DNA sequence *S*. Once the alignment is completed then it will match the characters sequentially one by one from the starting character of the given pattern. In the index table the occurrence of the *G* character is only 5 *i.e.,* least count is 5 so *G* will be used for matching process. Here the table has an extra field called count which increments every time as the character occurs in the sequence. The count helps to find the least count character which is available from the index table, so only those many comparisons can be done. The algorithm maps to the first occurrence of *G* according to the table and then starts comparing the first element of pattern with the possible match in the string relative to that *G*.

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

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

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

The second character also matches then it compares the third character.

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

The third character too matches then it compares fourth *i,e.,* final character.

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

All the characters are matched, so the pattern is found at position 2. Now go to the second index in index table for *G* and align *G* to it.

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

After the alignment first character matches then check for the second character.

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

Second character is not matched. Now jump to the third index stored in table for *G* and start matching.

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

First character in the pattern *P* is not matched with sequence *S*. Now go to the fourth index stored in table for *G* and start matching relative to it.

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

Here first character is not matched. Now go to the fifth index stored in table for *G* and start matching.

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

The first character matches then it compares the second character of the pattern to the corresponding character.

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

The second character also matches then it compares the third character.

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

The third and fourth character of the pattern *P* is also matches with sequence *S*. So the pattern is found at the index position 17.

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

All the characters are matched, so the pattern is found at position 17. No more *G* is available in the index table corresponding to *G* so the algorithm ends. The total number of patterns occurred is from the sequence is 2.

D. The DNA sequence data has been taken from the Multiple Skip Multiple Pattern Matching Algorithms MSMPMA [7] for testing the ISMPMC 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 as the string S of 1024 character in ∑.*

"AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAG
TGTTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGCT
GTTCAACGCTCCAAGCTCGTGACCTCGTCACTACGACGGC
GAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGACGCG
TGGTGAATCCTATGGGTTAGGATCGTGTCTACCCCAAATT
CTTAATAAAAAACCTAGGACCCCCTTCGACCTAGACTATC
GTATTATGGACAAGCTTTAACTGTCGTACTGTGGAGGCTT
CAAAACGGAGGGACCAAAAAATTTGCTTCTAGCGTCAAT
GAAAAGAAGTCGGGTGTATGCCCCAATTCCTTGCTGCCCG
GACGGCCAGGCTTATGTACAATCCACGCGGTACTACATCT
TGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAATCATAGC
GGTACTTCATAATGGGACACAACGAATCGCGGCCGGATA
TCACATCTGCTCCTGTGATGGAATTGCTGAATGCGCAGGT
GTGAATACTGCGGCTCCATTCGTTTTGCCGTGTTGATCGG
GAATGCACCTCGGGGACTGTTCGATACGACCTGGGATTTG
GCTATACTCCATTCCTCGCGAGTTTTCGATTGCTCATTAGG
CTTTGCGGTAAGTAAGTTCTGGCCACCCACTTCGAGAAGT
GAATGGCTGGCTCCTGAGCGCGTCCTCCGTACAATGAAGA
CCGGTCTCGCGCTAAATTTCCCCCAGCTTGTACAATAGTC
CAGTTTATTATCAAAGATGCGACAAATAAATTGATCAGCA
TAATCGAAGATTGCGGAGCATAAGTTTGGAAAACTGGGA
GGTTGCCAGAAAACTCCGCGCCTACTTTCGTCAGGATGAT
TAAGAGTATCGAGGCCCCGCCGTCAATACCGATGTTCTTC
GAGCGAATAAGTACTGCTATTTTGCAGACCCTTTGCCAGG
CCTTGTCTAAAGGTATGTTACTTAATATTGACAATACATG
CGTATGGCCTTTTCCGGTTAACTCCCTG"

The Index Table (S*Tab[4][n]*) 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 by taking patterns from 1 to 20 of different cases.

IV. EXPERIMENTAL RESULTS

For different patterns *P*'s the number of occurrences and the number of comparisons of two algorithms ISMPMC and IFBMPM[4] 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 proposed method different patterns are analyzed and the graph is plotted by using these results. As it is clear that ISMPMC outperforms very well when compared with some of the other popular existing algorithms.

TABLE.III. RESULTS OF ISMPMC WITH IFBMPM

Pattern(P's)	No.of char	No.of Occu	No.of comp IFBMPM	CPC ratio IFBMPM	No.of Comp ISMPMC	CPC ratio ISMPMC
A	1	259	518	0.505	259	0.253
AG	2	53	624	0.609	300	0.293
CAT	3	11	567	0.553	542	0.529
AACG	4	5	614	0.599	318	0.311
AAGAA	5	2	616	0.601	357	0.349
AAAAAA	6	3	627	0.612	647	0.632
AGAACGC	7	2	600	0.585	342	0.334
AAAAAAGG	8	1	634	0.619	368	0.359
GCTCATTAG	9	1	582	0.568	399	0.390
CCTTTTCCGG	10	1	562	0.548	584	0.570
TTTTGCCGTGT	11	1	650	0.634	350	0.342
TTCTTAATAAAA	12	1	651	0.635	382	0.373
GGGACCAAAAAAT	13	1	579	0.565	336	0.328
TTTTGCCGTGTTGA	14	1	638	0.623	353	0.345
CCTCCAAAAAAGGCT	15	1	578	0.564	589	0.575
GGCTGTTCAACGCTCC	16	1	598	0.583	356	0.348
TTTTCGATTGCTCATTA	17	1	643	0.627	359	0.351
GGGATTTGGCTATACTCC	18	1	598	0.583	347	0.339
GGCCTTGTCTAAAGGTATG	19	1	579	0.565	361	0.353
CCTGAGCGCGTCCTCCGTAC	20	1	570	0.556	592	0.578

Table. IV shows experimental results of different algorithms like IFBMPM, MSMPMA, Brute-Force, Trie-Match, naïve string with the ISMPMC algorithm. The performances of these algorithms are observed with two parameters namely number of comparisons and (CPC) ratio. The ISMPMC results gradually decreases and is nearly half when compared with some of the popular techniques. The results show that ISMPMC provides best performance and reduces gradually over all the other method. The present algorithm when compared with IFBMPM algorithm shows improvement in number of comparisons in almost all cases. It avoids many comparisons by using least occurring character.

TABLE .IV.COMPARISONS OF DIFFERENT ALGORITHMS

Pattern	ISMPMC		IFBMPM		MSMPMA		BRUTE-FORCE		TRI-MATCH		NAÏVE STRING	
	No.of Com	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*	*No.of Com*	*CPC*
A	259	0.2	518	0.5	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	300	0.2	624	0.6	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	542	0.5	567	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	318	0.3	614	0.5	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	357	0.3	616	0.6	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	368	0.3	634	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	382	0.3	651	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3

Fig.1. Shows comparison of different pattern matching algorithms with ISMPMC. By using the current technique it is clear that proposed algorithm outperforms when compared with some of the existing techniques and gives very good performance in reducing the number of comparisons when compared with other popular algorithms. The dash line shows the ISMPMC model and the IFBMPM, MSMPMA, Brute-Force, Trie-matching and Naïve searching are shown by solid lines.

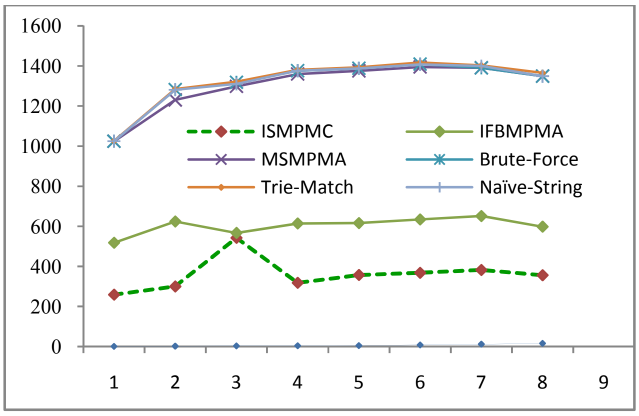

Fig 1. Comparison of different algorithms with ISMPMC

The following are observed from the experimental results

- Reduction in number of comparisons.
- The CPC ratio decreases gradually over other methods.
- Suitable for unlimited size of the input file.
- Once the indexes are created for input sequence it doesn't need to create them again.
- It gives good performance for DNA related sequence applications due to less number of characters.

V. CONCLUSION

A new ISMPMC algorithm for pattern searching is proposed. This paper gives the most efficient method for solving DNA pattern matching problems. It is very simple and straight approach for finding the multiple patterns from a given 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 dataset.

References

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

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

[3] 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.

[4] Raju Bhukya, DVLN Somayajulu,''An Index Based Forward backward Multiple Pattern Matching Algorithm, 'World Academy of Science and Technology . June 2010, pp. 347- 355.

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

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

[7] 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.