International Journal of Biological, Life and Agricultural Sciences
ISSN: 2415-6612
Vol:4, No:6, 2010

An Index based Forward Backward Multiple Pattern Matching Algorithm

Raju Bhukya, DVLN Somayajulu

Abstract**—**Pattern matching is one of the fundamental applications in molecular biology. Searching DNA related data is a common activity for molecular biologists. In this paper we explore the applicability of a new pattern matching technique called Index based Forward Backward Multiple Pattern Matching algorithm(IFBMPM), for DNA Sequences. Our approach avoids unnecessary comparisons in the DNA Sequence due to this; the number of comparisons of the proposed algorithm is very less compared to other existing popular methods. The number of comparisons rapidly decreases and execution time decreases accordingly and shows better performance.

Keywords**—**Comparisons, DNA Sequence, Index.

I. Introduction

PATTERN matching is an important and active research with large applications. 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*. As the size of the data grows it becomes more difficult for users to retrieve necessary information from the sequences. There are various kinds of tools available for the comparison which provides exact as well as approximate pattern matching. Hence more efficient 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 characters from a fixed alphabet set called $\sum = \{A, C, G, T\}$. Let *T* be a large text consists of characters in $\sum$ denoted as $\sum^*$. The problem is finding all the occurrences of *p* in *T*. It is important application widely used in data filtering to find selected patterns, in security applications, and used in DNA search. Many existing real time pattern matching algorithms are reviewed and classified in two categories.

1) Exact string matching algorithms
2) Approximate string matching algorithms.

Exact string matching algorithm means 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 match is exact one, meaning that the exact word or pattern is found.

Raju Bhukya is with the National Institute of Technology, Warangal, India. He is now with the Department of Computer Science and Engineering (Phone: +91-9700 5539 22; fax: 0091-8702459547; e-mail: raju@nitw.ac.in).

Dr. DVLN Somayajulu is with the National Institute of Technology, Warangal, India. He is now with the Department of Computer Science and Engineering (Phone: +91-9849 3365 47; fax: 0091-8702459547; email: soma@nitw.ac.in).

In Unix environment there is a useful command utility called "grep" [6] which allows user to search globally for lines matching the regular expression, and print them. Some exact matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm, Knuth-Morris-Pratt Algorithm[1],[2]. These pattern matching algorithms can be applied to find patterns in **DNA Sequences**.

Inexact/Approximate string matching: Inexact 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 character in similarity 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. Due to the possible DNA mutation the biological inference does not expect an identical match but rather a high sequence similarity usually implies significant functional or structural functionality. The field of bioinformatics has many applications in the modern day world includes text editors, search engine, molecular medicine, industry, agriculture and Comparative biology. In many information retrieval systems it is necessary to locate one or more patterns quickly.

Pattern matching algorithms have two main objectives.

1) Reduce the number of character **comparisons** required in the worst and average case analysis.
2) Reducing the time requirement in the worst and average case analysis.

The proposed work is based on an IFBMPM model for **DNA Sequence**. In this model input file is scanned from left to right until end of the file. The character **indexes** are stored in the 2D vector called **index** table. In the current model when we need to search some pattern *P* in text *S*, we start the search from the **indexes** stored in the row of **index** table which corresponds to the first character of the pattern *P*. If any

character mismatches in its position, we skip the search and go for the next **index** which corresponds to the first character of the pattern *P* according to the **indexes** stored in **index** table for matching. This process continues to search for *P* to the end of text *S*. By using the IFBMPM method, the number of **comparisons** and **comparisons** per character ratio (*CPC*) decreases when compared with some of the existing algorithms MSMPMA [8].

The rest of the paper is organized as follows. We briefly reviewed the background and related work in the section 2. In section 3 we provided a proposed model *i.e.*, IFBMPM and related algorithm for **DNA Sequence**. Results and discussion are presented in section 4. Section 5 concludes the paper.

II. Background and Related Work

In this section we review some work related to **DNA Sequences**. An alphabet set $\sum = \{A, C, G, T\}$ is the set of characters for **DNA Sequence** which used in this algorithm.

The following notations are used in this paper:

DNA Sequence characters $\sum = \{A, C, G, T\}$
ϕ denotes 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 – Comparisons per character.

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 be able to locate the occurrences of specific pattern in a sequence. In this section we discuss about these different types of string matching methods. Some of the exact string matching algorithms available, such as Naïve string search, Brute-force algorithm, Bayer-Moore algorithm, Knuth-Morris-Pratt algorithms [1], [2].

In Brute-force algorithm the first character of the pattern *P* is compared with the first character of the string *T*. If it is match, 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 [1] applies larger shift-increment for each mismatch detection. A main modification to the Naïve algorithm is the matching of pattern *P* and string *T* is done from right to left *i.e.*, after aligning *P* and string *T* the last character of *P* will matched to *T* first. 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 is $O(m+n)$ and the average case complexity is *O(n/m).* Although Knuth Morris-Pratt [2] algorithm has better worst case running time than the Boyer-Moore algorithm.

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 [4] algorithm and Smith-waterman algorithms 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.

In 1996 Kurtz [3] proposed another way to reduce the space requirements of almost *O(mn).* The idea was to build only the states and transitions actually reached in the processing of the text. The automation starts at just one state and transitions are built as they are needed. The transitions those were not necessary will not be build. Wu.S.Manber and Myer.E [7] proposed the algorithm for approximate limited expression matching, and Wu.S.Manber.U [6] proposed the algorithm for fast text searching allowing errors.

Ukkonen [5] 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.

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.

TABLE I
STRING MATCHING ALGORITHMS SUMMARY

Algorithm Name	Author	Comparison Order	Preproc-essing	Searching Time Complexity
Boyer Moore	R.S. Boyer and J.S. Moore	From right to left	Yes	*O(mn)*
Horspool	Nigel Horspool	Is not relevant	Yes	*O(mn)*
Brute force	-	Is not relevant	No	*O(mn)*
Kunth Morris Pratt	Michael O Rabin and Richard M Karp	From left to right	Yes	*O(n+m) independent from the alphabet size*
Quick Search	Sunday	Is not relevant	Yes	*O(mn)*
Karp Rabin	Michel O Rabin and Richard M Karp	From left to right	Yes	*O(mn)*
Zhu Takaoka	R. F. Zhu and T Takaoka	From right to left	Yes	*O(mn)*
Index Based	IFBMPM Model	From left to right	Yes	*O(mn)*

III. PROPOSED WORK

In the proposed work we use the **indexes** for the **DNA Sequence** belongs to ∑*. It is scanned from left to right and filled in their corresponding **indexes**. To search a pattern *P* in a string *S* whose alphabet set ∑. Let the string be *S* of having *n* characters and the pattern *P* of having *m* characters.

To search a pattern in a string whose alphabet set ∑= {A, C, G, T}. Let the string be *S* of having *n* and the pattern *P* of having *m* characters. Then *S*, $P \in \sum^*$, $|S| = n$ and $|P| = m$. Generally $|P| \leq |S|$ *i.e.*, $m \leq n$.

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 *S*.

Algorithm:

Step1: Integer arrays indexTab[4][n], charIndex[4]
Integer found:=1, n_occ:=0,n_cmp:=1;

Step2: FOR i:=0;i<n;i++
indexTab[(S[i]-64)%5][charIndex[(S[i]-64)%5]++]:=i;
End FOR

Step 3: FOR i:=0;i<chatIndex[(P[0]-64)%5];i++
found:=1;
IF i+m-1 > n-1
found:=0
SKIP the test, GOTO step 4.
End IF
FOR r:=0;r ≤ m/2;r++
n_cmp++;
IF P[r]=s[r+i]
n_cmp++;
IF P[m-r-1]=S[i+m-r-1]
DO Nothing
ELSE
found:=0
End IF
ELSE
found:=0
End IF
End FOR
Step 4:IF found:=1
n_occ++
PRINT "Pattern Found At Location i, Occurrence no is: n_occ"
End IF
End FOR

This algorithm first takes a string as input, and for each given pattern it checks whether the pattern occurs in the string or not. If the pattern occurs in the string it prints pattern with its starting position in the string.

It first builds up a table called **index** table, which is useful to reduce the number of **comparisons**. Once the **index** table is created for a string it is used for all the different patterns. For each pattern we start checking from the first character **indexes** of the pattern with using the **index** table which reduces the unnecessary **comparisons**.

The **index** based algorithm for multiple pattern matching uses a table(2D vector) called indexTab[4][n]. The basic idea used here is to store all the **indexes** of each character in its corresponding vector. The algorithm is suitable for biological applications such as DNA pattern matching which needs to compare two strings of the characters in ∑= {A, C, G, T}. i.e., the **index** of each occurrence of A is stored in indexTab[4][n] corresponding to the **index** of A in indexTab[4][n].The ASCII indexing technique is used here to reduce the preprocessing time and pre processing time **comparisons**. For each character in ∑ we compute its array subscript value in indexTab by using the following technique.

TABLE II
COMPUTING ARRAY SUBSCRIPT VALUES FOR DNA CHARACTERS.

Character	ASCII Value	ASCIIValue-64	(ASCIIValue-64)%5 (or) [*(S[i]-64)%5*]
A	65	1	1
C	67	3	3
G	71	7	2
T	84	20	0

From Table II, [*(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 the characters T, A, G, C respectively. So for each character of string the function *(S[i]-64)%5* directly references to its corresponding vector in the 2D vector indexTab[4][n]. The vector charIndex[4] stores the counter value of each occurrence of each character with reference to [*(S[i]-64)%5*].

TABLE III
ARRAY SUBSCRIPT VALUES AND THEIR CORRESPONDING CHARACTERS.

Array Subscript Value	Character
0	*T*
1	*A*
2	*G*
3	*C*

For each first occurrence of the first character of pattern this algorithm compares one character from left and one character from right until all characters are compared, if all characters matches to the pattern it prints the pattern found from the starting location. If any character mismatches it skips the test and continues to check the next occurrence of the first character of the pattern.

B. Mathematical Analysis

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 be denote the character at the *i* th location in the string(or pattern) *X*. Now for each value of *i*, we check

Whether $P_r = S_{i+r}$ if it so we will check $P_{m-r-1} = S_{i+m-r-1}$ for $r=0, 1, 2,\ldots, m/2$.

If these two **comparisons** are true until $r \leq m/2$ then we will print Pattern Found at the Location *i*.

And we continue the search for next value of *i*.

C. 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.

D. Example 1:

Let us take a string *S= ACTTAGGCTCAACGATGTTAGCATC* of 25 characters and *P=TTAG*.

The following **index** table stores all the **indexes** of each character A, C, G and T in its corresponding row. 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.

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

The first character in the pattern P is *T* so we start search for P from the 0th row(which stores the **indexes** of character *T*). The first **index** stored in 0th row is 2 so we start the algorithm from 2nd character in the string, the searching process is shown below.

The algorithm first compares the first character in the pattern with the character of first **index** of the 0th row in table.

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 last character of the pattern to the corresponding character in the string.

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
S= T T A G

The last character is matched then it compares the second character from the left.

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

Again it continues matching for 2nd character from the right.

*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***

Now all the character matches it prints the message the pattern found at the location 2 in the string.The 2nd **index**

stored in the 0th row of the above table is 3, we start search again from the **index** 3 of the string.

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 from the left is matched, then it checks for the match of the first character from the right.

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

It is also matched then it compares the second character from the left.

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*
S= T T A G

Here the match failed for 2nd character from the left, it stop the search and then continues the search from the next **index** stored in the 0th row of the table. The third **index** stored in the 0th row is 8 so we start search from the **index** 8 of the string S.

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 from the left is matched then it compares the first character from the right.

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 match failed for the first character from the right so it skip the test from the starting **index** 8. The next **index** stored in the 0th row of the **index** table is 15 so we start search from 15.

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

Clearly the first character from the left is matched. So we compare the first character from the right.

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 match failed at this point, so we skip the test from the **index** 15. Again continues from the next **index** stored in the 0th row of the **index** table which is 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

The first character from left is matched, then we compare the first character from the right.

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

It is also matched we continue search from the second character from the left.

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

It is also matched so compare the second character from the right.

*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 from the location 17, so we print the message Pattern found at the location 17, and continues the search for P from the next **index**18 in the 0th row of **index** table.

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 from the left is matched, so we compare the first character from the right

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 from the right is mismatched, here we stop the comparison.

The last **index** stored in the 0th row of the **index** table is 23, we need to start the search for P from 23rd character in S, there is only one character after the 23rd character in the string, but the pattern has 3 characters more from the 23rd location. So it is impossible to occur the pattern starting from 23rd location in S. Finally the search for P in S is completed, P occurred two times in the string S.

E. Example 2:

The DNA sequence data has been taken from the Multiple Skip Multiple Pattern Matching algorithm MSMPMA [8] for testing the IFBMPM 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.

AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAG
TGTTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGC
TGTTCAACGCTCCAAGCTCGTGACCTCGTCACTACGACG
GCGAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGAC
GCGTGGTGAATCCTATGGGTTAGGATCGTGTCTACCCCA
AATTCTTAATAAAAAACCTAGGACCCCCTTCGACCTAGAC
TATCGTATTATGGACAAGCTTTAACTGTCGTACTGTGGAG
GCTTCAAAACGGAGGGACCAAAAAATTTGCTTCTAGCGT
CAATGAAAAGAAGTCGGGTGTATGCCCCAATTCCTTGCT
GCCCGGACGGCCAGGCTTATGTACAATCCACGCGGTAC
TACATCTTGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAA
TCATAGCGGTACTTCATAATGGGACACAACGAATCGCGG
CCGGATATCACATCTGCTCCTGTGATGGAATTGCTGAAT
GCGCAGGTGTGAATACTGCGGCTCCATTCGTTTTGCCGT
GTTGATCGGGAATGCACCTCGGGGACTGTTCGATACGA
CCTGGGATTTGGCTATACTCCATTCCTCGCGAGTTTTCG

ATTGCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCAC CCACTTCGAGAAGTGAATGGCTGGCTCCTGAGCGCGTC CTCCGTACAATGAAGACCGGTCTCGCGCTAAATTTCCCC CAGCTTGTACAATAGTCCAGTTTATTATCAAAGATGCGAC AAATAAATTGATCAGCATAATCGAAGATTGCGGAGCATAA GTTTGGAAAACTGGGAGGTTGCCAGAAAACTCCGCGCC TACTTTCGTCAGGATGATTAAGAGTATCGAGGCCCCGCC GTCAATACCGATGTTCTTCGAGCGAATAAGTACTGCTATT TTGCAGACCCTTTGCCAGGCCTTGTCTAAAGGTATGTTA CTTAATATTGACAATACATGCGTATGGCCTTTTCCGGTTA ACTCCCTG.

The **index** table for *S* is very large to show here. So for different *P*'s the number of occurrences and the number of **comparisons** are shown in the following table.

TABLE IV
EXPERIMENTAL RESULTS OF PROPOSED ALGORITHM

S. No.	Pattern(*P*'s)	No. of Characters	No. of occurrences	No. of comparisons
1	*A*	1	259	518
2	*AG*	2	53	624
3	*CAT*	3	11	567
4	*AACG*	4	5	614
5	*AAGAA*	5	2	616
6	*AAAAAAGG*	8	1	634
7	*TTCTTAATAAAA*	12	1	651
8	*GGCTGTTCAACGCTC*	16	1	598

In molecular biology this type of large sequences are common to compare with some other sequences . To check whether the given pattern presents in the sequence or not we need a efficient algorithm which does the search in less time and with good complexity. The general algorithms like *Brute Force* or other conventional algorithms will take much time to do this. There are so many algorithms are introduced to solve this problem with less **comparisons** and in less time but each have their drawbacks. The proposed **Index** Based Forward Backward String Searching algorithm is one simple solution for such needs.

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

IV. RESULTS AND DISCUSSION

From the proposed algorithm it has been observed the following experiments when compared with some of the other algorithms. Fig.1 shows the number of **comparisons** made for different algorithms to the single pattern of length 1. For a single pattern "*A*" the proposed algorithm takes 518 **comparisons** whereas all the other algorithms take nearly 1024 **comparisons**.

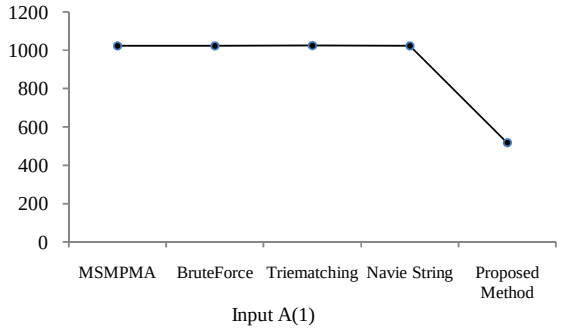

Fig.1 Experimental results of different algorithms

Fig.2 shows the number of **comparisons** made for different algorithms to the pattern of length 2. The pattern "*AG*", in the proposed algorithm takes 624 **comparisons** where as all the other takes more than 1230 **comparisons**. We are reducing the **comparisons** less than half by using the **index** based technique.

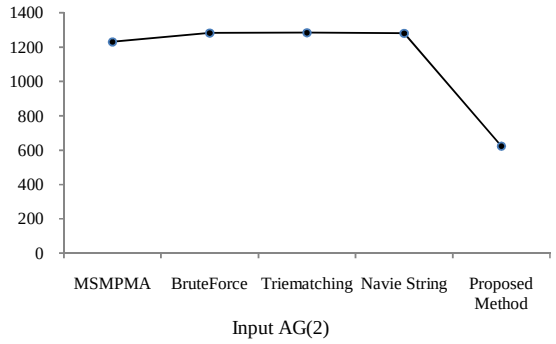

Fig.2 Experimental results of different algorithms

Fig.3 shows the number of **comparisons** made for different algorithms to the pattern of length 3. The pattern "*CAT*", in the proposed algorithm takes 567 **comparisons** where as all the other algorithms like Brute-force, MSMPMA and Trie-matching takes more than 1298 **comparisons**.

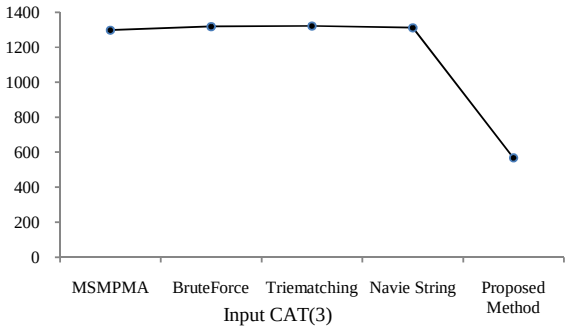

Fig.3 Experimental results of different algorithms

Fig 4. shows the number of **comparisons** made for different algorithms to the pattern of length 4. The pattern "*AACG*", in the proposed algorithm takes 614 **comparisons** where as all the other algorithms takes more than 1359 **comparisons**.

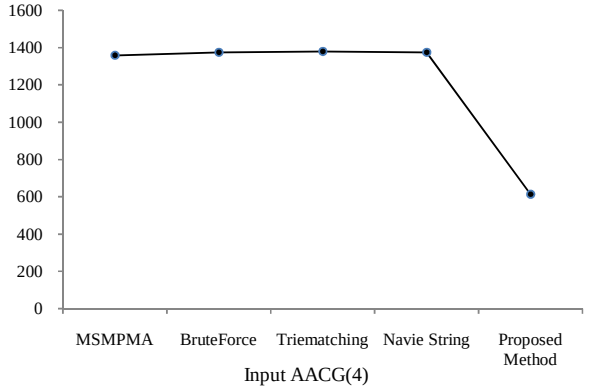

Fig.4 Experimental results of different algorithms

Fig 5. shows the number of **comparisons** made for different algorithms to the pattern of length 5. The pattern "*AAGAA*", in the proposed algorithm takes 616 **comparisons** where as all the other takes more than 1375 **comparisons**. By taking pattern size 5 our algorithm **comparisons** has increased slightly. In all other cases it is less than half where as in this case is more than the half.

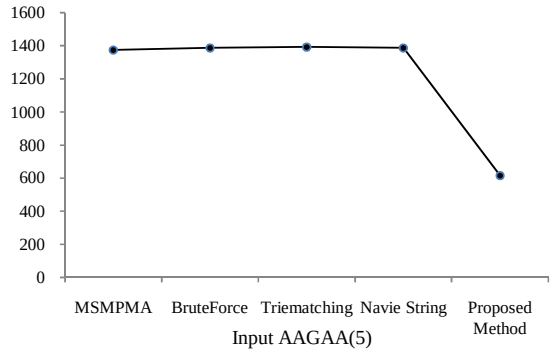

Fig.5 Experimental results of different algorithms

Fig 6. shows the number of **comparisons** made for different algorithms to the single pattern of length 8. The pattern "*AAAAAAGG*", in the proposed algorithm takes 634 **comparisons** where as all the other takes more than 1394 **comparisons**.

The CPC value is less than 1 in the **index** based matching algorithm where as in all other algorithms it is more than 1.

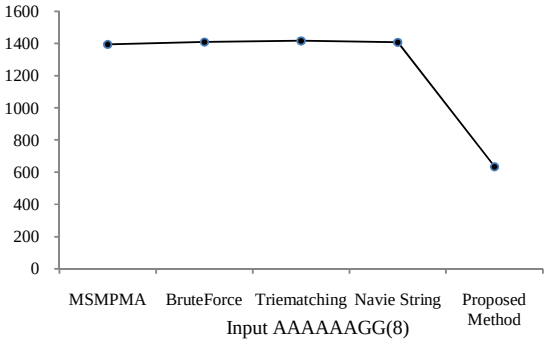

Fig.6 Experimental results of different algorithms

Fig 7. shows the number of **comparisons** made for different algorithms to the single pattern of length 12. The pattern "*TTCTTAATAAAA*", in this the proposed algorithm takes 651 **comparisons** where as all the other takes more than 1390 **comparisons**. In this case the comparison slightly decreased when compared with the earlier cases.

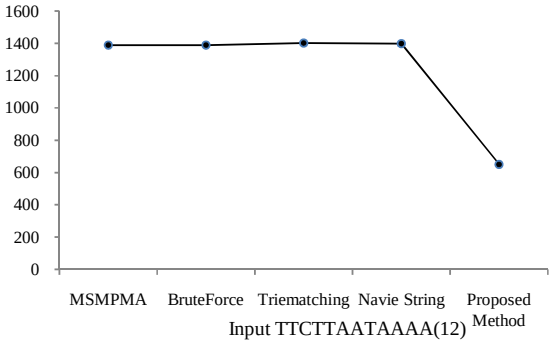

Fig.7 Experimental results of different algorithms

Fig 8. shows the number of **comparisons** made for different algorithms MSMPMA[8], Brute-force, Trie-matching, Naïve string search with the proposed pattern matching algorithm and tested with the pattern of length 16. The pattern "*GGCTGTTCAACGCTCC*", in this the **index** based sequential searching algorithm takes 598 **comparisons** where as all the other takes more than 1349 **comparisons**. Overall performance of the algorithm is very good when analyzed with other algorithms.

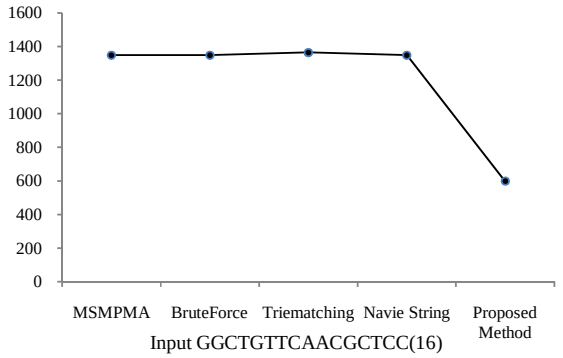

Fig.8. Experimental results of different algorithms

Fig 9. shows the comparison between the different algorithms like MSMPMA, Brute-force, Trie-Matching, Naïve-string matching and the **index** based sequential searching algorithms. It is clear that our **index** based algorithm outperforms when compared with all other algorithms.

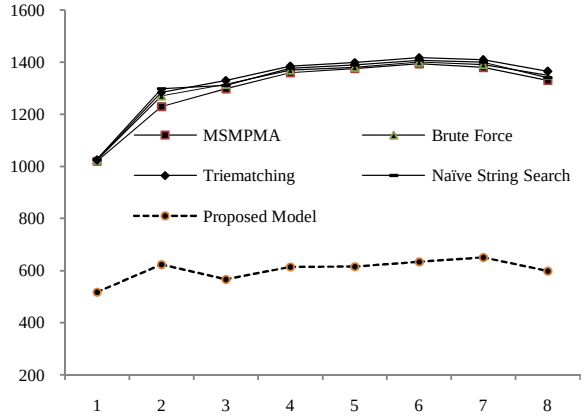

Fig 9. Comparison of different algorithms

Our **index** based algorithm gives very good performance in number of **comparisons** of the patterns when compared with the other popular algorithms. The dotted lines gives the **index** based where as the MSMPMA, Brute-Force, Trie-matching and Naïve string based is the other plotted lines in the graph.

Table.V shows experimental results of the **index** based algorithm, the number of **comparisons** decreases and comparison per character ratio is less than 1in case of **index** based method.

TABLE V
EXPERIMENTAL RESULTS OF IFBMPM FOR DIFFERENT PATTERN SIZES

S. No	Pattern(*P*'s)	Pattern Lenth *P*	No. of occurrence	IFBMPM Model Comparison	CPC Ratio
1	A	1	259	518	0.505
2	AG	2	53	624	0.609
3	CAT	3	11	567	0.553
4	AACG	4	5	614	0.599
5	AAGAA	5	2	616	0.601
6	AAAAAA	6	3	627	0.612
7	AGAACGC	7	2	600	0.585
8	AAAAAAGG	8	1	634	0.619
9	GCTCATTAG	9	1	582	0.568
10	CCTTTTCCGG	10	1	562	0.548
11	TTTTGCCGTGT	11	1	650	0.634
12	TTCTTAATAAAA	12	1	651	0.635
13	GGGACCAAAAAAT	13	1	579	0.565
14	TTTTGCCGTGTTGA	14	1	638	0.623
15	CCTCCAAAAAAGGCT	15	1	578	0.564
16	GGCTGTTCAACGCTCC	16	1	598	0.583
17	TTTTCGATTGCTCATTA	17	1	643	0.627
18	GGGATTTGGCTATACTCC	18	1	598	0.583
19	GGCCTTGTCTAAAGGTATG	19	1	579	0.565
20	CCTGAGCGCGTCCTCCGTAC	20	1	570	0.556

Table.VI shows experimental results of different algorithms used to compare and analyze the results related to the algorithms like MSMPMA[8], Brute-Force, Trie-Match, naïve string matching with the proposed algorithm. A comparison has been done on the basis of number of **comparisons** and comparison per character with the other algorithms. **Index** based algorithm gives the best performance and CPC(comparison per character) ratio comes to half in the current algorithm.

TABLE VI
COMPARISONS OF DIFFERENT ALGORITHMS WITH IFBMPM

Pattern	No.of occurances	IFBMPM Model		MSMPMA		Brute-Force		Tri-match		Naïvestring	
		No.of Comp	CPC	No.of Comp	CPC	No.of Comp	CPC	No.of Comp	CPC	No.of Comp	CPC
A	259	518	0.50	1024	1.00	1024	1.00	1025	1.00	1024	1.00
AG	53	624	0.60	1230	1.20	1282	1.25	1284	1.25	1281	1.25
CAT	11	567	0.55	1298	1.26	1318	1.28	1321	1.29	1310	1.27
AACG	5	614	0.59	1359	1.32	1376	1.34	1380	1.34	1376	1.34
AAGAA	2	616	0.60	1375	1.34	1388	1.35	1393	1.36	1387	1.35
AAAAAAGG	1	634	0.61	1394	1.36	1409	1.37	1417	1.38	1407	1.37
TTCTTAATAAAA	1	651	0.63	1390	1.35	1390	1.35	1402	1.36	1399	1.36
GGCTGTTCAACGCTCC	1	598	0.58	1349	1.31	1349	1.31	1365	1.33	1349	1.31

The following are observed from the experimental results.

1) Reduction in number of **comparisons**.
2) The ratio of **comparisons** per character has gradually reduced and is less than 1.

3) Suitable for unlimited size of the input file.
4) Once the **indexes** are created for input sequence we need not create them again.
5) For each pattern we start our algorithm from the matching character of the pattern which decreases the unnecessary **comparisons** of other characters.
6) It gives good performance for DNA related sequence applications.

V. CONCLUSION

A new algorithm for searching sequence pattern is proposed. This paper gives the time efficient method for solving pattern matching problem. It is very simple approach for finding the patterns. The proposed algorithm gives very good performance with the other algorithms. We have compared **comparisons** per character ratio, number of **comparisons**. We have implemented with **DNA Sequence** further it can be extended to protein sequence.

REFERENCES

[1] Boyer R. S., and J. S. Moore, ''A fast string searching algorithm, '*Communications of the ACM* 20 (October 1977), pp. 762 772.
[2] Knuth D., Morris.J Pratt.V Fast pattern matching in strings, *SIAM journal on computing*.
[3] Kurtz. S, Approximate string searching under weighted edit distance. *In proceedings of the 3rd south American workshop on string processing. (WSP 96)*. Carleton Univ Press, 1996 156-170.
[4] 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.
[5] Ukkonen,E., Finding approximate patterns in strings J.Algor. 6, 1985, 132-137
[6] Wu S., and U. Manber, ''Agrep — A Fast Approximate Pattern-Matching Tool,'' *Usenix Winter* 1992 *Technical Conference, San Francisco* (January 1992), pp. 153 162.
[7] WU.S.,Manber U., and Myers,E .1996, A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica* 15,1,50-67, Computer Science Dept, University of Arizona,1992.
[8] [MSMPMA] Ziad A.A Alqadi, Musbah Aqel & Ibrahiem M.M.EI Emary, Multiple Skip Multiple Pattern Matching algorithms. *IAENG International Journal of Computer Science* 34:2.

Raju Bhukya has received his B.Tech in Computer Science and Engineering from Nagarjuna University in the year 2003 and M.Tech degree in Computer Science and Engineering from Andhra University in the year 2005. He is currently working as an Assistant Professor in the Department of Computer Science and Engineering in National Institute of Technology, Warangal, Andhra Pradesh, India. He is currently working in the areas of Bio-Informatics and Data Mining.

Somayajulu DVLN has received his M. Sc and M. Tech degrees from Indian Institute of Technology, Kharagpur in 1984 and in 1987 respectively, and his Ph. D degree in Computer Science & Engineering from Indian Institute of technology, Delhi in 2002. He is currently working as Professor and Head of Computer Science & Engineering at National Institute of Technology, Warangal. His current research interests are bio-informatics, data warehousing, database security and Data Mining.