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 $\Sigma = \{A, C, G, T\}$. Let $T$ be a large text consists of characters in $\Sigma$ denoted as $\Sigma^*$. 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.

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 \textit{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 \( \sum = \{A, C, G, T\} \)
- \( \phi \) 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 \textit{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 \( O(mm) \). 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 \( O(mm) \) 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>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Author</th>
<th>Comparison Order</th>
<th>Preprocessing</th>
<th>Searching Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boyer Moore</td>
<td>R.S. Boyer and J.S. Moore</td>
<td>From right to left</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Horspool</td>
<td>Nigel Horspool</td>
<td>Is not relevant</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Brute force</td>
<td>-</td>
<td>Is not relevant</td>
<td>No</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Kunth Morris Pratt</td>
<td>Michael O Rabin and Richard M Karp</td>
<td>From left to right</td>
<td>Yes</td>
<td>O(n+m) independent from the alphabet size</td>
</tr>
<tr>
<td>Quick Search</td>
<td>Sunday</td>
<td>Is not relevant</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Karp</td>
<td>Michel O Rabin and Richard M Karp</td>
<td>From left to right</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Zhu Takaoka</td>
<td>R. F. Zhu and T Takaoka</td>
<td>From right to left</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
<tr>
<td>Index Based</td>
<td>IFBMPM Model</td>
<td>From left to right</td>
<td>Yes</td>
<td>O(mn)</td>
</tr>
</tbody>
</table>

In the proposed work we use the indexes for the DNA Sequence belongs to $\Sigma^*$. 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 $\Sigma$. 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 $\Sigma$ = {A, C, G, T}. Let the string be $S$ of having $n$ characters and the pattern $P$ of having $m$ characters. Then $S, P \in \Sigma^*, |S| = n$ 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 $S, P \in \Sigma^*$.

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

Algorithm:

Step 1: Integer arrays indexTab[4][n], charIndex[4]

$\text{indexText} = 1, \text{n_occ} = 0, \text{n_cmp} = 1$

Step 2: $\text{FOR} i = 0; i < \text{n}; i++$

$\text{indexTab}[\text{charIndex}[\text{charIndex}[\text{charIndex}[\text{charIndex}\mid S[i]-64\%5]\mid S[i]-64\%5]\mid S[i]-64\%5]\mid S[i]-64\%5]++; i$

End FOR

Step 3: $\text{FOR} i = 0; i < \text{charIndex}[\text{P[0]-64\%5}]; i++$

$\text{found} = 1$

$\text{IF} i + m - 1 > n - 1$

$\text{found} = 0$

$\text{SKIP the test, GOTO step 4.}$

End IF

$\text{FOR} r = 0; r < m/2; r++$

$\text{n_cmp}++$

$\text{IF} P[r] = S[r + i]$

$\text{n_cmp}++$

$\text{DO Nothing}$

$\text{ELSE}$

$\text{found} = 0$

End IF

$\text{ELSE}$

$\text{found} = 0$

End IF

End FOR

Step 4: IF $\text{found} = 1$

$n_{occ}++$

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 $\Sigma$ = {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 $\Sigma$ we compute its array subscript value in indexTab by using the following technique.
Let \( B \) 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 \( P_s = S_x \), if so we will check \( P_{m-r+1} = S_{x+m-r+1} \) 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 for any \( |P| \geq 0 \) then the number of occurrences of \( P \) in \( S \) is 0.

**Case ii:** 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 iii:** 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.

**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 \( S = \text{ACTTAGCTCAACGATGTAGC} \) of 25 characters and \( P = \text{TTAG} \).

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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Array Subscript Value</th>
<th>Corresponding Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>G</td>
</tr>
<tr>
<td>T</td>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

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 \( P_s = S_x \), if so we will check \( P_{m-r+1} = S_{x+m-r+1} \) 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 \( i \).
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.

\[ S = A C T T A G G C T C A A C G A T G T T A G C A T C \]

\[ P = T T A G \]

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 index18 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 \( S \in \Sigma^* \) of size \( n=1024 \) and pattern \( P \in \Sigma^* \). Let \( S \) be the following DNA sequence.

\[ AGAACGCAGACAAAGTTTCTCATTTGCTCTTCGCAATAG \]
\[ TGGTACAAACTCGGTGTCTTGGCTCCTCCCAAAAAAGGG \]
\[ TGTTACACGGTTACACAGCTGGCATCCTGACTAAGGCG \]
\[ GCGAGGTAAAGACCGCCAGAGGTAAAGGAACCTAATGAC \]
\[ GCTGGGATTCTATGCGGCTATGACTGTTGACCTCCCA \]
\[ AATTCCTTAATAAAAACCTTAGACCCCCCCTTGGACCTAAGAC \]
\[ TATCGTATATTGGGAAACAACCTTGAATTTTGTCTTCTAGCTG \]
\[ CAAATGAAAAGAATGCGGTTGTATGCCCCAATTCTTCGCTCT \]
\[ GCTGCGGTACCGGCGGCTGATTGAATCCTATGGGTTAGGGT \]
\[ GCCGCGCGCTGCGCGCTGATTGAATCCTATGGGTTAGGGT \]
\[ TGATCGGCTCTTCTGTTGACTTCTGTTGACTTCTGTTGACT \]
\[ CTAAGCTCTTTTGATTGACTTCTGTTGACTTCTGTTGACT \]
\[ TACATCGGCTCTTCTGTTGACTTCTGTTGACTTCTGTTGACT \]
\[ CCGGCGGCGGCGGCGGCGGCGGCGGCGGCGGCGGCGGCGGCGG \]
\[ GCGGATATCCATCGTCTCTGGTAGTGAATTGCTGAAT \]
\[ GCGGAGGTTGTAAACTCGCGCTACCTTGGGTGACCTACTACGG \]
\[ GTTTCTTTTCTTTTCTTTTCTTTTCTTTTCTTTTCTTTTCTTT \]
ATTGCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCAC
CCACTTCAGAAGTATGCGCTGCTCCTGAGACGCCTG
CTCCGTACAATGAAGACCGGTCTCGCGCTAAATTTCCCC
CAGCTTGTACAATAGTCCAGTTTATTATCAAAGATGCGAC
AAATAATTGATCAGATAATCAGAAGATTGCGAGCATTTA
GGTGGAAGAAGCAGTGTTGCCCAGAAAATTCCCGCC
TTCTTCGCAGATTGAATTAGATCGAGGCCCCGCC
GTCAATACCGATGTTCCTCGACGGATAAAGTACTGCTATT
TTGCAGACCCTTTGCCCAGGCCCTTGGCCATAGATGTTA
CTTAATTGACATAATCAGGTAATGCTGCTTTGCCCTGTA
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

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Pattern(P’s)</th>
<th>No. of Characters</th>
<th>No. of occurrences</th>
<th>No. of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>1</td>
<td>259</td>
<td>518</td>
</tr>
<tr>
<td>2</td>
<td>AG</td>
<td>2</td>
<td>53</td>
<td>624</td>
</tr>
<tr>
<td>3</td>
<td>CAT</td>
<td>3</td>
<td>11</td>
<td>567</td>
</tr>
<tr>
<td>4</td>
<td>AACG</td>
<td>4</td>
<td>5</td>
<td>614</td>
</tr>
<tr>
<td>5</td>
<td>AGAA</td>
<td>5</td>
<td>2</td>
<td>616</td>
</tr>
<tr>
<td>6</td>
<td>AAAAAGG</td>
<td>8</td>
<td>1</td>
<td>634</td>
</tr>
<tr>
<td>7</td>
<td>TTCTAAFAAA</td>
<td>12</td>
<td>1</td>
<td>651</td>
</tr>
<tr>
<td>8</td>
<td>GGTGTCTCAACGGTC</td>
<td>16</td>
<td>1</td>
<td>598</td>
</tr>
</tbody>
</table>

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 an 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 whereas all the other take 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 whereas 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.

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 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 6. 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. 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.

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

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.

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.
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


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 DVN 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.