Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair

Raju Bhukya
Assistant Professor
Department of Computer Science and Engineering
National Institute of Technology, A.P, India

DVLN Somayajulu
Professor
Department of Computer Science and Engineering
National Institute of Technology, A.P, India

ABSTRACT
Exact string matching algorithms are essential components in DNA applications of the computational biology. Pattern matching is an important task of the pattern discovery process in today's world for finding the structural and functional behavior in proteins and genes. Although pattern matching is commonly used in computer science and information processing, it can be found in everyday tasks. Molecular biologists often search for the important information from the databases in different directions of different uses. With the increasing need for instant information, pattern matching will continue to grow and change as needed from time to time. In this research we propose a new pattern matching technique called an Exact multiple pattern matching algorithms using DNA sequence and pattern pair. The current approach is used to avoid unnecessary comparisons in the DNA sequence. Due to this, the number of comparisons gradually decreases and comparison per character ratio of the proposed algorithm reduces accordingly when compared to the some of the existing popular methods. Proposed algorithm is implemented and compared with existing algorithms. Comparison results demonstrate that index based algorithm is efficient than the number of the existing techniques.

General Terms
Single pattern, Multiple pattern, Exact pattern, Inexact pattern

Key Words
Comparisons, DNA Sequence, Index.

1. INTRODUCTION
DNA pattern matching is a fundamental and upcoming area in computational molecular biology. The problem in pattern discovery is to determine how often a candidate pattern occurs, as well as possibly some information on its frequency distribution across the sequence/text. In general, a pattern will be a description of a set of strings, each string being a sequence of symbols. Hence, given a pattern, it is usual to ask for its frequency, as well as to examine its occurrences in a given sequence/text. Many algorithms have been developed each designed for a specific type of search. Although they all serve the same function but they vary in the way they process the search, and second in the methods they use to efficiently achieve the optimal processing time.

Bioinformatics is a multidisciplinary science that uses methods, principles from mathematics, computer science for analyzing the computer data. DNA is the basic blue print of life and it can be viewed as a long sequence over the four alphabets *A, C, G* and *T*. DNA contains genetic instructions of an organism. It is mainly composed of nucleotides of four types. Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). The amount of DNA extracted from the organism is increasing exponentially. So pattern matching techniques plays a vital role in various applications in computational biology for data analysis related to protein and gene. It focuses on finding the particular pattern in a given DNA sequence. The biologists often queries new discoveries against a collection of sequence databases such as GENBANK, EMBL and DDBJ to find the similarity sequences.

As the size of the data grows it becomes more difficult for users to retrieve necessary information from the sequences. Hence more efficient and robust methods are needed for fast pattern matching techniques. It is one of the most important areas which have been studied in bioinformatics. The string matching can be described as: given a specific strings *P* generally called pattern searching in a large sequence/text *T* to locate *P* in *T*. if *P* is in *T*, the matching is found and indicates the position of *P* in *T*, else pattern does not occurs in the given text. Pattern matching techniques has two categories and is generally divides into multiple pattern matching and single pattern matching algorithms.

- Single pattern matching
- Multiple pattern matching techniques

In a standard problem, we are required to find all occurrences of the pattern in the given input text, known as single pattern matching. Suppose, if more than one pattern are matched against the given input text simultaneously, then it is known as, multiple pattern matching. Whereas single pattern matching algorithm is widely used in network security environments. In network security the pattern is a string indicating a network intrusion, attack, virus, and snort, spam or dirty network information, etc. Multiple pattern matching can search multiple patterns in a text at the same time. It has a high performance and good practicability, and is more useful than the single pattern matching algorithms.

Let $P = \{p_1, p_2, p_3,...,p_m\}$ be a set of patterns of *m* characters and $T=\{t=t1,t2,t3...tn\}$ in a text of *n* characters which are strings of nucleotide sequence characters from a fixed alphabet set called $\sum = \{A, C, G, T\}$. Let *T* be a large text consisting of characters in $\sum$. In other words *T* is an element of $\sum^*$. The problem is to find all the occurrences of pattern *P* in text *T*. It is an important application widely used in data filtering to find selected patterns, in security applications, and is also used for DNA searching. Many existing pattern matching algorithms are reviewed and classified in two categories.

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

Exact pattern matching algorithm will find that whether the probability will lead to either successful or unsuccessful search. The problem can be stated as: Given a pattern p of length m and a string/Text T of length n ($m \leq n$). Find all the occurrences of p in T. The matching needs to be exact, which means that the exact word or pattern is found. Some exact string matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm [3], KMP Algorithm [7].

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

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

In many cases most of the algorithm operates in two stages. 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. Some pattern matching algorithm concentrates on pattern itself. Other algorithm compare the corresponding characters of the patterns and text from the left to right and some other perform the character from the right to left. The performance of the algorithm can be measured based upon the specific order they are compared. Pattern matching algorithms has two different phases.

- 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 pre-processing. Pattern analysis plays a major part for various analysis like discrimination of the cancer from gene expression, mutation evolution, data analysis, feature extraction, searching, disease analysis, structural and functional analysis, e-books, text processing, linguistic translation, data compression, search engine, speech reorganization, information retrieval, genomic data, protein-protein interaction in cellular activities, computer virus detection, network intrusion detection, parsers, spam filters, digital libraries, screen scrapers, word processors, natural language processing and computational biology.

The rest of the paper is organized as follows. We briefly present the Background and related work in section 2. Section 3 deals with Proposed model *i.e.*, EPMSPP algorithm for DNA sequence. Results and discussion are presented in Section 4 and we make some concluding remarks in Section 5.

2. BACKGROUND AND RELATED WORK

This section reviews some work related to DNA sequences. An alphabet set $\sum = \{A, C, G, T\}$ is the set of characters for DNA sequence which are used in this algorithm. The following notations are used in this paper.

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

ϕ Denotes the empty string.

$|P|$ Denotes the length of the string P.

$S[n]$ Denotes that a text which is a string of length n.

$P[m]$ Denotes a pattern of length m.

CPC-Character per comparison ratio.

String matching mainly deals with problem of finding all occurrences of a string in a given text. In most of the DNA applications it is necessary for the user and the developer to be able to locate the occurrences of specific pattern in a sequence. In Brute-force algorithm the first character of the pattern P is compared with the first character of the string T. If it matches, then pattern P and string T are matched character by character until a mismatch is found or the end of the pattern P is detected. If mismatch is found, the pattern P is shifted one character to the right and the process continues. The complexity of this algorithm is $O(mn)$. The Bayer-Moore algorithm [3] applies larger shift-increment for each mis-match detection. The main difference the Naïve algorithm had is the matching of pattern P in string T is done from right to left *i.e.*, after aligning P and string T the last character of P will matched to the first of T . If a mismatch is detected, say C in T is not in P then P is shifted right so that C is aligned with the right most occurrence of C in P. The worst case complexity of this algorithm is $O(m+n)$ and the average case complexity is $O(n/m)$. In IBKPMPM [11] choose the value of k and divide both the string and pattern into number of substring of length k, each substring is called as a partition. We compare all the first characters of all the partitions, if all the characters are matching while we are searching then we go for the second character match and the process continues till the mismatch occurs or total pattern is matched with the sequence. The worst case complexity is $O(m+n)$. The KMP algorithm [7] is based on the finite state machine automation. The pattern P is pre-processed to create a finite state machine M that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is $O(m+n)$.

In approximate pattern matching method the oldest and most commonly used approach is dynamic programming. In 1996 Kurtz [8] proposed another way to reduce the space requirements of almost $O(mn)$. The idea was to build only the states and transitions which are actually reached in the processing of the text. The automaton starts at just one state and transitions are built as they are needed. The transitions those were not necessary will not be build. In the MSMPMA [14] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. By using this index as the starting point of matching, it compares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges from 1 to m-1). Harspool [6] 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[2] 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[4] designed an algorithm quick search which scans the character of the window in any order and computes its shift with the occurrence shift of the character *T* immediately after the right end of the window. Ukkonen[13]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[3] 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.

By using dynamic programming approach especially in DNA sequencing Needleman-Wunsch[9] algorithm and Smith-waterman algorithms [12] 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 Raita algorithm [10] utilizes the same approach as Horspool algorithm[6] to obtaining the shift value after an attempt. Instead of comparing each character in the pattern with the sliding window from right to left, the order of comparison in Raita algorithm [10] is carried out by first comparing the rightmost and leftmost characters of the pattern with the sliding window. If they both match, the remaining characters are compared from the right to the left. Intuitively, the initial resemblance can be established by comparing the last and the first characters of the pattern and the sliding window. Therefore, it is anticipated to further decrease the unnecessary comparisons. The Aho-Corasick[1] algorithm consists of constructing a finite state pattern matching machine from the keyword and then using the machine to process the text in a single pass. It can find an occurrence of several patterns in the order of *O(n)* time, where *n* is the length of the text, with pre-processing of the patterns in linear time.

Two dimensional pattern matching methods are commonly used in computer graphics. Takaoka and Zhu proposed using a combination of the KMP [7] and RK methods in an algorithm developed for two dimensional cases. Three dimensional pattern matching is useful in solving protein structures, retinal scans, finger printing, music, OCR and continuous speech. Multi-dimensional matching algorithms are a natural progression of string matching algorithms toward multi-dimensional matching patterns including tree structure, graphs, pictures, and proteins structures. The Devaki-Paul algorithm[5] for multiple pattern matching requires a pre-processing of the given input text to prepare a table of the occurrences of the 256 member ASCII character set. This table is used to find the probability of having a match of the pattern in the given input text, which reduces the number of comparisons, improving the performance of the pattern matching algorithm. The probability of having a match of the pattern in the given text is mathematically proved.

3. AN EXACT PATTERN MATCHING ALGORITHM USING PAIR INDEXING FOR SEQUENCE AND PATTERN

Initially in the proposed work indexes are used for the DNA sequence. It first scans the 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 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 which reduces the number of comparison. There are 4 characters in our alphabet 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* in separate array where $p \in \sum_p$. After creating the index the algorithm searches for the pattern in the string using the pair p_i in *P* with least frequency in sequence S. 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.

By using the index we find the possible location of pattern in the sequence. Now to match the pattern in many algorithms we used to compare sequentially. A better approach is to match in such a way which maximizes the chances of a mismatch. For doing this first form an index table for the pattern in the same way as was done for sequence. Now we match the pattern in descending order of frequencies of character pairs in pattern *P*. The one with maximum count m_i is matched first followed by other pairs in decreasing order of frequencies. 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.

3.1 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], ptab[16][n], sidx[16], pidx[16]
Integer arrays ssort[16]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Integer arrays psort[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 **ssort** in ascending order according to values in **sidx**

Step 5: FOR i := 0; i < n; i++
ptab[(*pp & 1536)>>9 | (*pp & 6)<<1][pidx[(*pp &1536)>>9|(*pp & 6) << 1]++] = i;

```
        increment pp by two bytes
        END FOR
Step 6: Sort array psort in descending order according to values in
pidx
Step 7: 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 8: Store in dif the index of first occurrence of y in array p.
Step 9: FOR i := 0; i <=sidx[y]; i++
                found := 1;
                k := stab[y][i] - dif;
                IF k < 0
                continue;
                END IF
                sp := &S[k]
                FOR j = 0; j< m/2; j++
                        ncmp+=2;
                        a:=ptab[psort[b]][c]
                        sp:=&S[k+a*2]
        IF (*sp & 1536)>>9 | (*sp & 6)<<1) != psort[b]
                        found := 0;
                        break;
                        END IF
                        C++
                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. After this it forms one more table *ptab[16][n]* to store indexes of pairs in pattern *P*. This information is used to compare pairs in decreasing order of their frequency in pattern *P*. 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.1. Array subscript values for pair of DNA Characters

S.No	(*sp)	Binaryform (Little Endian form)	(*sp) & 536)>>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 represent different pairs of characters as shown in the table 1. 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.

3.2 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.

3.3 This section describes different examples using our proposed approach (EPMSPP) for the DNA sequences.

Let us discuss an example by taking a DNA sequence *S*=GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGGT of 40 characters and *P* = ACGGAC of 6 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 patter *P* which occurs least number of times in the string *S* for searching. For above example the count value for pairs *AC* and *GG* are 6 and 3 respectively shown in table 2. Count shows the number of times a pair has occurred. Here *GG* occurs least number of times, so p_i = '*GG'* which is used for initial alignment.

Table.2. Index values and count for pair of DNA sequences

Pair	INDEXES						Count/Occ
AA	20	21	22	23	28	34	6
AC	9	13	15	29	32	35	**6**
AT	24	38					2
AG							0
CA	14	19	33				3
CC							0
CT	4	30					2
CG	1	6	10	16	36		5
TA	31						1
TC	3	5	18				3
TT							0
TG	25						1
GA	8	12	27				3
GC	0						1
GT	2	17					2
GG	7	11	26				3

In this technique it first aligns the pair '*GG'* in pattern *P* with '*GG'* in the DNA sequence *S*. After the alignment is completed it matches the character. To decide which pair to match first we create table *ptab* for storing pattern indexes. For making the pattern index table we take a character only once in a pair. Each character is part of only one pair. This is because we require pattern table for comparisons and we need not compare a character twice. For above example the count value for pairs *AC* and *GG* are 2 and 1 respectively as shown in table 3. Count shows the number of times a pair has occurred. Here *AC* occurs most number of times, so m_i = '*AC'* which is compared first.

Table.3. Index values and count for pair pattern

Pair	INDEXES		Count/Occ
AC	0	4	**2**
GG	2		1

It stores the position of *GG* in the pattern *i.e.,* 2 in variable dif. Go to the first occurrence of *GG* using the above table *i.e.,*7 and align the string with pattern by subtracting 2 from the table value to find the starting pair *i.e.,* 7-2=5. Then start matching pairs in decreasing order of frequency in pattern *i.e.,* first all *AC* and then *GG*.

S= GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGGT
P=ACGGAC

The first pair of character doesn't matches so move to next occurrence of *GG* using stab *i.e.,* 11 and align the sequence with pattern by subtracting 2 *i.e.,* 11 – 2 = 9.

S= GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGGT
P=ACGGAC

The first pair of character matched so we match next *AC*.

S= GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGGT
P=ACGGAC

The next pair of character matches. Now all *AC*'s have been compared so now move to next frequent pair *i.e., GG*.

S= GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGGT
P=ACGGAC

All the pairs in the pattern are matched so pattern is found a position 9.Now move to next occurrence of *GG* using stab *i.e.,* 26 and align the sequence with pattern by subtracting 2 *i.e.,* 26 – 2 = 24.

S= GCGTCTCGGACGGACACGTCAAAAATGGAACTACAACGAT
P=ACGGAC

The pair doesn't matches. Now all occurrence of *GG* have been checked so search completes. So only 1 occurrence of pattern was found in the string *S*.

4. EXPERIMENTAL RESULTS

The below shown sequence dataset has been taken from the testing of EPMSPP algorithm. The DNA biological sequence $S \in \sum^*$of size n=1024 and pattern $P \in \sum^*$. Let *S* be the following DNA sequence. The index table (*indexTab[4][1024]*) for sequence *S* is very large in number of DNA sequence characters size1024. For different patterns sizes which has been chosen randomly from the DNA sequence the number of occurrences and the number of comparisons is shown in the Table.4. As we increase the size of the pattern the number of comparisons decreases in some of the cases of the proposed algorithm.

AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAG
TGTTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGC
TGTTCAACGCTCCAAGCTCGTGACCTCGTCACTACGACG
GCGAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGAC
GCGTGGTGAATCCTATGGGTTAGGATCGTGTCTACCCCA
AATTCTTAATAAAAAACCTAGGACCCCCTTCGACCTAGA
CTATCGTATTATGGACAAGCTTTAACTGTCGTACTGTGGA
GGCTTCAAAACGGAGGGACCAAAAAATTTGCTTCTAGCG
TCAATGAAAAGAAGTCGGGTGTATGCCCCAATTCCTTGC
TGCCCGGACGGCCAGTTCATAATGGGACACAACGAATCG
CGGCCGGATATCACATCTGCTCCTGTGATGGAATTGCTG
AATGCGCAGGTGTGCTTATGTACAATCCACGCGGTACTA
CATCTTGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAATC
ATAGCGGTACGAATACTGCGGCTCCATTCGTTTTGCCGTG
TTGATCGGGAATGCACCTCGGGGACTGTTCGATACGACC
TGGGATTTGGCTATACTCCATTCCTCGCGAGTTTTCGATT
GCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCACCCA
CTTCGAGAAGTGAATGGCTGGCTCCTGAGCGCGTCCTCC
GTACAATGAAGACCGGTCTCGCGCTAAATTTCCCCCAGC
TTGTACAATAGTCCAGTTTATTATCAAAGATGCGACAAA
TAAATTGATCAGCATAATCGAAGATTGCGGAGCATAAGT
TTGGAAAACTGGGAGGTTGCCAGAAAACTCCGCGCCTAC
TTTCGTCAGGATGATTAAGAGTATCGAGGCCCCGCCGTC
AATACCGATGTTCTTCGAGCGAATAAGTACTGCTATTTTG
CAGACCCTTTGCCAGGCCTTGTCTAAAGGTATGTTACTTA
ATATTGACAATACATGCGTATGGCCTTTTCCGGTTAACTC
CCTG"

For different patterns *P*'s the number of occurrences and the number of comparisons of the proposed algorithms EPMSPP is shown in the Table.4. Patterns are selected randomly for the

experimental analysis from the DNA data. To check whether the given pattern is present in the sequence or not we need an efficient algorithm with less comparison time and low complexity. By the current technique different patterns are analyzed and the graph is plotted by using these results. Here we have taken five fields in the table .4. The pattern , number of characters in the pattern, number of occurrences of a pattern, comparison by using the proposed method and the number of comparisons and comparisons per character. The number of comparisons per character (CPC) which is equal to: (Number of comparisons/file size) can be used as a measurement factor, this factor affects the complexity time, and when it is decreased the complicity also decreases.

Table.4 .Experimental results of EPMSPP algorithm

Pattern(P's)	No of char	No. of occur	Comparison EPMSPP	CPC
AG	2	53	106	0.10
CAT	3	11	100	0.09
AACG	4	5	140	0.13
AAGAA	5	2	134	0.13
AAAAAA	6	3	362	0.35
AGAACGC	7	2	130	0.12
AAAAAAGG	8	1	146	0.14
GCTCATTAG	9	1	142	0.13
CCTTTTCCGG	10	1	134	0.13
TTTTGCCGTGT	11	1	140	0.13
TTCTTAATAAAA	12	1	144	0.14
GGGACCAAAAAAT	13	1	144	0.14
TTTTGCCGTGTTGA	14	1	138	0.13
CCTCCAAAAAAGGCT	15	1	124	0.12
GGCTGTTCAACGCTCC	16	1	146	0.14
TTTTCGATTGCTCATTA	17	1	122	0.11
GGGATTTGGCTATACTCC	18	1	146	0.14
GGCCTTGTCTAAAGGTATG	19	1	120	0.11
CCTGAGCGCGTCCTCCGTAC	20	1	124	0.12

By the current technique different patterns are randomly taken from the DNA file and results are analyzed and the graph is plotted by using these results accordingly. From the below experimental results, improvement can be seen that EPMSPP algorithm gives good performance compared to the some of the popular methods like IBKPMPM, MSMPMA, Brute-force, Tri-match and naïve string search techniques. From the above table different pattern sizes has been taken for the comparison ranging from 2 to 20 of DNA sequences. Due to the pairing concept we have paired the characters so for single character it is not possible for the comparison.

Table .5. Comparisons of different algorithms with EPMSPP

It has been observed from the experimental result analysis the following in terms of relative performance of our algorithm with the existing popular methods. From the results shown in Table.5 performance can be seen with EPMSPP approach and some of 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 and IKPMPM algorithms. In Table. 4. we have the results of proposed model and the CPC ratio of the proposed model, where as in Table.5 shows the comparison between different existing algorithms with the proposed technique in terms of number of comparisons and the CPC ratio. The table.5 shows randomly chosen 8 different patterns from the above shown DNA data set , whereas Table.4.shows the pattern size starting from 2 character to 20 characters chosen from the DNA dataset.

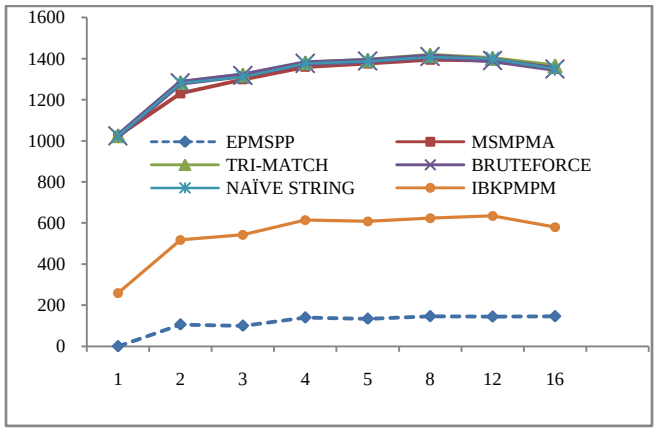

Fig 1. Comparison of different algorithms with EPMSPP

Fig.1. Shows the comparisons of different algorithms with the proposed EPMSPP technique. It is clear that proposed algorithm outperforms when compared with all other algorithms. The current technique gives good performance in reducing the number of comparisons compared with other popular methods. The dotted line shows the EPMSPP model where as MSMPMA, Brute-Force, Trie-matching and Naïve searching and IBKPMPM are shown by solid lines. Towards X-axis we have taken randomly different pattern sizes ranging from 1 to 20 whereas towards Y-axis shows the total number of comparisons.

The below shown are some of the advantages of the proposed algorithm. By our observation the experimental results has been carried out randomly by taking different pattern sizes from the given DNA data set.

- Reduction in number of comparisons.
- The ratio of comparisons per character (CPC) has gradually reduced and is less than 1.
- Suitable for any size of the input file.
- Once the indexes are created for input sequence we need not create them again.
- For each pattern we start our algorithm from the matching character of the pattern which decreases the unnecessary comparisons of other characters.

Pattern	EPMSPP		IBKMPM		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	NV	NV	259	0.2	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	106	0.1	518	0.5	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	100	0	542	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	140	0.1	614	0.6	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	134	0.1	607	0.5	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	146	0.1	623	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	144	0.1	634	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	146	0.2	580	0.5	1349	1.3	1349	1.3	1365	1.3	1349	1.3

- It gives good performance for DNA related sequence applications.

5. CONCLUSION

This study introduced a new Exact multiple pattern matching algorithms using DNA sequence and pattern pair. This paper gives the most efficient method for determining DNA pattern matching sequences. It is a simple approach for finding the multiple patterns from a given sequence file. We have taken the file size of 1024 characters and tested randomly by taking different pattern sizes. The proposed algorithm gives better performance compared with some of the other popular algorithms in case of number of comparisons and CPC ratio. Based on the experimental work carried out with DNA sequence data, EPMSPP approach gives very good performance related to the other methods.

6. REFERENCES

[1] Aho, A. V., and M. J. Corasick, ‘‘Efficient string matching: an aid to bibliographic Search, ’’ Communications of the ACM **18** (June 1975), pp. 333 340.

[2] 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.

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

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

[5] Devaki-Paul, “Novel Devaki-Paul Algorithm for Multiple Pattern Matching” International Journal of Computer Applications (0975 – 8887) Vol 13– No.3, January 2011.

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

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

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

[9] 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.

[10] Raita, T. Tuning the Boyer-Moore-Horspool string-searching algorithm. Software - Practice Experience 1992, 22(10), 879-884.

[11] Raju Bhukya, DVLN Somayajulu,”An Index Based K-Partition Multiple Pattern Matching Algorithm”, Proc. of International Conference on Advances in Computer Science 2010 pp 83-87.

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

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

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