An Index Based Skip Search Multiple Pattern Matching Algorithm

Raju Bhukya,

Assistant Professor, Department of CSE,
National Institute of Technology, Warangal.
Andhra Pradesh. India 506004

Balram Parmer

Department of CSE,
National Institute of Technology, Warangal.
Andhra Pradesh. India 506004

Anand Kulkarni

Department of CSE,
National Institute of Technology, Warangal.
Andhra Pradesh. India 506004.

***Abstract*- DNA Pattern matching, the problem of finding sub sequences within a long DNA sequence has many applications in computational biology. As the sequences can be long, matching can be an expensive operation, especially as approximate matching is allowed. 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 Skip Search Multiple Pattern matching algorithm (ISMPM), for DNA sequences. Our approach avoids 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 other existing popular methods. Our experimental results show that there is considerable amount of performance improvement. The total no of comparisons are drastically reduced when the length of the pattern increases in our algorithm.**

***Keywords*: *DNA Sequence; Index; Skip Search; Pattern Matching*.**

I. INTRODUCTION

We concentrate mainly on the DNA sequence pattern matching using the technique Skip Search based on the Indexing. DNA is the basic blue print of the life and it can be viewed as a long sequence over the four alphabets *A*, *C*, *G* and *T*. The field of bioinformatics has many applications in the modern world which includes text editors, search engine, molecular medicine, industry, agriculture and Comparative biology. 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.

Let $P = \{p_1, p_2, p_3,...,p_m\}$ be a set of patterns which are strings of nucleotide sequence 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 with us is to find all the occurrences of pattern P in text T. It is important application widely used in data filtering to find selected patterns, in security applications, and also used for DNA searching. Many existing pattern matching algorithms are reviewed and classified in two categories.

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

Exact string matching algorithm is for 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 exactly one, which means that the exact word or pattern is found. Some exact matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm[1], Knuth-Morris-Pratt Algorithm[3]. Inexact /Approximate 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 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.

The rest of the paper is organized as follows. We briefly reviewed the background and related work in the section II. In section III deal with proposed model *i.e.*, ISMPM algorithm for DNA sequence. Results and discussion are presented in section IV and conclusion is provided in section .V.

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 are 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-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 applications it is necessary for 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 are available, such as Naïve string search, Brute-force algorithm, Boyer-Moore algorithm [1], Knuth-Morris-Pratt algorithms [3], and Skip Search algorithms [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 Boyer-Moore algorithm [1] applies larger shift-increment for each mismatch detection. A main modification to the Naïve algorithm had 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 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 is $O(m+n)$ and the average case complexity is *O(n/m).*

The Knuth-Morris-Pratt algorithm [3] 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. Ukkonen[7] 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 [1] for exact pattern matching. The complexity of this algorithm in worst and average case is *O(m+n).*In this algorithm every row denotes number of errors and column represents matching a pattern prefix. Deterministic automata approach exhibits *O(n)* worst case time complexity.

In 1996 Kurtz[4] 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 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[8] proposed the algorithm for approximate limited expression matching, and Wu. S. Manber.U proposed the algorithm for fast text searching which allows errors. 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 that 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.

In the MSMPMA [9] 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 compare the file contents from the defined point with the pattern contents, and find the skip value depending upon the match numbers (ranges from 1 to m-1).In the IFBMPM[5] Index based forward backward multiple pattern matching technique the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches. In IBKPMPM [6] algorithm we first choose the value of k (a fixed value), and divide both the string and pattern into number of substring of length k, each substring is called as a partition. If k value is 3 we call it as 3-partition else if it is 4 then it is 4-partition algorithm. 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. If all the characters match then the pattern occurs in the sequence and prints the starting index of the pattern or if any character mismatches then we will stop searching and then go to the next index stored in the index table of the same row which corresponds to the first character of the pattern P

III. An Index based Skip Search MULTIPLE PATTERN MATCHING ALGORITHM (ISMPM)

The proposed work uses the indexes for the DNA sequence of character set $\sum$ to search a pattern in a string. Let S be a string of length n and the pattern P of length m. Let $\sum^*$ be the set of all possible strings with the alphabet set $\sum$. Then $S, P\ \sum^*$, $|S| = n$ and $|P| = m$. Generally $|P| \leq |S|$ *i.e.,* $m \leq n$.

The proposed algorithm consists of two phases *i.e.,* pre-processing and searching phase. In pre-processing phase we will collect the indexes of all four letters in different buckets with total number of times that letter occurred differently in both pattern and the string in different tables respectively. Next we will find out the letter with minimum Multiplicative Value which is calculated by multiplying count of individual letter which occurred in pattern and the count of the respective letter occurred in string for all the four letters {A,C,T,G}.

min_count = Minimum$\sum$*(A,C,T,G) {* (Total no of occurrences in Pattern *(Total no occurrences in String) }

The searching phase start with aligning the selected letter in string with the pattern and start comparing from the first character of the pattern with string till a match or a mismatch occurs. Keeping the letter index (i) fixed, we will compare the pattern with starting index $p=i-j$ till length of the ($i+m-j$) where j is the position of occurrence of that fixed letter and m is the length of the pattern. The Next index of the fixed character is taken from the Index table of the string in such a way that the next index(i) should be greater than the summation of previous position index (i) and of the length of rest of the pattern (j) where j is the first index of that same character in the pattern. This part is known as Skip Part as there is no possibility to find out the patter in this part of the string which is already compared if any such letter occurs in this part then we will exempt that letter and move forward. Here we are reducing the total number of time the outer loop runs and hence total number of comparisons are drastically reduced for the larger length Patterns. Our algorithm's vital part is to reduce the total number of times the outer loop run *i.e* skipping letters in string as much as possible ,and in best case it is $O(n/m)$.The pre-processing phase has time complexity $O(n+m)$.The searching phase has time complexity $O(nm)$.The total number of character comparisons are approximately in the order of n/m. Our algorithm works well with larger size of pattern.

A. ISMPM Algorithm

Input: String S of n characters and a pattern P of m characters, where S, P $\sum^*$.
Output: The number of occurrence and the positions of P in S.

```
Algorithm:
N=length of the string, M=length of the pattern.
S= String ,P=Pattern
Struct bucket {
                  int *a; //for storing the index of letters.
                 int count; //store total no  occurrences of the
                 }
Str_buck[4], Ptr_buck[4] //for storing the indexes of the letters A,C,T and G
Index   //index of the letter chosen with minimum count.
      1)   Pre-Processing Phase
//Store the index of the pattern
 1.Loop for i=1 to i=N
            Str_buck[A,C,T,G].a[i]=i ;
            Str_buck[A…T].cout+=1;
//Store the index of the pattern
  2.Loop for i=1 to i=M
        Ptr_buck[A,C,T,G].a[i]=i ;
            Ptr_buck[A…T].cout+=1;
3. Find  Index for which
((Str_buck[A..T].count)*(Str_buck[A…T].count) is minimum;
Assign ,
mini_scount=Str_buck[Index].count  mini_pcount=Ptr_buck[Index].count

      2)    Searching Phase
       LOOP1 i=1 to i=mini_scount
              q=S[Index].a[i]
             LOOP2  j=1 to j=mini_pcount
IF (q>=Previous Pattern Index + Position of jth Char in Pattern)
            t=Str_buck[index].a[index]-Ptr_buck[index].a[index];
 //t store the starting position from where it starts comparison   in the string.
                        LOOP3 k=1 to k=M
                        Compare (S[t++],P[k]) until a mismatch
                       occurs or Match Occurs.
     PRINT "Starting Index of the string for the Matched Pattern".
                     END LOOP3
             END LOOP3
      END LOOP1
```

The index based Skip Search multiple pattern matching algorithm uses a table (2D vector) called index table. The basic idea here is to store all the indexes of each character in its corresponding row in the 2D vector. A new technique called ASCII value based indexing technique, which is used to reduce the pre- processing time and number of comparisons. The subscript [*(S[i]-64)%5*] always returns a subscript value in the range 0,1,2 and 3 which is needed for subscripting 2D array of size *[4][n]* . The subscript values 0,1,2,3 represent the characters *A, C, G* and *T* respectively. So for each character in the string and pattern of the function (*(S[i]-64)%5) and ((P[i]-64)%5)* directly references to its corresponding row in the *Index Table* respectively. The vector *char Index* stores the counter value of each occurrence of each character with reference to [*(S[i]-64)%5*].

TABLE I .ARRAY SUBSCRIPT VALUES OF DNA CHARACTERS.

S.No	**DNA**	**ASCII Value**	**ASCII Value-64**	**(ASCIIValue- 64)%5**	**Array Subscript**
1	A	65	1	1	1
2	C	67	3	3	3
3	G	71	7	2	2
4	T	84	20	0	0

B. Trivial Cases in Comparisons

If the Index Table of String is either one of the character having count *0* and that same character has non-zero value in the pattern index table then pattern can't be found in the String. Other traditional cases are as follows
Case i: If $S = \varphi$ *i.e.*, $|S| = 0$ and $P = \varphi$ *i.e.*, $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
Case ii: If $S = \varphi$ *i.e.* $|S| = 0$ and for any $|P| \geq 0$ then the number of occurrences of *P* in *S* is 0.
Case iii: If $S \neq \varphi$ *i.e.*, $|S| \neq 0$ and for any $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
Case iv: If $S \neq \varphi$ *i.e.*, $|S| \neq 0$, $P \neq \varphi$ *i.e.*, $|P| \neq 0$ and $|S| \leq |P|$ then the number of occurrences of *P* in *S* is 0

C. This Section describes examples of the proposed approach (ISMPM) for the DNA sequences.

The following sequence and pattern has been taken for example. We have string *S* and pattern *P* as shown below.
S= TCAAGGTCACTGACTATCACTACTGACT

P= ACTGAC

1). Pre-processing Phase

The following index table stores all the indexes in pattern string of each character *A,C,G* and *T* in the corresponding row. The 0th row stores the indexes of occurrences of the charter *A*, 1st row stores the indexes of occurrences of character *C*, 2nd row stores the indexes of the occurrences of *G*, 3rd stores the indexes of occurrences of *T*.

TABLE. II. INDEXES OF A,C,G AND T IN PATTERN

Character	Character Index		Count
A 0	0	4	2
C 1	1	5	2
G 2	3		1
T 3	2		1

The following index table stores all the indexes in input string of each character *A*,*C*,*G* and *T* in the corresponding row. As we scan the input sequence and corresponding characters are placed in the below shown index table in an increasing order as they occur. Till this it is the pre-processing phase and once the characters in the DNA sequence is completed then searching phase begins.

TABLE III INDEX TABLE FOR THE *A, C, G, T* CHARACTER FOR STRING

Character	Character Index									Count
A0	2	3	8	12	15	18	21	25		8
C1	1	7	9	13	17	19	22	26		8
G2	4	5	11	24						4
T3	0	6	10	14	16	20	23	27		8

From the above table we have will select the letter *G* as it has the minimum Multiplicative value calculated from the sequence index table and from the pattern table *i.e.,* 4*1=4 as we will go for searching phase with letter *G* only. If we get the two count value in the sequence or in pattern are same then we will choose the letter having count with less value from the sequence table.

2). Searching Phase

As G is having the minimum multiplicative value calculated above so first align the letter of sequence *G* with pattern *G*.

*S= TCAA**G**GTCACTGACTATCACTACTGACT*

*P= ACT**G**AC*

Then start matching the first character of the of the pattern character with the sequence character. So it mismatches with the first character itself, then we go for the next index of *G*

*S= TCAAG**G**TCACTGACTATCACTACTGACT*

*P=ACT**G**AC*

The next index of *G* from string table is 5 but this is lesser than the (4+3)=8, so it will not consider it and skip this position of *G* since there is no possibility of getting pattern for this position of *G*. This step is known and Skip Part of our algorithm as now it has stopped the algorithm in the beginning to go through the comparison. Now we will go for the next Index of *G i.e.,* 11.

*S= TCAAGGTCACT**G**ACTATCACTACTGACT*

*P=ACT**G**AC*

Here the first character of matched string matches the first character of pattern. So the subsequent characters are also compared sequentially.

S= TCAAGGTCACTGACTATCACTACTGACT

*P=ACT**G**AC*

After comparing the first character it will compare the next character.

*S= TCAAGGTCACT**G**ACTATCACTACTGACT*

*P= ACT**G**AC*

G is already compared so it will go for the next character *A*.

*S= TCAAGGTCACT**G**ACTATCACTACTGACT*

*P= ACT**G**AC*

Next it will compare last character *C*.

*S= TCAAGGTCACT**G**ACTATCACTACTGACT*

*P=ACT**G**AC*

So the pattern matches with the string and index of the starting character of the matched string is returned. We now go for the next index by using the index table containing *G*.

*S= TCAAGGTCACTGACTATCACTACT**G**ACT*

*P=ACT**G**AC*

As first character matches we will compare now the next character *C*.

*S= TCAAGGTCACTGACTATCACTACT**G**ACT*

*P=ACT**G**AC*

Here C also matches with the sequence character we will now compare the next character *T*.

*S= TCAAGGTCACTGACTATCACTACT**G**AC*

*P=ACT**G**AC*

After the *T* character is matched successfully we will now compare the next character *A*.

*S= TCAAGGTCACTGACTATCACTACT**G**ACT*

*P=ACT**G**AC*

A also matches after comparing sequentially we will now compare *C* the last character in the pattern.

*S= TCAAGGTCACTGACTATCACTACT**G**ACT*

*P=ACT**G**AC*

So the pattern matches with the string and index of the starting character of the matched string is returned.

*D. The below shown sequence dataset has been taken from the testing of ISMPM algorithm. The DNA biological sequence $S \in \sum$*of size n=1024 and pattern $P \in \sum$*.*

AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAGTGTTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGC
TGTTCAACGCTCCAAGCTCGTGACCTCGTCACTACGACGGCGAGTAAGAACGCCGAGAAGGTAAGGGAACTAATGAC
GCGTGGTGAATCCTATGGGTTAGGATCGTGTCTACCCCAAATTCTTAATAAAAAACCTAGGACCCCCTTCGACCTAGAC
TATCGTATTATGGACAAGCTTTAACTGTCGTACTGTGGAGGCTTCAAAACGGAGGGACCAAAAAATTTGCTTCTAGCGT
CAATGAAAAGAAGTCGGGTGTATGCCCCAATTCCTTGCTGCCCGGACGGCCAGGCTTATGTACAATCCACGCGGTAC
TACATCTTGTCTCTTATGTAGGGTTCAGTTCTTCGCGCAATCATAGCGGTACTTCATAATGGGACACAACGAATCGCGG
CCGGATATCACATCTGCTCCTGTGATGGAATTGCTGAATGCGCAGGTGTGAATACTGCGGCTCCATTCGTTTTGCCGT
GTTGATCGGGAATGCACCTCGGGGACTGTTCGATACGACCTGGGATTTGGCTATACTCCATTCCTCGCGAGTTTTCGA
TTGCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGCCACCCACTTCGAGAAGTGAATGGCTGGCTCCTGAGCGCGTCCT
CCGTACAATGAAGACCGGTCTCGCGCTAAATTTCCCCCAGCTTGTACAATAGTCCAGTTTATTATCAAAGATGCGACAA
ATAAATTGATCAGCATAATCGAAGATTGCGGAGCATAAGTTTGGAAAACTGGGAGGTTGCCAGAAAACTCCGCGCCTA
CTTTCGTCAGGATGATTAAGAGTATCGAGGCCCCGCCGTCAATACCGATGTTCTTCGAGCGAATAAGTACTGCTATTTT
GCAGACCCTTTGCCAGGCCTTGTCTAAAGGTATGTTATTAATATTGACAATACATGCGTATGGCCTTTTCCGGTTAACT
CCCTG.

The Index Table (*index Table [4][1024]*) for sequence *S* is very large. For different patterns *P*'s the number of occurrences and the number of comparisons is shown in the Table. IV. The proposed ISMPM technique is one simple solution which gives better performance and less complexity.

E. System Specification for Algorithm Implementation Detail

We have implemented the algorithm in the C ++ language and implemented and used the Processor with RAM 2GB and 2GHZ speed with 160GB hard disk. We have taken the sequence of the standard input characters of 1024 DNA data set for testing which are taken as standard in the other programmers. We have obtained the number of comparisons for different input sized texts and calculated the CPC ratio. We have also performed the comparative study with respect to the existing methods of pattern matching which includes the Brute-Force, MSMPMA, Naïve-string, tri-match, IKPMPM and the other methods. The running complexity of the algorithm in terms of time is almost quadratic but includes the scanning of the whole input .The results obtained are purely experimental basis and can be considered for the further.

IV. RESULTS AND DISCUSSION

After implementation of the Indexed Based Skip Search Algorithm for the 1024 character and finding the no of comparisons and CPC ratio it has been concluded that the number of comparisons decreases drastically as the length of the pattern increases. Our algorithm give very good performance in terms of the best case but its preprocessing phase time is more of the length of the string is much more than the pattern length. We have drawn the a comparative analysis with respect to the various existing algorithm among which the Best case algorithm is K–Partition algorithm and Our ISMPM produces almost 50% less no of comparisons then the K-Partition algorithm .The number of comparisons are reduces as the length of the pattern decreases as we have collected the data for various existing algorithm and are shown below in the Table IV.

TABLE IV. EXPERIMENTAL RESULTS OF ISMPM ALGORITHM

S.No.	Pattern	No of Character	No of Occur.	No of Comparison
1	*A*	1	259	259
2	*AG*	2	53	237
3	*CAT*	3	11	542
4	*AACG*	4	5	200
5	*AAGAA*	5	2	232
6	*AAAAAAGG*	8	1	148
7	*TTCTTAATAAAA*	12	1	260
8	*GGCTGTTCAACGCTCC*	16	1	90

Table.V shows experimental results of different algorithms like MSMPMA, Brute-Force, Trie-Match, IKPMPM and naïve-string with the ISMPM algorithm. The performance of ISMPM is observed with two parameters namely number of comparisons and comparisons per character ratio (CPC).The results show that ISMPM provides best performance and gradually reduces over all the other methods. The number of occurrences is same for all the algorithms but the comparison and comparisons per character ratio is different for different algorithms. In ISMPM algorithm the CPC ratio is less than 0.5 where as all the other algorithms is greater than 1.

TABLE.V. COMPARISON OF DIFFERENT ALGORITHMS

Pattern	ISMPM		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	259	0.2	259	0.2	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	237	0.2	518	0.5	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	542	0.5	542	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	200	0.1	614	0.6	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	232	0.2	607	0.5	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	148	0.1	623	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	260	0.2	634	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	90	0.0	580	0.5	1349	1.3	1349	1.3	1365	1.3	1349	1.3

Fig.1 shows comparisons made by different pattern matching algorithms. It is clear that the proposed (ISMPM) algorithm outperforms when compared with all other algorithms. The upper lines are being drawn for the Brute Force, Naïve Algorithm and Tri-Match algorithm. Our ISMPM algorithm represented in the graph at the lowest point where it is compared with the Best algorithm till now that is K-Partition algorithm. Comparison of Different

Algorithm with ISMPM as shown below.

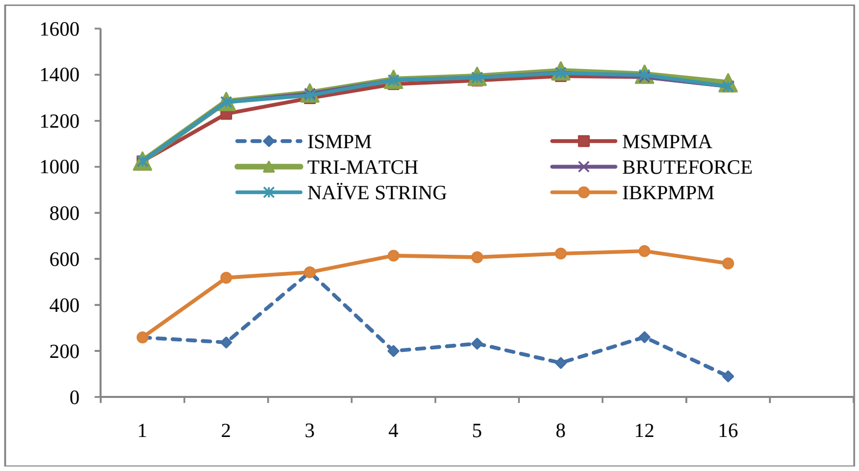

Figure.1. Comparison of different algorithms with ISMPM

It has been observed that the proposed algorithm has been very much improved when compared with the existing techniques. The total number of comparisons reduces as the total number of the character in the pattern increases. It gives the CPC ratio less than 0.5 for all the patters. It gives better performance related to the DNA sequence which has the lesser number of the input string.

V. CONCLUSION

Our ISMPM algorithm reduces the total number of comparison as well as the CPC ratio when compared with the some of the best known popular algorithm till now and its best case arises when the length of the pattern increases. Comparison of proposed algorithm is made with existing algorithms on the basis of the number of comparisons and the attempts made by different pattern sizes of different algorithms to complete the task with our randomly chosen patterns. Our algorithm outperform in case of number of comparisons related to the DNA sequences. The proposed model is shown to be very efficient and fast. Future work may include the work on reducing the pre-processing cost of the string which is costly for our proposed algorithm.

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] Charras C**.**, Lecroqt**.**, Pehoushek. J.D**.**,1998, A very fast string matching algorithm for small alphabets and long patterns, in *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching* , M. Farach-Colton ed., Piscataway, New Jersey, Lecture Notes in Computer Science 1448, pp 55-64, Springer-Verlag, Berlin.

[3] Knuth D., Morris.J Pratt.V Fast pattern matching in strings, SIAM journal on computing.

[4] 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.

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

[6] Raju Bhukya and DVLN Somyajulu ,"An indexed based K-Patition Multiple Pattern Matching Algorithm" Proc. of Int. Conf. on Advances in Computer Science 2010(ACEEE).

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

[8] 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.

[9] 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.

Authors Profile

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.