IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-5, May 2018 E-ISSN: 2347-2693

Comparative Study of String Matching Algorithms for DNA dataset

Pooja Manisha Rahate[1*], **M. B. Chandak**[2]

[1*] Dept. of Computer Science and Tech., Shri Ramdeobaba College of Engineering & Management, Nagpur, Maharashtra, India
[2] Dept. of Computer Science and Tech., Shri Ramdeobaba College of Engineering & Management, Nagpur, Maharashtra, India

**Corresponding Author: poojamanisha.pm@gmail.com, Tel.: +91-9623897159*

Available online at: www.ijcseonline.org

Accepted: 14/May/2018, Published: 31/May/2018

Abstract— String matching algorithms are widely used in computer science fields for information retrieval, intrusion detection, music retrieval, database queries, language syntax checker, bioinformatics, DNA sequence matching and etc. The most common and well-known use of string matching algorithms is for bioinformatics. In bioinformatics the DNA sequences of the normal human being and matched with the DNA sequence of a person having viruses or any kind disease. The pattern of any disease or virus is matched with the normal DNA genome sequence. If the pattern is found in the sequence which is in the form of string it is considered that the human being or patient is having the tested disease. Thus the pattern is matched with the large amount of DNA sequence which is sometimes very complex and not easy to retrieve. Thus to get the result or matched pattern in the less time with more accuracy the algorithms such as Knuth-Morris-Pratt(KMP), Boyer-Moore, Brute Force, Rabin-Karp and other algorithms are used. This paper presents five string matching algorithms from which four are exact matching algorithms and one is approximate string matching algorithm (Edit Distance). The above listed algorithms complexity will be compared using the DNA dataset to find the appropriate algorithm with high quality time and accuracy[1].

Keywords— String Matching Algorithm, DNA sequence

I. INTRODUCTION

String matching algorithm is an important component of the bioinformatics which is the application of computer science which stores the large amount of human health data to analyse the day to day problems which are commonly seen in the humans. Since the world is digitizing day by day, every single data of a person is stored somewhere into the databases including their health records, treatments taken previously and etc. The biological sequence data are growing in these databases at the exponential rate. These records can be useful while diagnosing the human DNA to find the future prediction disease that may affect a human and the precautions to be taken to avoid them. Thus all these things can be done by matching the exact as well as the approximate pattern with the person's DNA sequence. As the sequences are too long which are stored in various databases this process is quite expensive. Manually it will be tedious and time consuming to match or diagnose the pattern in the sequence. The computational demands are to find the exact or the approximate pattern in the sequence in less time complexity and with the good accuracy[1].

Deoxyribonucleic (DNA) is a molecule that carries the genetic instructions used in the growth, development, functioning and reproduction of all known living organisms and many viruses. DNA stores biological information. The two DNA strands are called polynucleotides since they are composed of simpler monomer units called nucleotides. Each nucleotide is composed of one of four nitrogen-containing nucleases (cytosine [C], guanine [G], adenine [A] or thymine [T]). The nitrogenous bases of the two separate polynucleotide strands are bound together, according to base pairing rules (A with T and C with G). For example, ATTCGTAACTAGGCCTAAGTAACGATA. Since the DNA contains only 4 characters ATCG in the sequence it becomes easier for the algorithms to find the exact matching or the approximating matching pattern in the sequence of DNA[2].

The organization of this paper is as follows. Section II presents the related work of string matching algorithms. In section III, methodology of the algorithms and their implementation is described. In section IV, experimental results are shown. In section V, concludes case study work. Sections VI contain the references used in this paper[1].

II. RELATED WORK

The string matching algorithm can be used to detect the inappropriate or the unusual patterns in the normal human being DNA sequence that may raise the risk of the disease or any major health problem for the particular human. The patterns are matched with the genome databases sequences which are the long DNA sequences available in the various databases. To get the exact match of the pattern in the sequence the exact matching algorithms such as KMP, Boyer-Moore, Rabin-Karp and Naive algorithms are used.

Thus the KMP algorithm is also known as the DNA sequencing algorithm since it gives the match in the less time as compared to the other string matching algorithms. The approximate string matching algorithm gives the approximate pattern found in the string and the minimum number of edit operations we can perform to get the exact pattern from the approximate pattern we found in the string. Thus, the demand for approximation algorithms is increasing day by day which gives all possible approximate patterns to be found in the DNA sequence with less cost and in more efficient manner[1].

III. METHODOLOGY

String matching algorithm is very important topic for text processing systems. As the system contains large numbers of data in text the string matching algorithms are the main components of such systems. String matching algorithms consists of 2 components the string or text sequence and the pattern to be matched. The string is denoted by S= {s_0, s1, s2, s3......$s_{m\}}$ which contains m characters in it and the pattern is denoted by P= {$p_{0,}$ p1, p2......p_n} contains n number of characters in it[1][5]. Thus the characters in pattern are less than the characters present in the string or the text sequence. The pattern is scanned through the string or text sequence using the size of window (i.e. the size of pattern). If the pattern does not match it gets shifted by the specific number derived from the pattern characters and if the pattern gets matched with the characters present in the sequence it gets shifted by the pattern size. This is the normal functionality of the String matching algorithm. [1]

The part of the dataset used for the analysis is shown below:

GCAAGATAGGCTATGATATCAGCTATGCATAAGCT
GGTACCGAGCTCGGATCTAACGACTGTTGATTGGTC
AGCGGAAGATAGATCTACAGTTAGGACTTGCGACA
GTCCATTTAGATCTACGCTTGATGTACCGGGCCGCA
TCCTTAGATCTACGTGGGAAGTGTAGGGTTCCATTG
ATAGATCTACACACTGACCCCCCATTCACCCCAATA
GATCTAGCCCACCCCTTCTACAAAACCCTACTAGAT
CTATCGGGTGGTTCTAATGCGGCACTTCTAGATCTA
TTCGTAGCCCCTCATCGCGCAATTCTAGATCTAGCC
TTCGTTTTTGTCAAGCTGCCCCTAGATCTAACACCT
ATGCGTCCCCCTCCCCCACTAGATCTACGACAGGCG
GTCGATCAATTTTGGTTAGATCTAAGGGGTTAAATT
GGGACGTGTGGAGTAGATCTACAATGCGTGATGTA
ATCGTGACCATTAGATCTAACGCAATGTTTCTCGCG
TCCCTCGCTAGATCTATCAGAATTTTAGGGATAGGA
CTATCTAGATCTACCGAGGGGGGCAGGTGTCGTTGT
CATAGATCCACTAGTAACGCCGCCAGTGTGCTGGA
ATTCTGCAGATATCCATCACAATGGCGGACGCTCGA
GCATGCATCTAGAGGCCCAATTCAGCCTATAGTGAG
TCGTATTACAAGTCACTGGCCGACGTCATACACGTC
CGGACTGGACAACCTGGCGTACCCCCTTAACGCTGG
AGGCATCCCTTTCAGTGGGAAAAAAAGAGGCAAGA
ACGCTCACAAATGGGACCAAGCAGAAGGACAAAAA
ACCGCCTCGGAGCAATCAACGCAGAACACCCTATA
ACTTTACAGCTTTCCCTGTTCCACGCNNAAAGACAC
AACAGAA[7][8].

The String Matching Algorithms presented in this paper are described below:

1. Naive (Brute Force) Algorithm:

The naive string matching algorithm is also known as Brute Force algorithm. This is the most simple string matching algorithm. This algorithm starts matching the pattern and sequence characters from left to right. This algorithm matches the each character of pattern with the corresponding character of string by shifting the pattern with 1.The algorithm stops when the matched pattern is found successfully in the sequence or their occurs a mismatch till the end of the sequence. If the pattern is of length m and the text sequence is of length n the total time complexity to search the pattern in the text sequence is O(n*m). Thus this process is time consuming. The search function of Brute force is given below[1][3][4].

Table 1. Naïve String matching Algorithm

INPUT: String str, Pattern pat

OUTPUT: Starting string str index where pattern pat matched

```
OUTER_LOOP:

for (int i = 0; i <= N - M; i++) {
   count = 0;
   for (int j = 0; j < pat.length; j++) {
      if (pat[j] == str[i + j]) {
         count = count + 1;
      }
   }
   if (count == pat.length) {
      System.out.println("Pattern found at index=" + i);
      break OUTER_LOOP;
   }
}
```

2. Rabin-Karp Algorithm:

The Rabin-Karp algorithm is the exact string matching algorithm. This algorithm uses the pre-processing function to generate the numeric value for the pattern. In pre-processing the algorithm has the numeric value for each character (i.e. from 1 to 26 for A to Z (same goes for the small alphabets also)). The numeric value is calculated using the character value and the smallest prime number corresponding to the number of characters in the pattern. If the pattern contains 3 characters the prime no. for that pattern will be 3, if the pattern contains 4 or 5 characters the prime number should be 5 or greater than 5. Thus, the selection of prime number for calculation should be at least equal to or greater than the length of the pattern. Thus, this removes the collision of values for the pattern to be matched and the mismatched pattern. The example using collision and without collision of values is given below. If the alphabets in the pattern gets repeated this also gives the different numeric value as compared to the value generated for the non-repetition of the alphabets in the pattern. The example of both is given below. The Rabin-Karp algorithm works from left to right. Since the algorithm works on the numeric value of the pattern, it selects the characters from the sequence equal to the length of the pattern. After selecting the characters equal to the length of the pattern, the algorithm generates the numeric value of selected characters form the sequence. If the value gets matched the pattern is found in the sequence else the pattern is not found.

To find the numeric value of the while sequence, the value of 1st substring is calculated the same way the value of pattern is obtained. For the remaining selected substring the value of the previously calculated substring and the calculations are made. The steps of calculation are explained below.

i. Take the pattern characters one at a time.
ii. Assign the character the value in power of prime number starting from 0 for the first character.
iii. Select the appropriate character serial number and multiply the number with power of prime number.
iv. Repeat Step 1 to 3 for all the characters of the pattern.
v. Add the values of all the characters obtained to get the numeric value of the whole pattern.

There are 2 methods in Rabin-Karp to find the string matching.

i) Take series number of the alphabets:
In this the numeric value of the alphabets is considered.

For example, Pattern: becd

b = 2,
e = 5,
c = 3,
d = 4

ii) Frequency count:
In this method the count of alphabets in the pattern is considered. The alphabets are arranged in the descending order of their count. To calculate the numeric value the serial no. is considered of the alphabet.

For example, String: engineering_college and Pattern: leg

Sr. no.	Alphabets	Frequency Count
1	E	5
2	G	3
3	N	3
4	I	2
5	L	2
6	C	1
7	O	1
8	R	1
9	_	1

The pre-process for calculating the numeric value of the pattern for both the methods is same.

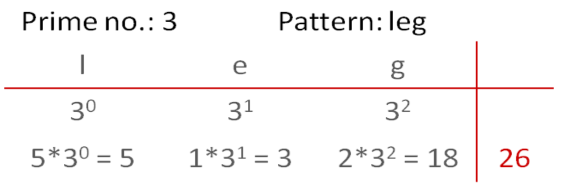

Figure 1. Pattern value calculation for Rabin-Karp

The **Error! Reference source not found.**., shows how the pattern is calculated using the series number of the alphabets with the prime number 3. The pattern value obtained is 26.

Table 2. Rabin-Karp String matching Algorithm

INPUT: String str, Pattern pat

OUTPUT: Starting string str index where pattern pat matched

```
for(int k=0;k<string.length-1;k++){
        if(k==0){
     while(x<pat.length) {
       curr_char = string[x];
       curr_char_unique_index =
                           unique_chars.indexOf(curr_char);
       curr_char_unique_index_value =

         unique_chars_index.get(curr_char_unique_index);
       sum_prev +=
                           curr_char_unique_index_value
                           * Math.pow(prime_no, x);
       x++;
      }
      if(sum==sum_prev){
        System.out.println("Pattern found at index="+k);
      }
   }else{
     int chck=k+pat.length-1;
     if(chck==string.length){
        break;
        }else{
          sum_str=0;
          curr_char=string[k-1];
          last_char=string[k+pat.length-1];
          curr_char_unique_index=
                                  unique_chars.indexOf(curr_char);
          last_char_unique_index=
                                  unique_chars.indexOf(last_char);
          curr_char_unique_index_value=

             unique_chars_index.get(curr_char_unique_index);
          last_char_unique_index_value=

             unique_chars_index.get(last_char_unique_index);
          sum_str=
             (((sum_prev-curr_char_unique_index_value)/
             prime_no)+(last_char_unique_index_value
             *Math.pow(prime_no,pat.length-1)));
          sum_prev=sum_str;
          if(sum==sum_str){
             System.out.println("Pattern found at index="+k);
          }
        }
     }
}
```

3. Knuth-Morris-Pratt (KMP) Algorithm:

The idea behind KMP algorithm is, whenever the mismatch occurs between the pattern and the text sequence characters we shift the pattern by the number of characters matched previously during traversal. Since we know the characters of the text sequence prior to mismatch, we can shift the pattern in such a way that, the new pattern matching starts from the mismatched index character by incrementing one to it[3][4][5].

The algorithm starts matching the pattern with the text sequence from left to right. The shift value is obtained from numeric value of the pattern characters. The numeric value is finding using the long prefix matched of the characters from the current position of the characters. In the prefix of the particular character and the suffix of the same character, some characters must be same. Thus this increases the shift of the pattern and the time from O(n^2) to O(n+m)[3].

The idea behind this to increase the shift by more than 1 and to reduce the complexity. While finding the prefix value of the pattern, if the prefix character does not match the value of that character is 0. While finding the prefix value, we always check from the starting characters of the pattern. To find the prefix value, the value of 1st character of pattern will be always 0.Increment the position of the pointer by 1. Match the character with the previous character of the pattern. If match found the character value will be the number of characters matched previously. The shift of the pattern will be calculated by subtracting the previous character index and the previous prefix value, where the mismatched occur[1][3][4][5].

Table 3.KMP String matching Algorithm

INPUT: String str, Pattern pat

OUTPUT: Starting string str index where pattern pat matched

Pattern Prefix Value Function:

```
int[] val = new int[pat.length];
int curr_index = 1;
int j=0, count = 0;
val[0] = 0;
int M = pat.length;

while (curr_index < M) {
   int x=0;
   for(;x<curr_index;x++){
      count=0;
      int a=curr_index;
      int b=x;
      while(b>-1){
         if(pat[a]==pat[b]){
            count=count+1;
            val[curr_index]=count;
         }else {
            break;
        }
         a--;
         b--;
      }
   } curr_index=curr_index+1;
}
```

Search Pattern Function:

```
OUTER_LOOP:
for (s = 0; s < N; ) {
   int count = 0;
   for (int j = 0; j < pat.length; j++) {
      if (s + j >= N) {
         break;
      } else {
         if (pat[j] == str[s + j]) {
            count = count + 1;
         } else {
            s += (j + 1) - (val[j]);
            break;
         }
      }
   }
   if (count == pat.length) {
      System.out.println("Pattern found at index=" + s);
      s += pat.length;
      break OUTER_LOOP;
   }
}
```

4. Boyer-Moore (BM) Algorithm:

The Boyer-Moore algorithm is the fastest string matching algorithm. The complexity of this algorithm is minimum. The algorithm starts scanning the pattern from right to left (i.e. the matching starts from the nth character of the pattern instead of the 1st.). This increases the shift more as compared to the KMP algorithm shift. In KMP the prefix value of the character is found for the shift. But in BM algorithm the method is different. In BM the value of the character is obtained by subtracting the 1 and the index value of the character from the total length of the pattern. The special symbol (*) asterisk is added at the end of the pattern. The value of the asterisk symbol and the last character of the pattern is equal to the length of the pattern. If the last character of the pattern and the 1st character of the pattern is same, then the value of 1st character will be equal to the last character of the value of the pattern. After constructing the value table, the unique characters of the pattern are found including the asterisk symbol. The values of these characters will be taken from the value table. If any character is repeated in the pattern, then the latest value of that character is considered from the LHS[3].

In Boyer-Moore algorithm the mismatch character of text sequence is considered rather than the character of pattern which done in KMP and Naive algorithms. Then the value of the mismatch character is found in the value table using unique characters value. If the mismatch character is not present in the unique characters, the value of asterisk is taken as the shift value, and the pattern is shifted using the same[3][4][5].

Table 4.Booyer-Moore String matching Algorithm

INPUT: String str, Pattern pat

OUTPUT: Starting string str index where pattern pat matched

<u>Pattern Numeric Value Function:</u>

```
for(int i=0; i<=pat.length;i++){
   index[i]=i;

   if(i<pat.length-1){
      val[i]=pat.length-1-index[i];
      patnew.add(pat[i]);
   }

   if(i==pat.length-1){
      val[i]=pat.length;
      patnew.add(pat[i]);
   }if(i==pat.length){
      patnew.add('*');
      val[i]=pat.length;
   }     if(!charbuff.contains(patnew.get(i))){
         unique_char.add(patnew.get(i));
         charbuff_val.add(val[i]);
      }else{
        int ind= unique_char.indexOf(patnew.get(i));
        charbuff_val.remove(ind);
        charbuff_val.add(ind,val[i]);
}
   j++;
   charbuff.add(patnew.get(i));
}
```

<u>Search Pattern Function:</u>

```
OUTER_LOOP:
 while(s<=n){
   int j=m-1;

     if(s+j>n-1){
       break;
     }else {
       while(j>=0 && pat[j]==str[s+j]){
          j--;
       }
      if(j<0){
          System.out.println("Pattern found at index:"+ (s));
          break OUTER_LOOP;
              }else{
                     if(!unique_char.contains(str[s+j])){
                           int u=unique_char.indexOf('*');
                           int v=charbuff_val.get(u);
                           s+=v;
                       }else{
                         int u=unique_char.indexOf((str[s+j]));
                          int v=charbuff_val.get(u);
                          s+=v;
                  }
              }
          }
   }
```

5. Edit Distance Approximation String Matching Algorithm:

The Edit Distance approximation algorithm is the approximation string matching algorithm whose objective is to find out the minimum edit distance between string and pattern, so that after performing the edit operations the string gets exactly matched with the pattern.

E.g. Let SATION is the pattern and ZATION be the string. Thus to get the string and pattern exactly matched, the minimum edit distance is 1 here. That is updating Z into S.

S A T I O N

Z A T I O N

There are 3 types of Edit Operations:

1. Insert
2. Delete
3. Update

The second objective of the edit distance algorithm is,

1. to find minimum distance,
2. location of the pattern (i.e. approximate available pattern in string),
3. Travel path to trace the pattern in the string.

To find the approximate string in the text sequence, the following steps need to be followed:

Step 1: Find the length of the string and pattern

Step 2: Create matrix where,
rows = length of pattern + 1
columns = length of string sequence + 1

Step 3: Enter 0 in the 0th row of the matrix.
Enter 0....m in the 0th column of the matrix.
where,
m=length of pattern

Step 4: To fill the cell of the matrix below formula is used,

$$e(x,y)=\min[(e(x-1,y-1)+\delta(x,y),\ e(x,y-1)+1,\ e(x-1,y)+1)]$$

where, $\delta(x,y)$=update cost (i.e. If update, then 1,if not update then 0)

Suppose we have matrix[i][j], where i=3 and j=4. Hence to fill the cell of the matrix where i=3 and j=4, we need to take 3 cells of the matrix into consideration,
i.e. matrix[i-1][j-1] = matrix[2][3] + 1,
matrix[i][j-1] = matrix[3][3] + 1,
matrix[i-1][j] = matrix[2][4] + 1,
hence using the value of above cells, if the character at pattern index 3 and the character at string index 4 does not match, we need to take the min value from the above cells and fill the matrix[3][4] with the min value[4][6].

If the character at pattern index 3 and character at pattern index 4 gets matched, then the diagonal value is taken. In this way, the whole matrix gets evaluated.

Step 5: Find the minimum edit distance.
The minimum edit distance is the minimum value of the last row.

Step 6: Travel path to travel the pattern in string.
Once the minimum value is obtained, traverse the path until we reach the 0th row (i.e. from last row traverse upwards). While traversing, again the e(x-1,y-1), e(x,y-1) and e(x-1,y) is taken into consideration.

Step 7: Find the positions where the edit operations need to form.
While traversing, if we move,

i. To the upper cell (e(x-1,y)), then, the insert operation is to be performed at that position.

ii. To the diagonal cell (e(x-1,y-1)), then, 2 conditions need to checked,
   a. If value of diagonal and current cell gets matched, then, there is a match of the character.
   b. If value of diagonal and current does not matched, but still, the diagonal is taken, there the update operation needs to be performed.

iii. To the left cell (e(x, y-1)), then the remove/delete operation is to be performed at that position[3][6].

Table 5.Edit Distance Approximate String Matching Algorithm

INPUT: String str, Pattern pat

OUTPUT: Finds the approximate string with respect to input pattern and gives the minimum edit distance to make the obtained pattern equal to the desired pattern

Generate Matrix Function:

```
public void Generate_Matrix(char[] str, char[] pat) {
   //i,j-1    Insert/Remove
   //i-1,j    Insert/Remove
   //i-1,j-1  Update
   int M = str.length;
   int N = pat.length;
   int matrix[][] = new int[pat.length + 1][str.length + 1];

   for (int i = 0; i <= N; i++) {
      matrix[i][0] = i;
   }

   for (int j = 0; j <= M; j++) {
      matrix[0][j] = 0;
   }

   int i = 1;

   while (i <= N) {
      for (int j = 1; j <= M; j++) {
         if (pat[i - 1] == str[j - 1]) {
            matrix[i][j] = matrix[i - 1][j - 1];
         } else {
            matrix[i][j] = min(matrix[i - 1][j - 1], matrix[i][j - 1], matrix[i -
1][j]);
         }
      }
      i++;
   }
   getLastRow_MinValue(matrix,pat,str);
}
```

Finding Approximate String Function:

```
public void Get_Approximate_Matched_String
(int[][] matrix, int index,char[] pat,char[] str) {

    ArrayList<Integer> approximate_str_index = new
ArrayList<Integer>();
    int i = pat.length;
    ArrayList<Character> approximate_str = new ArrayList<Character>();
    ArrayList<String> char_oprations = new ArrayList<>();
    int count = 0;

    while (i > 0) {
        if (!approximate_str_index.contains(index - 1)) {
            approximate_str_index.add(index - 1);
        }
        index = minValue_index(matrix, i, index, char_oprations);
        i--;
    }

    for (int j = approximate_str_index.size() - 1; j >= 0; j--) {
        approximate_str.add(str[approximate_str_index.get(j)]);
    }
    for(int a=0;
a<approximate_str.size();a++){
        System.out.println(approximate_str.get(a));
    }

    for (int k = 0; k < char_oprations.size(); k++) {
        System.out.println("The edit operations to be performed are:" +
char_oprations.get(k));
    }

  }
```

IV. RESULTS AND DISCUSSION

In this section, the comparison of all the algorithms with some specific examples is tabularized below. The below table explains the best, worst and average complexity of the algorithms using the same pattern and the string. The complexity shown in the below table is in the form of milliseconds while searching the pattern in the string at the starting of the string, middle of the string and at the end of the string.

Table 6. Comparison Results of Exact String Matching Algorithms

Algorithm			Naive	Rabin-Karp	KMP	Boyer-Moore
String 1	Pattern 1	Best Case	0.042865	1.387940	0.025961	0.039242
	Pattern 2	Avg. Case	0.391821	1.569700	0.111690	0.421404
	Pattern 3	Worst Case	1.112676	3.796864	0.263227	0.479967
String 2	Pattern 1	Best Case	0.043468	1.076453	0.042261	0.46488
	Pattern 2	Avg. Case	0.386991	1.151919	0.111087	0.133425
	Pattern 3	Worst Case	0.710591	1.345716	0.160593	0.154555
String 3	Pattern 1	Best Case	0.040450	1.058340	0.033809	0.032602
	Pattern 2	Avg. Case	0.374917	1.075245	0.120143	0.222776
	Pattern 3	Worst Case	0.687650	1.086112	0.161197	0.351371
String 4	Pattern 1	Best Case	0.048903	1.042643	0.031394	0.027771
	Pattern 2	Avg. Case	0.382765	1.064377	0.112898	0.176893
	Pattern 3	Worst Case	0.725081	1.119317	0.168441	0.365257

V. CONCLUSION

This paper provides the comparative study of the exact string matching algorithms and the approximate string matching algorithm with respect to DNA sequence. These algorithms are very effective in the DNA sequence matching in the bio filed since the DNA sequences are very huge and complex to retrieve easily. The comparative study shows the best case, average case and worst case time complexities of the exact string matching algorithms.

The best case time complexity for Naïve Algorithm is in the range of 0.040 milliseconds and worst case is in the range of 0.725 milliseconds. The best case time complexity for Rabin-Karp Algorithm is in the range of 1.042 milliseconds and worst case time complexity is in the range of 3.796 milliseconds. Since the calculation is done to find the pattern number and string numeric value to be matched in Rabin-Karp, the complexity of this algorithm is always more as compared to the matching algorithms. The best case time complexity for KMP algorithm is in the range of 0.0259 milliseconds and the worst complexity is in the range of 0.2632 milliseconds. The best case time complexity for Boyer-Moore algorithm is in the range of 0.0277 milliseconds and the worst case complexity is in the range of 0.4799 milliseconds. Thus according to the comparative study of various exact string matching algorithm, the best suited algorithm of DNA sequence matching is KMP algorithm for best case time complexity as well as for worst case complexity. This paper also presents the implementation of the Edit Distance Approximate string matching algorithm to find the approximate pattern in the DNA sequence. According to the implementation of this algorithm it takes 0.100219 milliseconds (i.e. 100219 nanoseconds) to find the approximate string with respect to pattern in the DNA sequence. Finally this paper gives optimal result according to each result.

REFERENCES

[1] NYO ME TUN, THIN MYA MYA SWE, "*Comparison of Three Pattern Matching Algorithms using DNA Sequences*", IJSETR, Vol.3, Issue.35, pp.6916-6920, 2014.

[2] https://en.wikipedia.org/wiki/DNA

[3] Robert Sedgewick, Kevin Wayne, "*Algorithms*", Fourth Edition, Addison-Wesley,Pearson Edition, India, pp. 760-776, 2011.

[4] Thomas Cormen,Charles E. Leiserson,Ronald L. Rivest,Clifford Stein, "*Introduction to Algorithms*", McGraw-Hill Publication, India, pp.909-926, 2001.

[5] Raju Bhukya, DVLN Somayajulu, "*Exact Multiple Pattern Matching Algorithm using DNA Sequence and Pattern Pair*", International Journal of Computer Applications, Number 8,Article 6, pp.32-38, 2011.

[6] Petteri Jokinen, Jorma Tarhio and Esko Ukkonen, "*A Comparison of Approximate String Matching Algorithms*", Software-Practice and Experience, Vol.1(1), pp.1-4, 1988.

[7] http://ccg.vitalit.ch/cgibin/htpselex/show?htpselex&tf=NF1_1&clone

[8] https://archive.ics.uci.edu/ml/machine-learning-databases/molecular-biology/promoter-gene-sequences/promoters.data