Pattern Matching Algorithm in DNA Sequence Analysis

[1]GHANSHAYM CHAURASIA, [2]SARITA SONI (Supervisor)

[1]M. Tech, BBAU, Lucknow
[2]Assistant Professor, BBAU Lucknow

Abstract:. DNA sequences has been for years a large worry for many research papers in Bio-Informatics.The DNA sequence is a long chain of characters specifying the nucleotides presented in the DNA. In bioinformatics the most well-known application is DNA sequence detection. Stored DNA sequence of several diseases is retrieved and compared in order to find out for the existence of a disease.To seek for the pattern a well-based pattern matching algorithm is called for in parliamentary procedure to fix the result at the monetary value of sufficient quantity of time.We've specifically mentioned the DNA sequences instead of any text strings and put through the algorithms upon it.This report evaluates four patterns matching algorithms' performance and then nominates a novel algorithm based upon a Rabin Karp algorithm which ensures that character comparisons can be eradicated from the Rabin Karp algorithm. These algorithms look for the specified pattern in a huge strand of DNA sequence.

Index terms: DNA sequence , Pattern matching , Rabin Karp algorithm, Human Pathogen, String Matching, Virus Detection.

I. INTRODUCTION

Fundamental ideas of this paper, i.e. the pattern matching problem has been talked about.In subsequent sections i.e. in section 6, 7, 8 and 9, the Brute Force, Knuth-Morris-Pratt algorithm, Boyer Moore algorithm and Rabin Karp algorithm respectively has been drawn. In part 10 we've described our idea to improve the Rabin Karp algorithm and in section 11 the references used in this paper have been dedicated.

II. PRELIMINARIES

Every human has his/her singular flair.Genes are made up of DNA sequences.DNA is held in each living cell of an organism, and it is the carrier of that organism's genetic code. The transmitted code is a set of Sequences which define what proteins to establish inside the organism.DNA consists of two filaments,

each being a chain of four nitrogenous bases, i.e. Adenine, Cytosine, Guanine and

Thymine. In a computer we represent each nitrogen base with a single character: A for Adenine, G for Guanine and C for Cytosine and T for Thymine. Thymine (T) & Adenine (A) always come in couples. Likewise, Guanine (G) & Cytosine (C) bases come in concert as well. Using these codes an entire DNA can be coded based upon their nucleotides contained in a filament. For example: ATGCGATATGCATGCATGCATAT. The term DNA sequencing comprehends biochemical methods for defining the parliamentary procedure of the nucleotide bases, adenine, guanine, C, and thymine, in a DNA oligonucleotide [10].Squaring up the DNA sequence is thus useful in basic

research studying fundamental biological processes, as considerably as in applied domains such as diagnostic or forensic research.The force and simplicity of using sequence information has, however, made it the method of choice in the modern Bioinformatics analysis. [11]

III. DISEASE CAUSED BY GENETIC FACTORS

An unhealthy symptoms or a specific illness in the body is termed as a disease. Disease refers to any unnatural condition of an organism that affects normal functions. The disease may be referred to disabilities, disorders, syndromes, symptoms [9]. Genes are the basic building blocks of genetic endowment. They get given from parent to kid. They contain DNA, the instructions for building proteins. A familial disease is any disease that is induced by an abnormality in an individual's genome. Some of the genetic disorders are inherited from the parents, while other genetic diseases are induced by mutations in a pre-existing gene or group of factors.

IV. DETECTION OF DISEASE USING PATTERN MATCHING

Over the final decade, genetic studies have

identified numerous associations between chromosomal alleles in the human genome and important human diseases.Alas, thus expanding

findings of casual variants in the region of DNA is not a straight forward task [8]. Causal variant identification typically involves searching through sizable regions of genomic DNA in the locality of disease-associated SNPs (single nucleotide Polymorphism). When we know a

particular sequence is the causal agency for a disease, the trace of the succession in the DNA and the number of occurrences of the sequence specifies the strength of the disease. As the DNA is a large database, we require to go for efficient algorithms to discover out a particular sequence in the given DNA.

V. THE PATTERN MATCHING PROBLEM

In pattern-matching problem with strings, we are handed a text string T of length n and a pattern string P of length m, and want to find whether P is a substring of T. The significance of a "match" is that there is a substring of text T starting at some index I that matches pattern P, so that T [I] =P [0], T [i+1] =P [1]... T [i+m-1] =P [m-1] i.e. P= T [I.. i+m-1].Therefore, the end product of a pattern-matching algorithm is either an indication that the pattern P does not exist in T or the starting index in T of a substring matching P. [12]

T =" abacaabaccabacabaabb "

And the pattern string:

P = "Abacab".

Then P is a substring of T. Videlicet, P = T [10...15]. At that place are various pattern-matching algorithms. Here we are to review four patterns matching algorithms and prove an algorithm which is based upon Rabin-Karp algorithm, but modified. These efficient algorithms can be used to observe the sequence of DNA in a huge genetic database. Chase are the four algorithms which are described below.

- Brute-Force
- Knuth-Morris –Pratt
- Boyer-Moore
- Rabin-Karp Algorithm

VI. BRUTE FORCE ALGORITHM

It is as well known as Naive String Matching algorithm and not need for pre processing phase , needs contant extra space. It always shifts the window by exactly one position to the right. It requires 2n expected text characters comparisons. It detects all valid shifts using a closed circuit that determines the condition P [1.... m] =T [s+1... s+m]
for each of the n-m+1 possible values of s. The algorithm is the following:

BRUTE_FORCE(T, P)

n ← length[T]

m ← length[P]

for s ← 0 to n − m

do if P[1 . .m] = T [s + 1 . . S + m]

then print "Pattern comes with shift" s

The Brute force string-matching operation can be presented as shifting the shape over the text, observing for which switches all of the parts of the pattern equal the corresponding references in the text, as instanced in the accompanying case.

T=ANPANMAN

P=MAN

Complexity

Procedure BRUTE_FORCE takes time O (m) in best case, i.e. when the blueprint is found within first my characters of text. And inward the worst case the rule will be matched total (m (n-m+1)). For Object lesson, see the text string "AN" (a chain of n a's) and the pattern "AM".For each of the (n−m+1) possible values of the shifts, the loop on line 4 to compare corresponding characters must execute m times to validate the transformation. The worst-case running time is therefore O (mn).The playing time of BRUTE_FORCE is equal to its matching time, since there is no pre-processing.

Drawbacks OfThis Approach

In O(mn) approach. if 'm' is the length of pattern 'p' and 'n' is the length of string 'S'. Suppose S=ATGATAATGAAG and p=AATA.

Figure1: Brute Force comparison process

j=	0	1	2	3	4	5	6	7	8	9	10
S=	A	T	G	A	T	A	A	T	G	A	G
p=	A	T	A	A							
		A	T	A	A						
			A	T	A	A					
				A	T	A	A				

In table 1 we've shown when a mismatch is discovered for the beginning time in comparison of p [3] with S [3], pattern 'p' would be moved one place to the right and matching procedure resumes from here. Here the first comparison that would take place would be between p[0]='A' and S[1]='T'. It should be remarked here that S [1] had been previously implied in a comparison in 2nd iteration of the loop in this algorithm. This is a repetitive use of S[1] in another comparison. It is these repetitive comparisons that run to the runtime of O (mn), which fixed it really boring.

VII. KMP ALGORITHM

We now give a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. The basic thought behind the algorithm discovered by Knuth, Morris, and Pratt is this: when a mismatch is found, our false start (which is the primary drawback of the Brute Force algorithm) consists of roles that we know in advance (since they're in the form).Somehow we should be able to take advantage of this data instead of backing up the pointer over all those known parts

VII. I The Prefix Function

For A Pattern Fully skipping past the pattern on detecting a mismatch as described in the previous paragraph won't work when the pattern could match itself at the tip of the mismatch. To compute the positions of the pattern as to how much a rule need to reposition itself so that the corresponding qualities of text paired with it. The table is called as next table or sometimes failure function (figure 2) for the blueprint to be searched [14].Take some other example of this next table. This next [j] is the character position in the design which should be held next after such a mismatch, then that we can slip the rule (j - next [j]) places relative to the text [6].

Figure 2: Next table

j	1	2	3	4	5	6	7	8	9	10
pattern	A	T	G	A	T	G	A	G	A	T
Next	-1	0	0	-1	0	0	-1	4	-1	0

Here next [j] = 0 means that we are to slide the shape all the way past the current text character. Today we shall talk about how to pre compute this table; fortunately, the computations are quite simple, and we will discover that they require only O (m) steps. Now we know why represent following the algorithm to compute the following function or prefix function:

next (p)//p signifies pattern,

int I=0, j=-1;

```
next [I] =j;
for (I=0;i<m;i++)
{
if (I==0) next [I] =j;
else if (p [I] ==p [j])
{
next [I] =next [j];
}
Else
{
next [I] =j;
}
while (j>=0 && p [I]! =p [j])
j=next [j];
j++;
}
```

This program takes O (m) units of time, as next [t] in the innermost loop always shifts the upper copy of the form to the right, then it is performed a total of my times at most. A somewhat different way to examine that the running time is bounded by a constant times m is to observe that the variable starts at 0 and it is increased, m- 1 times, by 1; furthermore its value remains non negative. Thus the operation next [j], which always decreases j, can be performed at most m-1 times [6].

The Pattern Matching Algorithm

The Knuth-Morris-Pratt matching algorithm is presented in pseudo code below as the procedure KMP-MATCHER. KMPMATCHER calls the auxiliary procedure next() to compute next table. Below T & P signifies text & pattern respectively.

```
KMP-MATCHER (T, P)
n ←
length[T]  m
← length[P]
next=next(P)   //array consisting of prefix values
j ← 0          //Number of characters matched.
for k ← 1 to n //Scan the text from left to right.
Do while j > 0 and P [j + 1] ≠ T [k]
do j ← next [j] //Next character does not meet.
if P[j + 1] = T [k ] then j ← j + 1
                //Next character matches.
if j = m       //Is all of P matched?
Then print “Pattern comes with shift” k–m
j ← next [j]   // Look for the following match.
```

For convenience, let us presume that the input text is present in an array text T [1… n], and that the rule appears in pattern P [1… m].We shall as well assume that m > 0, i.e., that the pattern is non empty. Let k and j are integer variables such that text T [k] announces the current text character and pattern P [j] denotes the corresponding pattern character. Show that the design is fundamentally aligned with position p + 1 through p + m of the text, where k =p +j [15].

Complexity

The KMP algorithm works by reversing the patterns given into a machine, and then feeding the car. It requires O (m) space and time complexity in pre-processing phase, and O (n+m) time complexity in searching phase (independent of the alphabet size).

KMP linear time string matching algorithm. [6]

VIII. BOYER-MOORE ALGORITHM

A significantly faster string searching method can be built up by scanning the pattern from right to left when trying to match it against the textbook. The Boyer-Moore algorithm (BM) was produced by R. S. Boyer and J. C. Moore in 1977 [7].The Boyer Moore algorithm scans the characters of the pattern from right to left beginning with the rightmost one and does the comparisons from right to left.

VIII. IBad Character Rule

To convey the mind of the bad character rule, let us imagine that the final (rightmost) character of pattern P is you and the character in text T it aligns with is x, $x \neq y$. When a mismatch occurs, we can safely shift P to the right, so that the rightmost x in P is below the mismatched x in T, and this is possible if the rightmost side of character x exists in pattern P. This observation is formalized below
[16]. For a particular coalition of a pattern P against text T, the rightmost (n-i) characters of a pattern P match their counterparts in text T, but the next character to the left, P (I), doesn't match with its twin, say in position k of T. The bad character rule says that P should be shifted right by Max[1,i - R(T(k))] places. The stage of this shift rule is to shift P by more than single character when possible. Example, T (5) = t mismatches with P (3) and R (t) = 1 so P can be shifted right by two sides. Later on the switch, the comparison of P and T leads off again at the correct end of P.

Figure3: Compare from right

```
    1                   2
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
T A C T  C  T  T  G  A  T  G  C  T  C  T
T A  C
         A G A T G  A
P        T
```

VIII.II. Extended Bad Shift Rule

When a miss match occurs at position i of pattern P and the mismatched character in text T is x, then shift P to the right so that the closest x to the left of position i in P is below the mismatched x in T.

VIII. III The Good Suffix Rule

Now we bring in another convention called the good suffix rule. Conjecture for a given pattern P and text T, a substring t of text T matches a suffix of pattern P, but a mismatch occurs at the side by side comparison to the left. Then determine, if it survives, the rightmost copy t' oft in P such that t0 is not a suffix of P and the character to the left of t' in P differs from the part to the left of it in P. Shift P to the right so that substring t0 in P is below substring t in T (see Figure 4). If t' does not survive, then switch the odd end of P. Past the left end of it in T by the least sum of money so that a prefix of the shifted pattern matches a suffix oft in T. If no such switch is possible, then shift P n places to the right. If an occurrence of P is found, then shift P by the least sum of money so that a proper prefix of the shifted P matches a suffix of the

occurrence of P in T. If no such switch is possible, then shift P by n places, i.e., shifting P past t in T.

Figure 4: case when good suffix rule applies

```
  0                    1
  1  2 3 4 5 6  7 8 9  0 1 2  3 4 5 6 7 8
T p r  s t  a b  c  t  u b a b v q x r s t
                       ^
                     B  d
P       q c  a b d  a a      b
        1 2  3 4  5 6 7  8 9 0
```

Good suffix shift rule, where character x of T mismatches with character y of P. Characters y and z of P are guaranteed to be distinct from the good suffix rule, so z has a prospect of matching x.When the mismatch occurs at location 8 of P and position 10 of T, t = ab and t0 occurs in P starting at location 3. Hence P is shifted right by six places resulting in the following alignment.

Figure 5: Shifting using good suffix rule

```
  0                  1
  1 2 3 4 5 6 7 8 9  0 1 2 3 4 5 6 7 8
 T p r s t a b c t u b     a b v q  x r  s t
                          ^
 P                    q  c a b d a b a b
                      1  2 3 4 5 6 7 8 9
```

At present in cases where we agree the final m characters of the pattern P before failing, we clearly wish to transfer our attention down string by 1+m.So, L(i) is the largest index j less than n such that Nj (P)≥|P[i..n]| (which is n - i + 1). L'(i) is the largest index j less than n such that Nj(P) = |P[i..n]|
= (n - i + 1).Now The preprocessing stage must also prepare for the case when L'(i) = 0 or when an occurrence of P is found. l'(i) equals the largest j ≤
|P[i..n]|, which is n-i+1, such that Nj (P) = j. Thus we can state that the required shift will be max (L (I), L'(I)).

The complete Boyer-Moore algorithm:

Given the pattern P, //pre-processing stage Compute L'(i) and l(i) for each position i of P, and compute R(x) for each character x ∈ ∑ //Search stage

```
        k := n;
        while k ≤ m do
    begin
        i := n;
        h := k;
while i> 0 and P(i) = T(h) do
    begin
        i := i - 1;
        h := h - 1;
    end;
if i = 0 then
begin
    report an occurrence of P in T
     ending at position k.
    k := k + n – l'(2);
  end
else
        shift P (increase k) by the maximum
            amount determined by the
        (extended) bad character rule and
the good suffix
    rule.
    end
```

Observe that although we have always talked about shifting P", and given rules to determine by how much P should be "shifted", there is no shifting in the actual implementation. Instead, the index k is increased to the degree where the right end

of the P would be shifted". Hence, each bit of shifting P takes constant time [17].

The good suffix rule in Boyer-Moore method has a worst-case running time of O (m) provided that the rule does not come out in the textbook. This was first proved by Knuth, Morris and Pratt [6].

Algorithm Complexity

The BM algorithm is successful at reaching a sub linear running time in the average case, and if some special conditions occurred and so also was capable of O (n+m) in the worst event.

IX. RABIN-KARP ALGORITHM

Previous three algorithms which we've seen is based upon string matching to see whether the pattern is matched with the text portion or not. RABIN KARP matcher is one of the most effective string matching algorithms. To obtain a numeric pattern 'P' from a given text 'T'. It first divides the figure with a predefined prime number 'q' to calculate the modular of the pattern P. Then it examines the first m characters (m=|P|) from text T to compute the residue of my references from text T. If the residue of the Pattern and the remainder of the text T are equal only then we compare the characters of the text portion with the pattern otherwise there is no need for the comparison [1].We've to recapitulate the procedure for the next set of references of text for all the possible shifts which are from s=0 to NM (where n denotes the length of text and m denotes the length of P).Thus according to these two numbers n1 and n2 can only be equal if

REM (n1/q) = REM (n2/q) [1]

After division, we will be having three cases: -

• Case 1: Successful hit: - In this case if

REM (n1) = REM (n2) and also characters of n1 matches with characters of n2.

• Case 2: Spurious hit: - In this case REM (n1) = REM (n2) but characters of n1 are not equal to characters of n2.

• Case 3: Compare the value of REM (n1) is not equal to REM (n2), then no need to compare n1 and n2.

For a given text T, pattern P and prime number q

T=23456789979779797653435667888675 6456890 975545343434 24545475655454

P=667888

q=11

So to find out this pattern from the given text T we will hire an equal number of cases from the textual matter as in the pattern and carve up the value of these characters with predefined number q and also split the practice with the same predefined number q.Now compare their remainders to decide, whether to compare the text with pattern or non.

REM (Text)

=234567/11=3 REM

(Pattern) =667888/11=1

As both the residues are not equal then there is no demand to compare text with pattern. Immediately move on to set of characters of the same length next from text and repeat the operation.

The Boyer Moore Algorithm goes as follows:

```
Rabin_Karp_Matcher (T,P,d,q)
  {
      n =Length (T)
      m= Length (P)
      t0=0
      p=0
      h=d^(m-1) mod q
      for i=1 to m
```

```
{
        p = (d * p + P[i]) mod q
        t0 =(d * t0 + T[i] ) mod q
  }
      for s =0 to n-m
    {
 if ts=p
        {
      //comparison for spurious
   hits if P[1….m] =
   T[s+1…….s+m]
                then print pattern
                        matches at
shift ‘s’
        }
      if s<= n-m
   ts+1= (d(ts-h*T[s+1]) + T[s+1+m] )
   mod q
    }
}
```

Thus the entire process can be written as follows: where Say P has a length L and S has length n. One room to search for P in S:

1. Hash P to get h(P).

2. If a substring hash value does match h (P), do a string comparison on that substring and P, stopping if they do match and continuing if they answer not.

IX.I Numerical Example:

Let’s step back from strings for a second. Say we have P and S be two integer arrays:

P = [5; 0; 3; 3; 0]

S = [4; 8; 5; 0; 3; 3; 0; 8]

The length 5 substrings of S will be denoted as such:

S0 = [4; 8; 5; 0; 3]

S1 = [8; 5; 0; 3; 3]

S2 = [5; 0; 3; 3; 0]

And so on…

We want to determine if P ever appears in S using the trio steps in the method above. Our hash function will be:

h (k) = (k [0] * 104 + k [1] * 103 + k [2] * 102 + k [3] * 101 + k [4] * 100)mod m

Or in other words, we will admit the length 5 array of integers and concatenate the integers into a 5 digit number, then select the number mod m. h(P) = 50330 mod m, h(S0) = 48503 mod m, and h(S1) = 85033 mod m. Notice that with this hash function, we can use h (S0) to help calculate h (S1).We start with 48503, chop off the first digit to get 8503, multiply by 10 to get 85033, and then add the next digit to obtain 85033. More formally:

h(Si+1) = [(h(Si) - (105 * first digit of Si)) * 10 + next digit after Si] mod m

We can imagine a window sliding over all the substrings in S. Counting on the hash value of the next substring. In this numerical example, we looked at single digit integers and set our base b = 10 so that we can understand the arithmetic easier. To generalize for other base band other substring length L, our hash function is h (k) = (k [0] b^{L-1} + k [1] b^{L-2} + k [2] b^{L-3}.... K [L - 1] b^{0}) mod m

And calculating the next hash value can be caused by:

h (Si+1) = b (h (Si) – b^{L-1}S [I]) + S [I + L] mod m

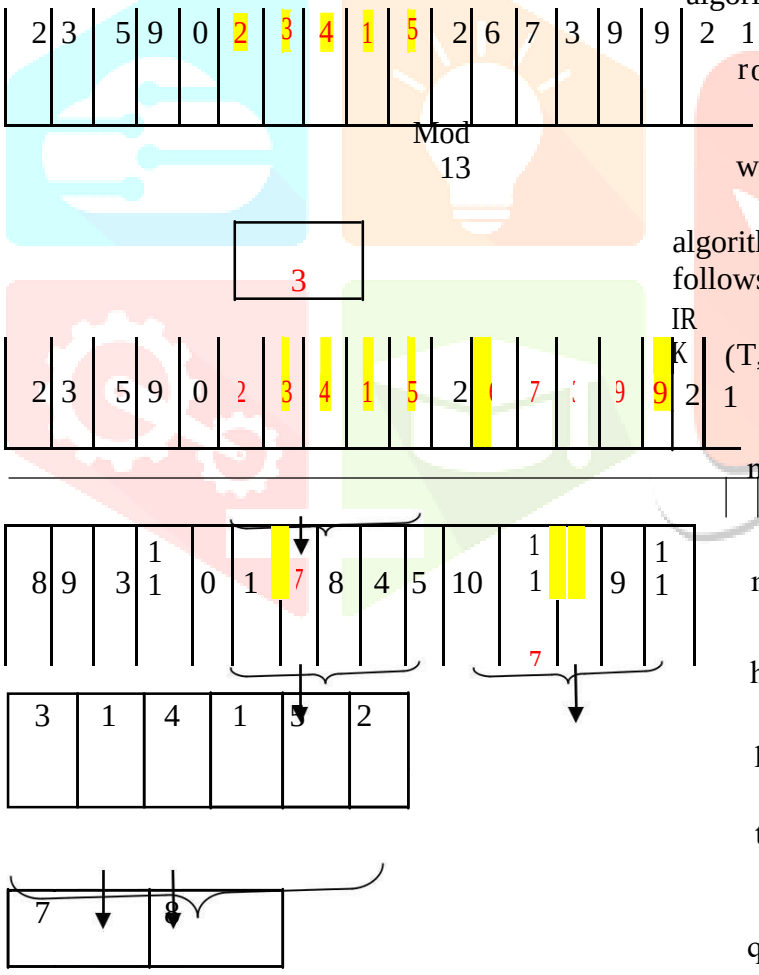
Following is the example taken from [15]:

Figure 6:

second, beginning at text position 13, is a spurious hit.(c) Computing the value for a window in constant time, given the value of the old window. The first window has value 31415.

X. IMPROVED IDEA:

Theory As we can see, spurious hit is an extra onus on an algorithm which increases its time complexity when we hold to compare the text with pattern and won't be capable to find the pattern at that switch.Thus to ward off this extra matching, we've improved the Rabin Karp algorithm slightly, called IRK algorithm which says that along with remainders compare the quotients also. That is IRK checks whether, REM (n1/q) =REM (n2/q) and QUOTIENT (n1/q) = QUOTIENT (n2/q), where n1= pattern & n2=Text & q is the prime figure. Thus, agreeing to this technique along with the computation of the remainder, we will also find out the quotient and if both remainder and quotient of text match with the pattern, then it is successful hit otherwise it is an unsuccessful hit or spurious hit and then we can take away the possibility of comparing the spurious hits. That implies there is no extra computation of spurious hits if remainder and quotient are same then pattern found else pattern not found.



Essentially the algorithm is same as the original robin Karp algorithm, only with small adjustments, which are expressed in bold italic type face. The algorithm works as follows:

IRK (T, P, d, q)

n ← length (T) //text length

m ← length (P) //pattern length

h ← d^{m-1} mod q

p ← 0

t0 ← 0

q_p ← 0 //quotient post hash calculation for pattern

The above human body [15] illustrates (a) A text string. A window of length 5 is shown shaded. The mathematical value of the shaded

The turn is computed modulo 13, yielding the value 7.(b) The same text string with values computed modulo 13 for each potential position of a length-5 window. Taking the pattern P = 31415, we look for windows whose value modulo 13 is 7, since 31415
≡ 7 (mod 13).The first, beginning at text position 7, is indeed an occurrence of the rule, while the

//quotient post hash calculation for portions of text of size m

q_t ← 0

for me ← 1 to m //Pre processing do

temp_p ← (d*p + P [I])

q_p ← temp_p / q
p ← temp_p mod q

temp_t ← (d*t0 + T [I])

q_t ← temp mod q

t0 ← temp mod q

for s ← 0 ton – m // Matching

//comparison only if the quotient matches, removal of spurious hit

do if p = ts &&q_p = q_t

then print “Pattern comes with shift” then

if s < n – m

//quotient, post hash calculation of next m characters in text.

temp_t ← (d * (ts – T[s + 1] * h) + T[s + m + 1]) / q

q_t ← temp_t /q

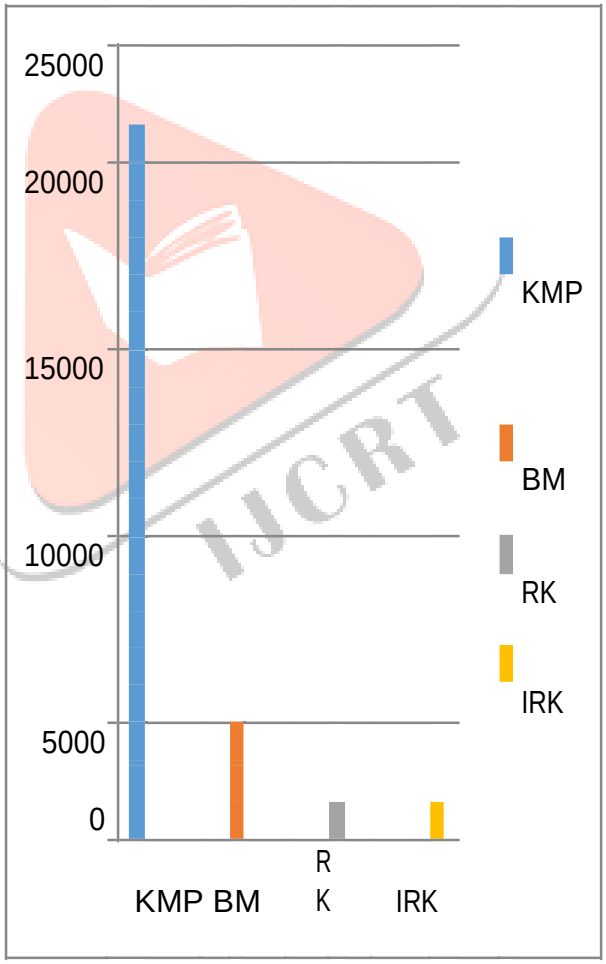
//subtracting LSB, Shifting and adding MSB then

ts+1 ← (d * (ts – T[s + 1] * h) + T[s + m + 1])mod q ts= ts+1

Comparison using Graphs:

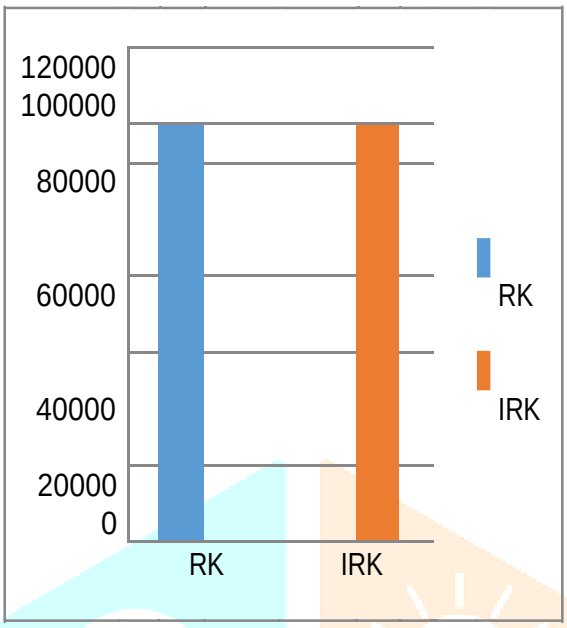
The results of our experiments are depicted in the graphs below. In the first graphs we have represented the performance of the algorithms with a specified text file size of 1MB. Y axis represents time in microseconds and X-axis represent the corresponding algorithms.

Figure7: comparison of algorithms with respect to 1 MB text file



Today we compare only between Rabin Karp and IRK algorithms with the same text file size of 1

MB, in **figure 8**: Comparison of Rabin Karp and IRK algorithms using file size 1MB.



Rabin Karp scores the running time of 100750 microseconds and IRK adjusted the operating time within 95500 microseconds, both upon same 1 MB text file. Below is the graph which depicts the comparison between Rk and IRK algorithm using a 2MB file size. Besides we've compared the algorithm upon 2 MB text file size, whose readings are as follows 260750 for Rabin Karp and 175250 for IRK algorithm.

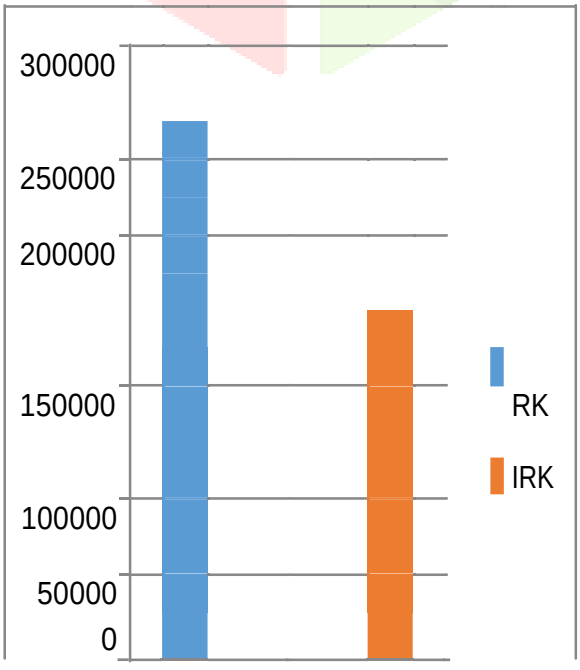


Figure 9: depicts the comparison of Rabin Karp and IRK algorithms using file size 2MB.

X.II Example of IRK algorithm:

T= ABBCABCA //text

P= BCA //pattern

q=13(say)

d=256 (for character)

Hash(P)= (66 * 2562 + 67 * 2561 + 65) mod q

p = 0 // hash value for pattern

q_p = 334045 //quotient

A	B	B	C	A	B	C	A

hash(ABB) = 0 // same

hash q_t0 = 328965 //but quotient different

A	B	B	C	A	B	C	A

hash(BBC) = 1

q_t1 = 334026

A	B	B	C	A	B	C	A

hash(BCA) = 0

q_t2 = 334045

A	B	B	C	A	B	C	A

hash(CAB) = 7

// both

q_t3 = 339047 matched

A	B	B	C	A	B	C	A

A hash(ABC) = 11

q_t1 = 328984

A	B	B	C	A	B	C	A

hash(BCA) = // hash
0 matched

q_t2 = 334045 // quotient matched

Since the hash =0 and quotient = 334045 both matched. But the pattern BCA is matched. And hash(ABB) = 0 and quotient = 328965, which has not matched, ABB is not compared.

X.3 Time Complexity

In Best case doesn't differ a lot from the original Rabin Karp algorithm, but the in average case complexity can be amended significantly. Due to imposing of constraint of matching the quotient post hashing as well as the hash value of the textbook portion of size m, reduction in comparison has been realized. Which cuts the time complexity during worst case from O ((n-m+1) m) to O (nm+1). This time complexity is hugely depends on the selected prime number, q.

XI. CONCLUSION AND FUTURE SCOPES

This version of the Rabin Karp algorithm can be used with Genetic Algorithm in order to look for a pattern in huge text files of size >500MB.Implementation using GA can produce an improved variation of this algorithm for more sophisticated exercise and can constitute the search even faster by applying the genetic operators such as selection, mutation, crossover etc. Our Future scope lies among this thinking that it could be possible for us to implement this IRK algorithm using GA for optimize the practice analysis. Further analysis and improvement of this algorithm is welcome from any scholars.

REFERENCES

[1] Richard M. Karp, Michael O. Rabin, Efficient Randomized patternmatching algorithms, International Business Machine, 1987

[2] Roberto Ierusalimschy, A Text Pattern-Matching Tool based on Parsing Expression Grammars, 2008

[3] Rajesh S., Prathima S., Reddy L.S.S., Unusual Pattern Detection in DNA Database Using KMP Algorithm, International Journal of Computer Applications (0975 - 8887), Volume 1 – No. 22, 2010

[4] Jiyeon Choi, Myka R. Ababon, Mai Soliman, Yong Lin, Linda M. Brzustowicz, Paul G. Matteson, James H. Millonig, Autism Associated Gene, ENGRAILED2, and Flanking Gene Levels Are Altered in Post-Mortem Cerebellum-PLOS ONE, 2014

[5] Gupta, A.R., and State, M.W. (2007) Recent Advances in the Genetics of Autism. Biological Psychiatry 61, 429-437.

[6] Donald E. Knuth, James H. Morris, Jr And Vaughan R. Pratt, FAST PATTERN MATCHING IN STRINGS, Vol. 6, No. 2, June 1977, SIAM J. COMPUT.

[7]R. Boyer and J. Moore, A fast string searching algorithm", CACM, 20, 10, 1977, pp.262-272.Ashish ProsadGope et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (4) , 2014, 5450-5457www.ijcsit.com 5455

[8] Christopher B. Kingsley, Identification of Causal Sequence Variants of Disease in the Next Generation Sequencing Era, Methods in Molecular Biology, Volume 700, 2011, pp 37-46.

[9] Melissa Conrad Stoppler MD (2014, Jan 15). Genetic Diseases Overview [Online]. Available: http://www.medicinenet.com/genetic_disease/article.htm.

[10] DNA Sequencing, Wikipedia, http://en.wikipedia.org/ wiki/Genetic_analysis#DNA_Sequencing

[11] Biological Databases, http://biotech.fyicenter.com /resource/Biological_databases.html

[12] Michael T. Goodrich; Roberto Tamassia; David M. Mount, 2011. Data Structures and Algorithms in C++, Second Edition

[13] Akhtar Rasool Amrita Tiwari et al, (IJCSIT) Vol. 3 (2) , 2012,3394 – 3397, International Journal of Computer Science and Information Technologies.

[14] Sedgewick, Robert, 1984-Algorithms., ADDISON-WESLEY PUBLISHING COMPANY 15. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein et al. 2009, 3RD edition, Introduction to Algorithms, MIT Press.

[16] Dan Gus field. COMPUTER SCIENCE AND COMPUTATIONALBIOLOGY, University of California, Davis, 2007

[17] Boyer-Moore Tutorial, The University of California, Davis, http://www.cs.ucdavis.edu/~gusfield/ cs224f11/ bnotes.pdf, 2007 .