ISSN:0975-9646

A Novel Pattern Matching Algorithm in Genome Sequence Analysis

Ashish Prosad Gope [#1], Rabi Narayan Behera[#2]

[#1]M.Tech Student (4th Semester) Dept. of Information Technology, Institute of Engineering & Management, Salt Lake, Kolkata, India

[#2]Asst. Professor, Dept. of Information Technology, Institute of Engineering & Management, Salt Lake, Kolkata, India

Abstract— **DNA sequences has been for years a great concern for many research papers in Bio-Informatics. DNA sequence is a long string of characters specifying the nucleotides presented in the DNA. In bioinformatics the most well-known application is DNA sequence detection. Stored DNA sequence of various disease are retrieved and compared in order to check for the existence of a disease. To search for the pattern a well-established pattern matching algorithm is needed in order to get the result at the cost of sufficient amount of time. We've specifically referred the DNA sequences instead of any text strings and implemented the algorithms upon it. This paper evaluates four pattern matching algorithms' performance and then proposes a new algorithm based upon Rabin Karp algorithm which ensures that character comparisons can be eliminated from Rabin Karp algorithm. These algorithms look for the specified pattern in a huge strand of DNA sequence.**

I. INTRODUCTION

Bioinformatics is an interdisciplinary research area that is the interface between the biological and computational sciences. The advent of electronic computers has arguably been the most revolutionary development in the history of Science and technology. The Ultimate goal of bioinformatics is to uncover the wealth of Biological information hidden in the mass of data and obtains a clearer insight into the fundamental biology of organisms. This new knowledge could have profound impacts on fields as varied as human health. Agriculture, the environment, energy and biotechnology. There are many other applications of bioinformatics, including predicting entire protein strands, learning how genes express themselves in various species, and building complex models of entire cells. As computing power increases and our databases of genetic and molecular information expand, the realm of bioinformatics is sure to grow and change drastically, allowing us to build models of incredible complexity and utility.

When we know a particular sequence is the cause for a disease, the trace of the sequence in the DNA and the number of occurrences of the sequence defines the intensity of the disease. As the DNA is a large database, I propose String and Pattern matching algorithms to find out a particular sequence in the given DNA. This paper entirely focuses on a novel approach for detecting the patterns present in the gene database. Pattern matching is a mechanism to find out the exact location of a specified pattern, iff the pattern exists in the text.

Before moving forward let us convey you about the structure of our paper. We've discussed the preliminaries needed to move forward in section 2, after that in section 3, disease caused by genetic factors has been revisited. In section 4 we discussed detection of disease using pattern matching and in section 5 the central ideas of this paper i.e. the pattern matching problem has been discussed. 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 described. In section 10 we've described our idea to improve the Rabin Karp algorithm and in section 11 the references used in this paper has been given.

II. PRELIMINARIES

Every human has his/her unique genes. Genes are made up of DNA. DNA is contained in each living cell of an organism, and it is the carrier of that organism's genetic code. The genetic code is a set of Sequences which define what proteins to build within the organism. DNA consists of two strands, each being a string 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 pairs. Likewise, Guanine (G) & Cytosine (C) bases come together too. Using these codes an entire DNA can be coded based upon their nucleotides contained in a strand. For example: **ATGCGATATGCATGCATGCATAT**. The term DNA sequencing comprehends biochemical methods for determining the order of the nucleotide bases, adenine, guanine, cytosine, and thymine, in a DNA oligonucleotide [10]**.** Determining the DNA sequence is therefore useful in basic research studying fundamental biological processes, as well as in applied fields such as diagnostic or forensic research.

The power and ease of using sequence information has however, made it the method of choice in modern bioinformatics analysis.[11]

III. DESEASE 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. Diseasemay be referred to disabilities, disorders, syndromes, symptoms[9].Genes are the basic building blocks of heredity. They get passed from parent to child. They hold DNA, the instructions for making proteins. A genetic disease is any disease that is caused by an abnormality in an individual's genome. Some of the genetic disorders are inherited from the parents, while other genetic diseases are caused by mutations in a pre-existing gene or group of genes.

IV. DETECTION OF DISEASE USING PATTERN MATCHING

Over the last decade, genetic studies have identified numerous associations between chromosomal alleles in the human genome and important human diseases. Unfortunately, these extending 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 cause for a disease, the trace of the sequence in the DNA and the number of occurrences of the sequence defines the intensity of the disease. As the DNA is a large database we need to go for efficient algorithms to find out a particular sequence in the given DNA.

V. THE PATTERN MATCHING PROBLEM

In pattern-matching problem on strings, we are given 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 meaning 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]. Thus, the output from 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. Namely, P = T [10...15]. There are various pattern-matching algorithms. Here we are to review four pattern matching algorithms and present an algorithm which is based upon Rabin-Karp algorithm but modified. These efficient algorithms can be used to trace the sequence of DNA in a huge gene database. Following are the four algorithms which are described below.

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

VI. BRUTE FORCE ALGORITHM

It is also known as Naive String Matching algorithm. It has no pre-processing phase, needs constant extra space. It always shifts the window by exactly one position to the right. It requires 2n expected text characters comparisons. It finds all valid shifts using a loop that checks the condition P[1....m]=T[s+1...s+m] for each of the n-m+1 possible values of s. The algorithm is as 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 occurs with shift" s
```

The Brute force string-matching procedure can be presented as shifting the pattern over the text, observing for which shifts all of the characters of the pattern equal the corresponding characters in the text, as illustrated in the following example.

T=ANPANMAN
P=MAN

VI.I. Complexity

Procedure BRUTE_FORCE takes time O(m) in best case i.e. when the pattern is found with in first m characters of text. And in the worst case the pattern will be matched total (m (n-m+1)). For example, consider the text string "AN" (a string of n a's) and the pattern "AM". For each of the (n−m+1) possible values of the shift s, the loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus O(mn). The running time of BRUTE_FORCE is equal to its matching time, since there is no preprocessing.

VI.II. Drawbacks Of This 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.

Figure 1: 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 mismatch is detected for the first time in comparison of p[3] with S[3], pattern 'p' would be moved one position 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 noted here that S[1] had been previously involved 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 lead to the runtime of O(mn), which made it very slow.

VII. KMP ALGORITHM

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

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 point of the mismatch. To calculate the positions for the pattern as to how much a pattern need to shift itself so that the corresponding characters of text match with it. The table is called as **next table** or sometimes **failure function** (figure 2) for the pattern to be searched [14]. Consider another example of this next table. This **next[j]** be the character position in the pattern which should be checked next after such a mismatch, so that we can slide the pattern (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 pattern all the way past the current text character. Now we shall discuss how to pre-compute this table; fortunately, the calculations are quite simple, and we will see that they require only O(m) steps. Now we represent following the algorithm to calculate the next 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 pattern to the right, so it is performed a total of m times at most. A slightly different way to prove 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 nonnegative. Therefore the operation next[j], which always decreases j, can be performed at most m-1 times [6].

VII.II. *The Pattern Matching Algorithm*

The Knuth-Morris-Pratt matching algorithm is given in pseudo code below as the procedure KMP-MATCHER. KMP-MATCHER 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 match.
if P[j + 1] = T [k ]
 then j ← j + 1       //Next character matches.
 if j = m             //Is all of P matched?
 then print “Pattern occurs with shift” k– m
 j ← next[j]          // Look for the next match.
```

For convenience, let us assume that the input text is present in an array text T[1…n], and that the pattern appears in pattern P[1…m]. We shall also assume that m > 0, i.e., that the pattern is nonempty. Let k and j be integer variables such that text T[k] denotes the current text character and pattern P[j] denotes the corresponding pattern character; thus, the pattern is essentially aligned with positions p + 1 through p + m of the text, where k =p +j [15].

VII.III. Complexity

The KMP algorithm works by turning the patterns given into a machine, and then running the machine. It takes 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 is a linear time string matching algorithm. [6]

VIII. BOYER-MOORE ALGORITHM

A significantly faster string searching method can be developed by scanning the pattern from right to left when trying to match it against the text. The Boyer-Moore algorithm (BM) was developed 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 performs the comparisons from right to left.

VIII.I Bad Character Rule

To convey the idea of the bad character rule, let us suppose that the last (rightmost) character of pattern P is y and the character in text T it aligns with is x, x ≠ y. When 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 position of character x exists in pattern P. This observation is formalized below [16].

For a particular alignment of pattern P against text T, the rightmost (n-i) characters of pattern P match their counterparts in text T, but the next character to the left, P(i), doesn’t matches with its counterpart, 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 point of this shift rule is to shift P by more than one character when possible. In the below example, T(5) = t mismatches with P(3) and R(t) = 1 so P can be shifted right by two positions. After the shift, the comparison of P and T begins again at the right end of P.

Figure 3: 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
P			A	G	A	T	G	A	T								

VIII.II. Extended Bad Shift Rule

When a mismatch 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 introduce another rule called the good suffix rule.

Suppose 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 next comparison to the left. Then find, if it exists, the rightmost copy t’ of t in P such that t0 is not a suffix of P and the character to the left of t’ in P differs from the character to the left of t 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 exist, then shift the left end of P. past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P n places to the right. If an occurrence of P is found, then shift P by the least amount so that a proper prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift 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
										^								
P			q	c	a	b	d	a	B	d	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 by the good suffix rule, so z has a chance of matching x. When the mismatch occurs at position 8 of P and position 10 of T, t = ab and t0 occurs in P starting at position 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	

Now in cases where we have matched the final m characters of pattern P before failing, we clearly wish to shift our attention down string by 1+m. So, L(i) is the largest index j less than n such that $N_j(P) \geq |P[i..n]|$ (which is n - i + 1). L'(i) is the largest index j less than n such that $N_j(P) = |P[i..n]| = (n - i + 1)$.Now The pre-processing stage must also prepare for the case when L'(i) = 0 or when an occurrence of P is found. l'(i) equals the largest $j \leq |P[i..n]|$, which is n-i+1, such that $N_j(P) = j$. So we can say that the required shift will be max (L(i),L'(i)).

VIII.IV. *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
```

Note 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. Rather, the index k is increased to the point where the right end of P would be shifted". Hence, each act 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 pattern does not appear in the text. This was first proved by Knuth, Morris and Pratt [6].

VIII.V. *Algorithm Complexity*

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

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 find a numeric pattern 'P' from a given text 'T'. It first divides the pattern with a predefined prime number 'q' to calculate the modular of the pattern P. Then it tests the first m characters (m=|P|) from text T to compute remainder of m characters from text T. If the remainder 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 repeat the process for next set of characters from 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). So according to this 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: If 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=2345678997977979765343566788867564568909755453434324545475655454
P=667888
q=11
So to find out this pattern from the given text T we will take equal number of characters from text as in pattern and divide the value of these characters with predefined number q and also divide the pattern with the same predefined number q. Now compare their remainders to decide whether to compare the text with pattern or not.
Rem (Text) =234567/11=3
Rem (Pattern) =667888/11=1
As both the remainders are not equal so there is no need to compare text with pattern. Now move on to set of characters of same length next from text and repeat the procedure. The Boyer Moore Algorithm goes as follows:

```
Rabin_Karp_Matcher (T,P,d,q)
{
        n =Length (T)
        m= Length (P)
        $t_0$=0
        p=0
        h=$d^{m-1}$mod q
        for i=1 to m
        {
                p = (d * p + P[i]) mod q
                $t_0$ =(d * $t_0$ + T[i] ) mod q
        }
        for s =0 to n-m
        {
        if $t_s$=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
        t_s+1= (d(t_s-h*T[s+1]) + T[s+1+m] ) mod q
        }
}
```

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

1. Hash P to get h(P).
2. Iterate through all length L substrings of S, hashing those substrings and comparing to h(P).
3. 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 do 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 see if P ever appears in S using the three steps in the method above. Our hash function will be:

$h(k)= (k[0] * 10^4 + k[1] * 10^3 + k[2] * 10^2 + k[3] * 10^1 + k[4] * 10^0) \bmod m$

Or in other words, we will take the length 5 array of integers and concatenate the integers into a 5 digit number, then take the number mod m. h(P) = 50330 mod m, h(S0) = 48503 mod m, and h(S1) = 85033 mod m. Note 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 get 85033. More formally:

$h(S_{i+1}) = [(h(S_i) - (10^5 * \text{first digit of } S_i)) * 10 + \text{next digit after } S_i] \bmod m$

We can imagine a window sliding over all the substrings in S. Calculating 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 interpret the arithmetic easier. To generalize for other base b and other substring length L, our hash function is

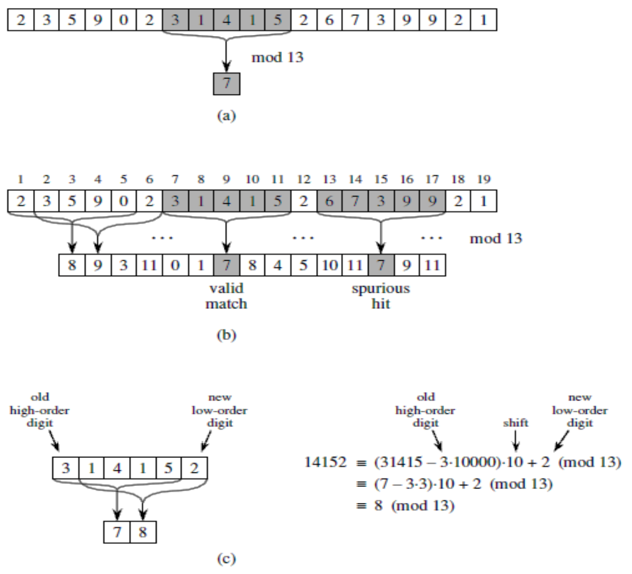
$h(k) = (k[0]b^{L-1} + k[1]b^{L-2} + k[2]b^{L-3} .... k[L - 1]b^0) \bmod m$

And calculating the next hash value can be done by:

$h(S_{i+1}) = b(h(S_i) – b^{L-1}S[i]) + S[i + L] \bmod m$

Following is the example taken from [15]:

Figure 6:



The above figure[15] illustrates (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. (b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming 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 pattern, while the second, beginning at text position 13, is a spurious hit. (c) Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152.

X. Improved idea:

Theory As we can see, spurious hit is an extra burden on algorithm which increases its time complexity when we have to compare the text with pattern and won't be able to get the pattern at that shift. So to avoid 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 number. So, according to this technique along with the calculation of remainder, we will also find out the quotient and if both remainder and quotient of text matches with pattern then it is successful hit otherwise it is an unsuccessful hit or spurious hit and then we can remove the possibility of comparing the spurious hits. That means there is no extra computation of spurious hits if remainder and quotient are same then pattern found else pattern not found.

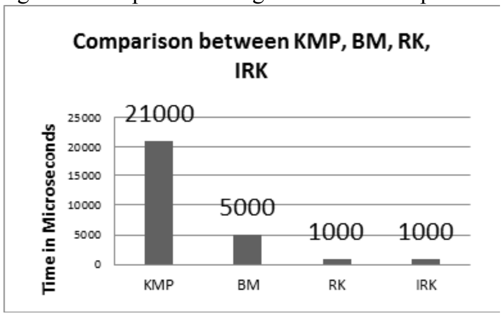
Basically the algorithm is same as the original rabin karp algorithm, but with little modifications, which are shown in bold italic font. The algorithm goes as follows:

```
IRK( T, P, d, q )
n ← length (T ) //text length
m ← length ( P ) //pattern length
h ← d^(m-1) mod q
p ← 0
t_0 ← 0
q_p ← 0  //quotient post hash calculation for pattern
//quotient post hash calculation for portions of text of size m
q_t ← 0
for i ← 1 to m        //Preprocessing
        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
        t_0 ← temp mod q
for s ← 0 to n – m  // Matching
//comparison only if quotient matches, removal of spurious hit
        do if p = ts && q_p = q_t
                then print "Pattern occurs with shift" s
        if s < n – m
//quotient, post hash calculation of next m characters in text.
        temp_t ← ( d * ( t_s – T[ s + 1 ] * h ) + T[ s + m + 1 ] ) /
q
        q_t  ← temp_t /q
//subtracting   LSB,   Shifting   and   adding   MSB   then
t_s+1  ← ( d * ( t_s – T[ s + 1 ] * h ) + T[ s + m + 1 ] )mod q
t_s= t_s+1
```

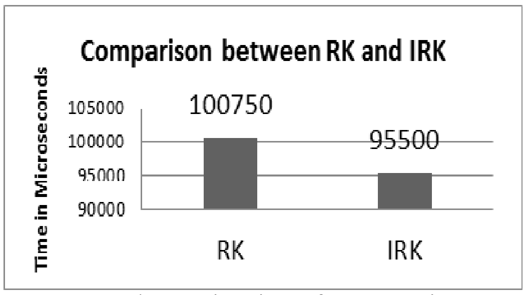
X.I. *Comparison using Graphs:*

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 fixed text file size of 1MB. Y axis represents time in microseconds and X-axis represent the corresponding algorithms.

Figure 7: comparison of algorithms with respect to 1 MB text file

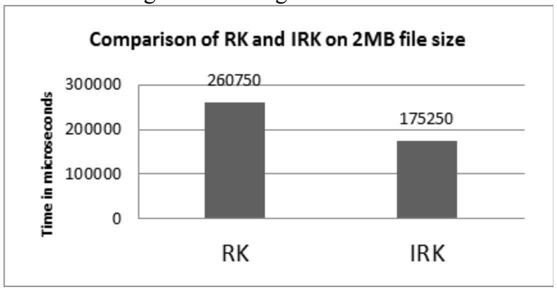


Now we compare only between Rabin Karp and IRK algorithms with the same text file size of 1 MB, in figure 8. Figure 8: Comparison of Rabin Karp and IRK algorithms using file size 1MB.



Rabin Karp scores the running time of 100750 microseconds and IRK adjusted the running 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. Also 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)= $(66 * 256^2 + 67 * 256^1 + 65) \bmod q$
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_t_2 = 334045

A	B	B	C	A	B	C	A

hash(CAB) = 7
q_t3 = 339047 // both matched

A	B	B	C	A	B	C	A

hash(ABC) = 11
q_t_1 = 328984

A	B	B	C	A	B	C	A

hash(BCA) = 0 // hash matched
q_t2 = 334045 // quotient matched

Since the hash =0 and quotient = 334045 both matched. Only 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 much from the original Rabin Karp algorithm, but the in average case complexity can be improved significantly. Due to imposing of constraint of matching the quotient post hashing as well as the hash value of the text portion of size *m* , reduction in comparison has been seen. Which reduces 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. So selecting the right prime number gives this algorithm a satisfiable optimization in terms of worst case time complexity.

XI. Conclusion And Future Scopes

This version of Rabin Karp algorithm can be used with Genetic Algorithm in order to search for a pattern in to huge text files of size >500MB. Implementation using GA can produce an improved version of this algorithm for more sophisticated use and can make the search even faster by using 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 pattern analysis. Further analysis and improvement of this algorithm is welcome from any scholars.

References

1. Richard M. Karp, Michael O. Rabin, Efficient Randomized pattern-matching 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.

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 Gusfield. 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