International Journal of Scientific & Engineering Research, Volume 3, Issue 11, November-2012
ISSN 2229-5518

A Genetic Algorithm Based Pattern Matcher

Sagnik Banerjee, Tamal Chakrabarti, Devadatta Sinha

Abstract— Pattern matching is the method of searching a pattern in a text. There are several existing algorithms which successfully locate the presence of a pattern in a text. In particular, Bioinfomaticians often search for a Deoxy-Ribo Nucleic Acid (DNA) pattern in a very long DNA sequence. These algorithms search the entire text in order to locate the pattern. Genetic algorithms, on the other hand, deal with search procedures that are based on natural selection. Nature selects those individuals that are, in comparison, healthier than others in a certain generation. Even though this method of natural selection relies on probability, the final result of the selection has generally led to better and healthier individuals. In this paper we have presented a scheme, which searches only certain portions of the text, determined by the genetic algorithm, where the probability of finding the pattern is the maximum.

Index Terms— Bio-informatics, DNA, Genetic algorithm, Non-deterministic process, Pattern matching.

1 Introduction

Pattern matching is a mechanism to locate the presence of a sequence of characters (called the pattern) in a much longer sequence of characters (called the text) [3]. In order to locate the pattern in the text, most existing algorithms, such as the Knuth-Morris-Pratt (KMP) [5] and Boyer Moore (BM) [2] algorithms scan the entire text for the presence of the pattern. This causes the run time to increase for very large strings such as the DNA.

DNA is a chemical compound containing four types of nitrogen bases Adenine, Guanine, Cytosine and Thymine. DNA is made up of a certain order of these bases. In a computer we represent each nitrogen base with a single character: A for Adenine, G for Guanine, C for Cytosine and T for Thymine [11]. A DNA sequence is a representation of the genetic code contained within an organism. DNA pattern matching [10] is an identification of a pattern of nucleotides in one or more sections of a given genetic code [7]. Biologists use the pattern matching algorithms to discover evolutionary divergence [6], the origins of disease [9], and ways to apply genetic codes from one organism into another [8].We treat every DNA sequence as a file which consists of a sequence of characters belonging to the set S, where S= {A, G, T, C}. Typically these files can grow to be extremely large, for example size of the file which stores the genome of Polychaos dubium (Amoeboid) is about 670GB. The algorithms which search the entire file [1] for a pattern take longer and longer time, as the file sizes continue to grow. To avoid searching the entire text, our algorithm attempts to break up the text into smaller independent units and then search for the pattern in each of those units. The points at which the text would be broken are decided based on a genetic algorithm.

- *Sagnik Banerjee is currently pursuing master's degree program in Computer Science and Engineering from Jadavpur University, Kolkata, India, PH-919038749129. E-mail: sagnikbanerjee15@gmail.com*
- *Tamal Chakrabarti is currently working as an Assistant Professor in the department of Computer Science & Engineering at Institute of Engineering & Management, Kolkata, India, PH-919836237632. E-mail: tamalc@gmail.com*
- *Devadatta Sinha is currently working as a Professor in the department of Computer Science & Engineering at Calcutta University, Kolkata, India, PH-919830269278. E-mail: devadatta.sinha@gmail.com*

Genetic algorithm (GA) basically uses the method of "Natural Selection" to select the best chromosomes from a generation [4]. Then GA uses genetic operators to like crossover and mutation to create better individuals. Computer algorithms have been designed to simulate these operators. All these algorithms are probabilistic in nature. In our algorithm we have used genetic operators to break up the text into independent units. Then on each unit we have the applied existent pattern matching algorithms Knuth-Morris-Pratt (KMP) and Boyer Moore (BM).

This algorithm basically searches for the presence of a given pattern only in that portion of a text where there is a high chance for it to exist. Since, effectively, the text size is reduced the algorithm can extensively be applied to the DNA pattern matching problem for achieving considerable improvement of the search-time.

2 Related Work

The Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM) algorithms are the most widely used in pattern matchers. We will denote 'n' as the size of the text and 'm' as the size of the pattern (n >> m) in subsequent discussions. Table 1 depicts the same.

TABLE 1
IMPORTANT NOTATIONS-1

Abbreviation	Description
n	Size of text
m	Size of pattern

The KMP algorithm makes use of the observation that when a mismatch occurs during pattern matching, the pattern itself contains enough information to determine where the next match could begin. The algorithm skips re-examination of previously matched characters.

KMP does preprocessing on the pattern. Using the pattern it creates a table which is also called a failure function. It basically indicates the amount by which the pattern should be shifted when a mismatch occurs. The goal of computing the table is to ensure that every character in the text is matched

exactly once. By preprocessing we examine all possible substrings of the pattern and prepare a list of all possible shifts that bypass a maximum of useless characters while not sacrificing any potential matches in doing so. The failure function computes the length of the longest proper prefix and the common longest proper suffix for all possible substrings of the pattern.

The algorithm searches for a pattern in a text from left to right. At any position if there is a mismatch between the characters of the text and of the pattern, then the pattern is shifted by a value given by the failure function.

The complexity of KMP is O(m+n). The preprocessing function runs in O(m) time which basically scans the pattern. Then the algorithm that actually matches the pattern with the text runs in O(n) time. One of the biggest advantages of KMP is that during matching the pattern with the text, it never processes a character in the text more than once.

The BM algorithm also searches for a pattern in a text from left to right. But unlike KMP there are two heuristics instead of one. The BM algorithm initially places the pattern with the text at the starting point of the text. Then matching is done from right to left. In case of a character match the previous character in the alignment is attempted to be matched. If by doing so the front of the pattern is reached then a match is declared. But in the event of a mismatch the pattern is shifted to the right according to the maximum value permitted by a couple of rules, called the bad-character and good-suffix rules. These shift rules are generated during pattern preprocessing.

If the pattern does not appear in the text then the worst case complexity is O(m+n). When the pattern does occur in the text then the worst case complexity is O(mn), the best case complexity is O(n/m).

From the time complexity it is evident that BM will perform better when the size of the pattern is large in the best case, which is a major improvement over KMP. But in the worst case the complexity increases to be quadratic. The main disadvantage of BM is its dependence on the pattern, and that it does not work well with a small alphabet size.

3 GENETIC ALGORITHM

Genetic algorithm is an adaptive search heuristic in the field of Artificial intelligence that imitates the process of natural evolution. In the cells of every living organism there exists a set of chromosomes which is same for every other cell of that individual. A chromosome consists of genes, which are blocks of DNA. During reproduction, the fittest individuals are selected by the process of natural selection. These individuals undergo the method of recombination (or crossover). In this step portions of the healthy chromosomes are randomly exchanged in the hope of achieving healthier individuals. After crossover, the process of mutation takes place. In mutation parts of a chromosome is changed randomly. This is done because during crossover there is a slight chance that a healthy portion from an individual may get destroyed due to recombination.

Genetic algorithms result in better offspring and much evolved species. This characteristic of GA can be used to optimize problems in other domains as well. Genetic Algorithms belong to the group of evolutionary algorithms which are probabilistic in nature. In this paper we use genetic algorithm to determine locations, in texts, where a certain pattern can be found. The challenge lies in modeling the real life problem into strings which can be processed by a computer. Table 2, depicts the notations that will be used in subsequent discussions.

TABLE 2
IMPORTANT NOTATIONS-2

Abbreviation	Description
N	Population size, the number of chromosomes
look_up	Hashed table
GA	Genetic algorithm
A	Area to be scanned

Initially a certain number of strings are considered. These strings are analogous to chromosomes. They form the first generation. A fitness function is designed which accepts a chromosome as an input and returns the fitness value of that chromosome, which is indicative of the health of the chromosome. Then depending on the fitness of the chromosomes a Roulette wheel is designed. On that wheel chromosomes having higher fitness occupy larger area than chromosomes which have lesser fitness. The Roulette wheel is rotated N times where N is the population size. Each time a preset pointer chooses a chromosome. Therefore after each rotation a very healthy individual is chosen. After selection the process of crossing over takes place. In this step parts of a randomly selected pair of chromosomes are exchanged in the expectation of yielding better individuals. Then in the next step some randomly selected locations of randomly selected chromosomes are mutated with the expectation of reviving healthy portions of chromosomes which might have been lost due to crossover. After these processes we get N chromosomes again. These chromosomes form the new generation. The genetic operators can be again applied to this generation in order to get an ever healthier generation.

Genetic algorithms have several advantages over normal optimization procedures.

- Genetic Algorithms do not work with derivatives. Therefore there is no problem if the derivative of the fitness function does not exist at any point.
- Many optimization problems, like Hill Climbing, suffer from the problem of local maxima. If the algorithm starts at a poor location it may stop after reaching a local maximum and not continue to search for the global maxima. In case of GA there is hardly any chance of getting stuck at local optima. Since there are a population of points (chromosomes), instead of a single point, GAs climb many peaks in

parallel, that is, they are multimodal. Thus the probability of finding a false peak is reduced.

- In many cases GAs give multiple solutions instead of just one.

The disadvantage of GA is that, like most Artificial intelligence problems, it cannot guarantee a constant response time. Moreover there is no absolute assurance that a GA will find a global optimum.

4 Pattern Matching Combined With Genetic Algorithm

Genetic Algorithms can be useful in breaking the text into several smaller segments, which are independent, and then apply KMP or BM over the smaller sized segments. The pattern can be anywhere within the text. But successful determination of the location of the pattern is impossible without complete examination of the text. Hence, to some extent, the problem of finding a pattern in a text can be called non-deterministic. Therefore we use GA, which is a tool for handling non-determinism. The GA is used to locate portions in the text where the pattern may be found. Once a match is found the algorithm terminates with success returning the position of the text where the pattern was found. This mechanism eliminates the necessity to search the entire text to locate the pattern. If the pattern is present at the extreme end of the text then both KMP and BM has to search the entire text. But if GA is used to break up the text then only a very small portion needs to be searched. For the searching we have used conventional KMP and BM. We have used three genetic operators in our algorithm, namely selection, crossover and mutation. For illustrations we have used the following text and pattern.

Text

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
h	a	v	e		a		n	i	c	e		d	a	y

Fig. 1: Text

Pattern

1	2	3	4
n	i	c	e

Fig. 2: Pattern

4.1 Chromosomes

A chromosome is an organized structure of DNA and protein found in cells. DNA is made up of several genes. Genes are the bearers of traits. Therefore chromosomes are indicative of the characteristics of the population. In order to apply GA the set of inputs should be coded. We have considered the binary representation of positions in the text as the set of chromosomes. For example the position 10 is encoded as 00000000000000000000000000000000000000000000000000000000000001010. We have used 64 bits to represent each chromosome. For future references to chromosomes we have used four bit representation instead of 64 bit representation.

The initial population of chromosomes is created by randomly selecting positions. It is ensured that the position of the chromosomes never cross the length of the text.

4.2 Population Size

It is the total number of chromosomes in a generation. The population size must be even[1]. Let us consider a population size of four for illustration purpose. Let the chromosomes created in the first generation be 3, 7, 12 and 14. Their representation, as a string, will be 0011, 0111, 1100 and 1110.

4.3 Fitness Function

Fitness function is a function that is used to determine the health of a chromosome. It is a real valued function that takes the chromosome as input and returns a real value which indicates the fitness of the chromosome.

We have considered a look up table for which the data structure we used is a two dimensional array of size 256×256, where all the positions are initialized to zero. The pattern is preprocessed and pairs of consecutive elements from the pattern are chosen. The ASCII values of the consecutive elements form indexes to the look up table. Then that position in the look up table is set to one.

For example, for the pattern "nice", we have four characters n, i, c and e in pattern with ASCII values 110, 105, 99 and 101 respectively.

Then positions look_up[110][105], look_up[105][99] and look_up[99][101] are all set to one. Here, for each position in the look up table, the values of the indices are ASCII values of the characters in the pair.

In our algorithm the fitness function accepts a chromosome, which is indicative of a position in the text. Let us call this position a 'cut'. It selects m-1 elements to the left and m elements to the right of the cut. Let us denote this area of the text by A. Therefore the length of the area A is equal to twice the length (m) of the pattern. It is ensured that the extreme left of area A is minimum zero and the extreme right of area A is maximum n, i.e. Area A lies within the text. The scanning begins from the leftmost index A and continues to the rightmost index of A. It selects a pair of characters and compares them to their corresponding location, which is formed out of the ASCII values of the characters, in the look_up table. If the position in the look_up table is zero then a heuristic value, which was previously set to zero, is incremented once. On the other hand if the position in the look_up table is one then the heuristic value in incremented by two. We increment the heuristic by one even if there is a mismatch to maintain diversity in the population. Finally after the entire area 'A' has been scanned, the heuristic value is returned as the fitness value of the chromosome.

It is evident from the fitness function that a chromosome which exists near the pattern, or across the pattern, will have very high fitness value. This will ensure its selection during the process of application of the genetic operator selection. The fitness function can be mathematically represented as

[1]This is required for crossover

$$F: \mathbb{N} \rightarrow \mathbb{N},$$

Where,

$$F(pos) = \begin{cases} \sum_{k=pos-m+1}^{k=pos+m-1} val(k), if\ (pos-m+1) \geq 1\ and\ (pos+m-1) < n \\ \sum_{k=1}^{k=pos+m-1} val(k), if\ (pos-m+1) < 1\ and\ (pos+m-1) < n \\ \sum_{k=pos-m+1}^{k=n-1} val(k), if\ (pos-m+1) > 1\ and\ (pos+m-1) \geq n \\ \sum_{k=1}^{k=n-1} val(k), if\ (pos-m+1) < 1\ and\ (pos+m-1) \geq n \end{cases} \quad (1)$$

And

$$val(k) = \begin{cases} 1, if\ look\_up[Text[k]][Text[k+1]] = 0 \\ 2, if\ look\_up[Text[k]][Text[k+1]] = 1 \end{cases} \quad (2)$$

Therefore, we can see that F is a discrete function, which is in its domain $\mathbb{N}$, but not differentiable. Hence the scope of application of conventional optimization techniques is restricted due to the non-differentiability of the fitness function.

For example the fitness value of the chromosomes can be calculated as under.

F(3)=val(1)+val(2)+val(3)+val(4)+val(5)+val(6)
=1+1+1+1+1+1=6
F(7)=val(4)+val(5)+val(6)+val(7)+val(8)+val(9)+val(10)+val(11)
+val(12) = 1+1+1+1+2+2+2+1+1=14
F(12)=val(9)+val(10)+ val(11)+ val(12)+ val(13)+ val(14)
=2+2+1+1+1+1=8
F(14)= val(11)+ val(12)+ val(13)+ val(14)
=1+1+1+1=4

4.4 Genetic Operators

An operator, which is used in genetic algorithms, to combine existing solution into others and maintain genetic diversity is called a genetic operator. These may be either unary or binary.

4.4.1 Selection

This operator makes use of the fitness function to decide how healthy a chromosome is. It gives preference to better individuals, allowing them to pass on their genes to the next generation. We have used Roulette wheel selection method here. This method, at first, randomly selects a number between 0 and the total fitness of the current population. After this, the algorithm chooses a chromosome at each iteration, and keeps adding its fitness value. Whenever the cumulative fitness value crosses the random number chosen, that chromosome is selected for further operation. Though this process is randomized, it depends on the fitness of each chromosome. Hence only very healthy chromosomes are chosen for crossover.

For example, for the set of four chromosomes having fitness values as 6, 14, 8 and 4 the Roulette wheel will be

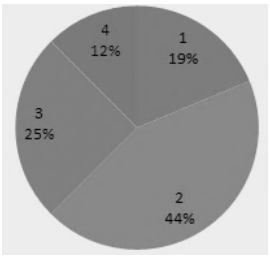

Fig. 3: Roulette wheel

4.4.2 Crossover

This is a binary genetic operator. It operates on two chromosomes. A pair of chromosomes C1 and C2 is chosen randomly from the population, so the size of the population needs to be even. We follow scattered crossover method here. We select a random number R. Firstly, we XOR R with C1.Then we XOR the NOT of R, say Ŕ, with C2.After this we OR these values together to get C3.Again we XOR R with C2 and XOR Ŕ with C1.Then we OR these values together to get C4.Thus, C3 = (C1^R)|(C2^Ŕ) and C4=(C2^R)|(C1^Ŕ). If C3 and C4 are less than the length of the text then these new chromosomes are passed for further operations.

If both the resulting chromosomes are within the bounds of the text then they are passed for the next genetic operation. Otherwise the chromosomes are reset to their previous values. We present an example here with a smaller length chromosome,

If the chromosomes are
C1=0011
C2=1100
And the random number is
R=1001
Then the NOT of this random number is
Ŕ=0110
C3= (0011^1001)|(1100^0110)=1010
C4= (1100^1001)|(0011^0110)=0101

4.4.3 Mutation

Mutation is a unary genetic operator. It operates on one chromosome. A chromosome is chosen randomly and a random bit position of that chromosome is toggled. If the mutated chromosome value is less than length of the text then it is passed on to the next generation.

For example, if the chromosome is C1=1110, and the randomly chosen mutation point is 4, then the resulting chromosome will be C2=0110.

4.5 Generation

A generation consists of N chromosomes. Only for the first generation, the chromosomes are chosen randomly. For all other generations the chromosomes are generated by applying the genetic operators. If the fitness of the current generation is better than the previous generation then the previous generation is replaced by the current generation.

4.6 Pattern Matching

After the completion of the phase of genetic algorithm, we arrive at chromosomes which are nothing but cut points. We sort the population with respect to their cut points and also of their fitness value. For the two cases we use two different lists. If we find that all the fitness values are same then, we choose the first chromosome and the last chromosome and search for the pattern there. Same fitness value for the entire population suggests that the probability of finding the pattern is same around all the cut points. Since the cut points are haphazardly placed in the text, we search the first and last portions of the

text initially, as they have not been probed before. If the fitness values are different, we choose chromosomes depending on their fitness value. Depending on the chromosome chosen, a portion of text is selected which starts from the previous chromosome and extends up to the next chromosome in the list. The pattern is searched for in this portion. During this process we mark the portions of the text already searched. This will later prevent any redundant check on the text. For pattern matching we use the conventional algorithms KMP and BM. For example after application of the genetic operators the chromosomes generated are 2, 3, 9 and 14. From the fitness value chromosome with position nine gets chosen.

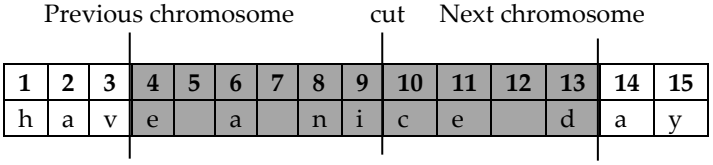

Fig. 4: Effective text area searched

The shaded area, which spans from the previous chromosome to the next chromosome, is searched for the pattern. Next section illustrates the algorithm.

4.7 Algorithm

The *patternMatcher* algorithm computes the look_up table. It also applies genetic algorithm to generate the finest individuals. Then it calls other functions to match portions of the text with the pattern.

```
patternMatcher(Text, Pattern)// finds the existence of Pattern in the Text.
    Set population size and number of generations.
    Preprocess the pattern and generate the look_up table.
    Create the initial population of chromosomes by randomly selecting positions from the text.
    foreach (chromosome ∈ chromosomes)
        fitness = findFitness(chromosome)
    repeat
        selectChromosomes()
        crossover()
        performMutation()
        chooseGeneration()
        generations ← generations – 1
    until (generations = 0)
    arrange chromosomes according to their fitness
    match()
    display result
```

The *findFitness* algorithm examines a portion of the text around the cut. It uses the look_up table to match pairs of characters from this portion. Depending on a match or a mismatch it updates a particular value. After the entire portion is probed the final value of the heuristic is returned.

```
findFitness(pos) // m is the length of the pattern, n is the length of the text
    l←pos – m +1
    if l<0 then
        l←1
    r←pos+ m
    if r> n then
        r← n
    i←l, j←i+ 1, val←0
    while j ≠ r
        if lookup_up[Text[i]][Text[j]] = zero then
            val←val+1
        else
            val←val+2
        i←i + 1
        j←j+1
    return val
```

The *selectChromosomes* algorithm uses roulette wheel method to select the best chromosomes. It uses random numbers to decide which chromosome to select. At every step it adds the fitness of each chromosome with a variable. When the value of that variable crosses the value the chosen random number, the chromosome is selected.

```
selectChromosomes()
    repeat
        Select a random number R between 0 and the total fitness of the generation
        i←1, fitnessSoFar←0
        while i ≤ N
            fitnessSoFar←fitnessSoFar + fitness value of ith chromosome
        if fitnessSoFar>R then
        select chromosome i
until (population size is N)
```

The *crossover* algorithm simulates the process of crossing over. A pair of chromosomes, at a time, is considered for this operation. Depending on the value of a variable, which is chosen randomly, portions of the first chromosome and portions from the other *chromosome* is combined to form new chromosomes.

```
crossover()
        repeat
                Randomly select two chromosomes for crossover C1 and C2
        choose a random number R
        Ŕ ← NOT of R
        C3←(C1^R) | (C2^Ŕ)
        C4←(C2^R) | (C1^Ŕ)
        If C3 < n AND C4 < n then
        C1←C3
        C2←C4
until (N chromosomes have been processed)
```

The *mutation* algorithm chooses a certain number of bits, determined by mutation probability. Then for randomly selected chromosomes it toggles a random bit.

mutation()
k← number of bits to be mutated.
repeat
Select a chromosome randomly.
Select a bit position randomly.
Toggle the value of that bit.
if the mutated chromosome <n then
pass it on to the next generation.
k←k - 1
until (k = 0)

The *chooseGeneration* algorithm finds out if the fitness of the newly formed generation is better than the previous one. If it is so then the new generation becomes the current generation and computation continues with the new generation.

chooseGeneration()
Compute the fitness of the newly created generation.
h← highest value of fitness in the new generation.
if (h> the highest value of fitness in the previous generation) then
newgeneration←current generation

The *match* algorithm determines from the portion of the text to be searched for the existence of the pattern. For this it takes help of the fitness values of the chromosomes in the newest generation.

match() // chromosomes are sorted based on fitness
if (the fitness values of the chromosome in the final generation are all same) then
Choose the first chromosome.
l← 1
r← value of the next chromosome in the sorted list
search for Pattern in Text[l … r] using KMP or BM
Choose the last chromosome.
l← value of previous chromosome in the sorted list-(length_pattern+1)
// ensure that the pattern is found even if it exists across a cut
r ← length of text.
search for Pattern in Text[l … r] using KMP or BM
repeat
Choose a chromosome randomly and mark it.
l ← value of previous chromosome in the sorted list-(length_pattern+1)
r← value of next chromosome in the sorted list.
search for Pattern in Text[l … r] using KMP or BM
until all the chromosomes have been processed
else
repeat
C1 ← the chromosome with the highest fitness
l ← value of previous chromosome of C1 in the sorted list-(length_pattern+1)
r← value of next chromosome of C1 in the sorted list.
if the positions are marked then
Skip this portion to ensure that no portion of the text is searched more than once.
else
search for Pattern in Text[l … r] using KMP or BM
until all the chromosomes have been processed

4.8 Complexity

TABLE 3
IMPORTANT NOTATIONS-3

Abbreviation	Description
N	Population size
G	Generations
m	Length of pattern
n	Length of text
p_m	Probability of mutation
l	Left index of the portion of array chosen
r	Right index of the portion of array chosen

The functions and their asymptotic complexities are given below:
selectChromosomes: O(N2G)
findFitness: O(2mG) = O(mG)
crossover: O(GN/2) = O(NG)
mutation: O(64*pm*NG)=O(NG)
chooseGeneration: O(NmG)
match (for KMP): O(r-l+m)
match (for BM): O((r-l)/m)

5 EXPERIMENTS

We have executed our algorithm, with different sets of text and pattern files, and compared its time with that of the conventional KMP and BM algorithms. We have tested with texts of size 500MB, 600MB, 750MB, 900MB and 1000MB. The pattern files are of size 500B, 750B and 1000B.

We chose population size as 500 and 1000 and generations as three and five. For a particular text file and a pattern file we chose a specific population size and generation. Then we executed the tests 100 times and computed the average.
We have conducted our experiments in the following environment.

- Hardware
  - Processor - Intel® Core™2 Duo CPU T6500 @ 2.10GHz × 2
  - RAM - 3 GB
  - Disk - 320 GB
- Software
  - Operating system – Open SUSE Kernel version 3.1.0-1.2-desktop
  - OS type – 64-bit
  - Compiler used – GCC version 4.6.2 (SUSE Linux)

6 OBSERVATIONS

The results of our experiments are depicted in the graphs below. In the first two graphs we have represented the

performance of the algorithms with changing pattern size, with a fixed text file size of 1GB. In the next two graphs we have represented the performance of the algorithms with change in text size. We have conducted our experiment with 5 different text sizes of 500MB, 600MB, 750MB, 900MB and 1GB. We have used a fixed 1000 bytes file as a pattern for our experiment. The four lines in each of the graphs are for four different algorithms we used for the text and the pattern. The algorithms are conventional KMP, conventional BM, KMP PMGA (KMP with genetic algorithm) and BM PMGA (BM with genetic algorithm). We have used the following notations:

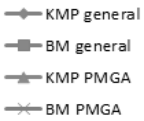

Fig. 5: Reference for graphs

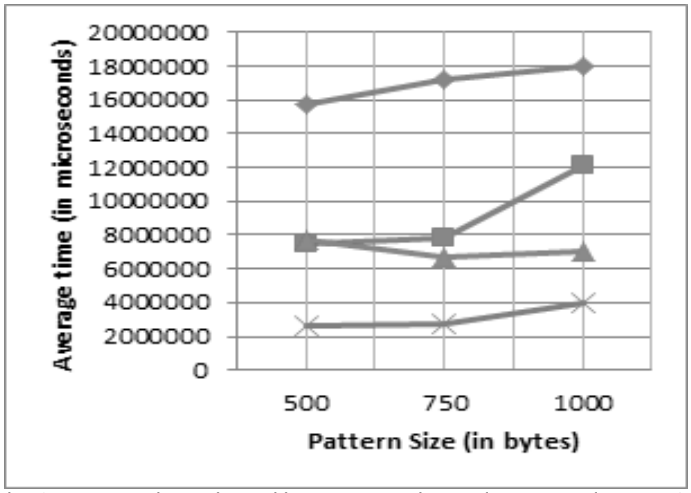

Fig. 6: Average time taken with respect to change in pattern size, N=500 and G=5

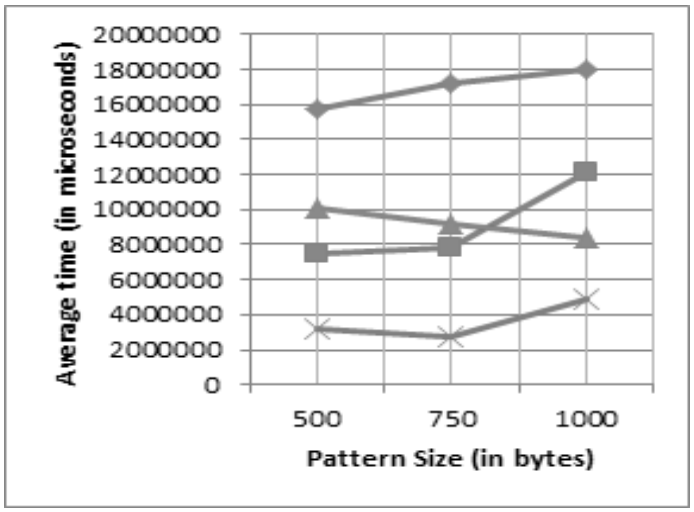

Fig. 7: Average time taken with respect to change in pattern size, N=1000 and G=3

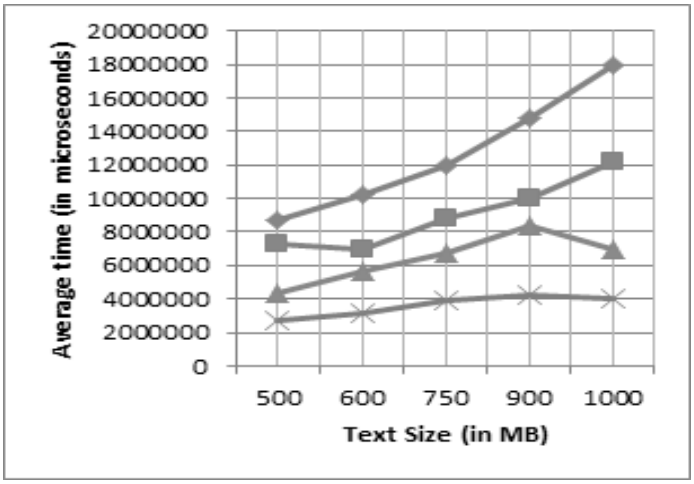

Fig. 8: Average time taken with respect to change in text size,N=500 and G=5

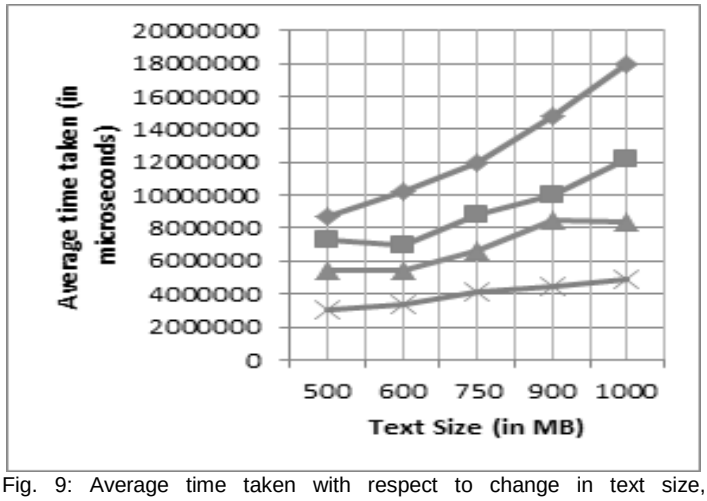

Fig. 9: Average time taken with respect to change in text size, N=1000 and G=3

7 CONCLUSION

From the graphs we can find that KMP when used with genetic algorithm gives much better result than when conventional KMP is applied. This is also true for BM. The most significant advantage of this algorithm is that if the pattern is present, then our algorithm does not need to search the entire text. The search is concentrated on an area where the probability of successfully locating the pattern is highest.

The algorithm is designed in such a way that even if a pattern lies across a cut its detection will be possible, because we are searching from the previous cut to the next cut.

If, however, the pattern is not present in the text, then our algorithm will take more time to give a correct result. This is due to the fact that the entire text has to be searched in order to conclude that the pattern does not exist. But before conducting the search, the GA has to be applied which will take some additional time. Also our algorithm is probabilistic so it will depend on how well pseudo-random number generators work.

REFERENCES

[1] Baeza-Yates. R. A. String Searching Algorithms Revisited. *Lecture Notes in Computer Science*, 382:75–96, 1989
[2] Boyer R. and Moore J.S. A Fast String Searching Algorithm. *Comm. of the ACM*, 20:762–772, 1977.
[3] Colussi. L. Fastest Pattern Matching in Strings. *Journal of Algorithms*, 16:163–189, March 1994.
[4] Goldberg, D.E. (2011): Genetic Algorithms in Search, Optimization and Machine Learning, Pearson.
[5] Knuth D., Morris J. and Pratt V. Fast Pattern Matching in Strings, SIAM Journal of Computer Science, pp323 – 350, 1977.
[6] Lander E.S., Langridge R., and Saccocio D.M. Mapping and Interpreting Biological Information. Communications of the ACM, 34(11):33 – 39, November 1991.
[7] Mount David W., Bioinformatics – Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2001.
[8] Navarro G. and M. Raffinot. Fast and Simple Character Classes and Bounded Gaps Pattern Matching, With Application to Protein Searching. In Annual Conference on Research in Computational Molecular Biology, Montreal, Canada, 2001.
[9] 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
[10] Simone Faro and Thierry Lecroq. An Efficient Matching Algorithm for Encoded DNA Sequences and Binary Strings. Lecture Notes in Computer Science, 2009, Volume 5577/2009, 106-115
[11] Smith-Keary. P. Molecular Genetics. Macmillan Education Ltd, London, 1991.