Finding all Occurrences of a Pattern by a Genetic Algorithm based Divide-and-Conquer Method

Sagnik Banerjee
Department of Computer Science & Engineering, Jadavpur University, 188 Raja S.C. Mullik Road,
Kolkata-700032, West Bengal, India

Tamal Chakrabarti
Department of Computer science & Engineering, Institute of Engineering and Management, Y-12, Block -EP, Sector-V, Salt Lake Electronics Complex, Kolkata-700091, West Bengal, India

Devadatta Sinha
Department of Computer Science & Engineering, Calcutta University, 92 AcharyaPrafulla Chandra Road,
Kolkata-700009, West Bengal, India

ABSTRACT
The method of finding a sequence of characters, called the pattern, in another much longer sequence of characters, called the text, is known as pattern matching. Several pattern-matching algorithms exist, that locate all the positions where a pattern occurs in a text. In this paper we have presented an algorithm which implements a divide and conquer technique, which divides the text in smaller independent sub-texts and then looks for the existence of the pattern in each such sub-text. The said divide and conquer method thus eventually finds all the occurrences of the pattern in the given text. The points of division have been chosen using a genetic algorithm.

General Terms
Pattern Matching, Genetic Algorithm, Divide and conquer

Keywords
Divide and Conquer; Pattern Matching; Bioinformatics; Genetic Algorithm.

1. INTRODUCTION
Many computing applications today requires the task of finding the first, or all of the occurrences of pattern in a text — string searching — to be performed [2]. A string matcher usually aligns the pattern with the beginning of the text and keeps shifting the pattern forward until a match is found or the end of the text is reached, indicating the existence or otherwise of the pattern in the text [4]. Complex methods of string matching [9] today are used in a variety of areas, such as locating DNA sequences (genetics), fingerprint assessment (criminology), soil patterns (geology), retinal blood vessel assessment (medicine), design of assembly lines to improved flow (business), continuous speech profiles [8] and many other fields.

For example, DNA Pattern matching is an important task of the pattern discovery process in Bioinformatics for finding the structural and functional behavior in genes [7], [11]. Molecular biologists often search for the important information from the DNA databases in different directions of different uses, such as detection of disease etc. [10]. With the increasing need for instant information, pattern matching algorithms will continue to play a very important role in the application of Bioinformatics [12].

Most widely used algorithms for pattern matching are Knuth Morris Pratt (KMP) and Boyer Moore (BM). These algorithms are sequential in nature. In order to locate all occurrences of a given pattern in the text these algorithms search the entire length of text. Due to the sequential nature of these algorithms the run time of the operation of pattern matching becomes very high. Our algorithm breaks up the text into independent units.

The division of the entire text into independent units is done based on a genetic algorithm (GA). Genetic algorithm is an adaptive search heuristic in the field of Artificial Intelligence that imitates the process of natural evolution [5]. It works on a philosophy of "natural selection" to pick the best among the candidates from a generation. For GA we have used the most common genetic operators like crossover and mutation to create better individuals.

2. RELATED WORK
The Knuth-Morris-Pratt (KMP) [6] and the Boyer-Moore (BM) [3] algorithms are the two most widely used 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

Abbreviations	Description
n	Size of text
m	Size of pattern

KMP makes use of the observation that when a mismatch occurs, the pattern itself embodies enough information to determine where the next match could begin. KMP, therefore, does pre-processing of the pattern.

Unlike KMP, the BM algorithm makes use of two heuristics, a good character heuristic and a bad character heuristic. The shifting of the pattern in the event of a mismatch is decided based on the value of these heuristics.

The complexity of KMP is O(m+n), whereas the complexity of BM is O(n/m).

We had designed an algorithm [1] to search for the presence of a given pattern only in that portion of a text where there is a high chance for it to exist. The portion of the text, where the existence of the pattern is very probable, was identified by a genetic algorithm. Experimental results established that the algorithm runs faster than the KMP or BM algorithms. But using our algorithm one could only tell whether a given pattern exists in a text or not. In case a text had multiple

occurrences of a pattern, it is sometimes necessary to find out all such occurrences. This paper presents a pattern matcher to do exactly that, by employing a divide and conquer technique combined with the genetic algorithm.

3. ALGORITHM

Initially, using genetic algorithm the location of one occurrence of the pattern in the text is found. The genetic algorithm initializes the population with random chromosomes. We have considered the binary representation of positions in the text as chromosomes. For example the position 15 is 00001111 (we had used 64 bit representation). For each such chromosome their fitness is calculated.

3.1 Fitness function

For the purpose of calculating the fitness of each chromosome a look up table is generated [1]. For every chromosome we chose an area of m-1 elements to the left of the position and m elements to the right of that position. Let this area be 'A'. Then A is scanned sequentially. A pair of characters is chosen from the area 'A'. If that position in the look_up table contains zero then the fitness value is incremented once. If that position contains one then the increment of the fitness value is one greater than the increase in the previous iteration. The equations are given below:

$$F: N \to N$$

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

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

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

3.2 Genetic operators

Selection of the chromosomes is done by spinning the roulette wheel. Thus fitter individuals have a greater chance of being selected. These selected individuals are then crossed over to produce new individuals. We follow the procedure of scattered crossover here. After crossover we go for mutation. In this case we have used the procedure of single-bit mutation. It is a unary operation where the bit at any random position is flipped. These operations are repeated over and over again for a certain number of times to produce fitter individuals.

Thus by applying our genetic algorithm, we can locate one of the occurrences of the pattern in the given text [1]. Let the starting of this area be denoted by left and ending be denoted by right. Then this point of occurrence was used to divide the larger text in two smaller parts. If the pattern was located at position p in the text, then the algorithm would divide the text into two parts, one from left to (p + FK[1] - 1) and another one from (p + length_of_pattern - FK[1] + 1) to right. Here FK is the failure function generated during pattern pre-processing by KMP. This has been illustrated in the following pseudo code.

```
Algorithm patternMatcher(left, right)
If (left < right)
        p = search (left, right)
        // use GA to search for the pattern
        // between left and right indices
        patternMatcher (left, p + FK[1] - 1)
        patternMatcher (p + length_of_pattern-FK[1] + 1 ,right)
end if
end function
```

The function *search()* applies GA to locate one occurrence of the pattern in the text within the positions *left* and *right*. Following the call to function *search()* there are calls to the function *patternMatcher()* with different parameters. These are basically recursive calls which search the portion of the text before position *p* and after position *p*. The following example illustrates the procedure. We consider the following text and pattern.

Initially the algorithm *patternMatcher* starts with *left=0* and *right=25*. Then let us assume that the search procedure returns 9. If the portion of the text under consideration has more than a single occurrence of the pattern then the search procedure can return the starting position of any such occurrence. Then the algorithm divides the entire text into two parts, one part from 0 to 9 and the other one from 10 to 25. Then it calls the search procedure recursively on these two parts. The first portion returns 1 and the next portion returns 24. Now each of these individual portions will get further subdivided. The recursive procedure will end when the value of *left* is no longer less than that of *right* or if the *search* procedure returns 0, i.e. the portion of text under consideration does not have the pattern.

The following example depicts detection of a smaller gene sequence (pattern) within a larger gene sequence (text). We assume that the pattern is a disease causing gene sequence. We will apply our algorithm to locate all occurrences of such a pattern within a genetic text.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	G	C	T	A	A	T	C	G	T	G	C	T	A	T	C	A	T	G	C	A	C	T	A	A	C

Figure 1: Text

0	1	2
C	T	A

Figure 2: Pattern

0	1	2
-1	0	0

Figure 3: FK

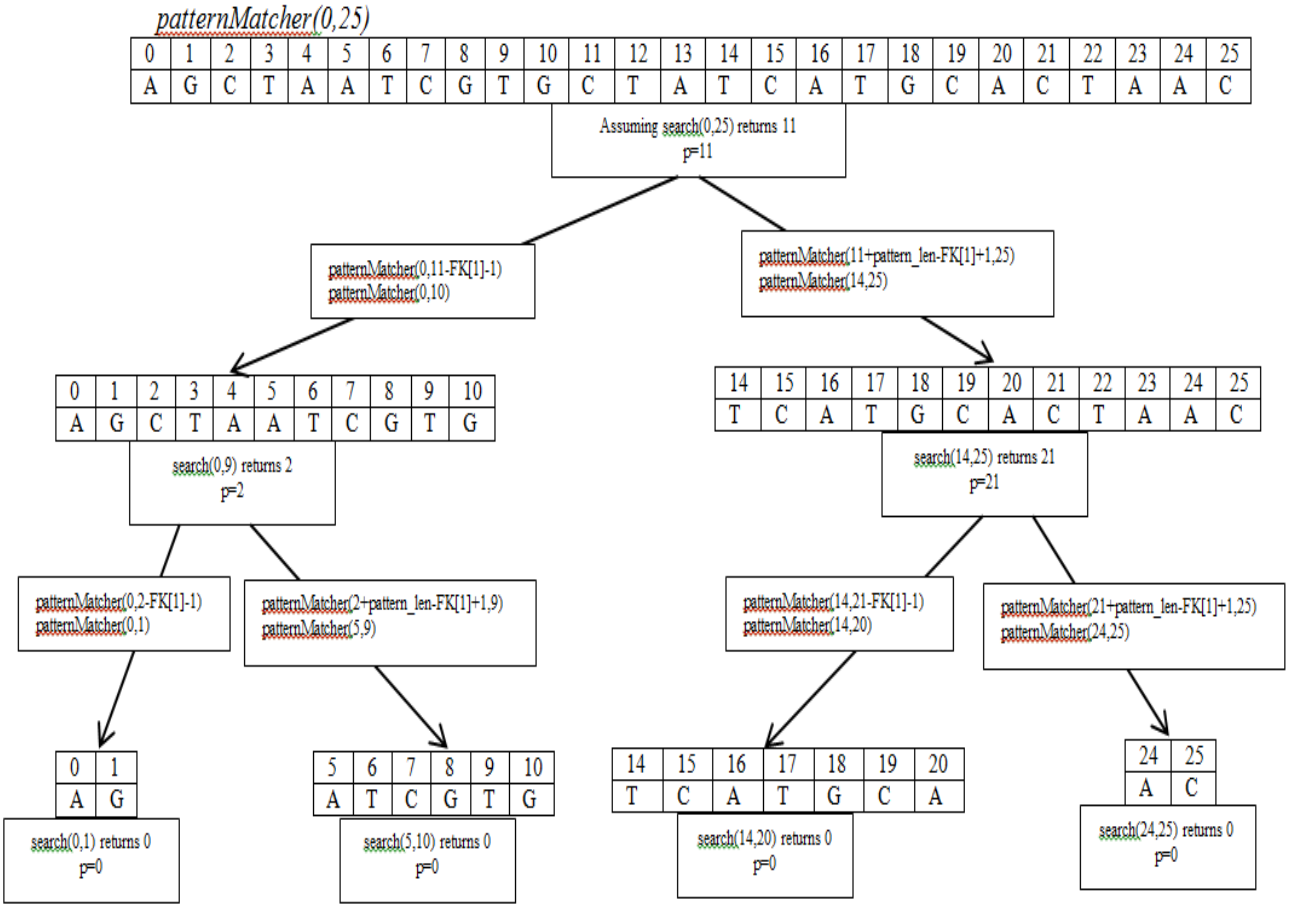

Figure 4: Recursion Tree

The above procedure works exactly like the previous illustration. Therefore we can detect presence and the locations of disease causing genes in a given larger gene sequence. The extent of the disease can also be estimated from the number of times such disease causing gene occurs in the sequence. Since the locations of the genes can be pin-pointed exactly, genetic engineering techniques can be applied to cure such diseases.

4. EXPERIMENTS

We have executed our algorithm, with different sets of text and pattern files. We have tested with texts of size 750MB, 900MB, 1024MB, 1280MB and 1536MB. The pattern file was of size 100B. With this set up we have later demonstrated a comparison of the behavior of our algorithm, when it is used in conjunction with the KMP algorithm and the BM algorithm. This experiment shows how change in text size affects run time.

In another experiment we demonstrate the change in run time with respect to change in pattern size. For this case we chose a text file of length 750MB. The pattern files were of sizes 100B, 125B, 150B, 200B, 250B.

We have conducted our experiments in the following environment.

- Hardware
  - Processor - Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8
  - RAM – 8GB
  - Disk 1000 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)

5. OBSERVATIONS

The results of our experiments have been depicted below. The first graph shows the change of average execution time of the program with respect to change in text size. The second graph displays the change in average execution time with respect to change in pattern size.

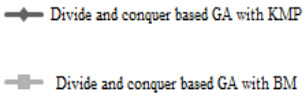

Figure 5: References

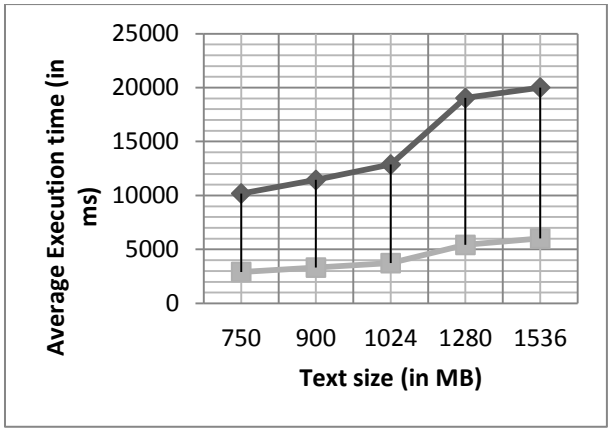

Figure 6: Average execution time vs text size

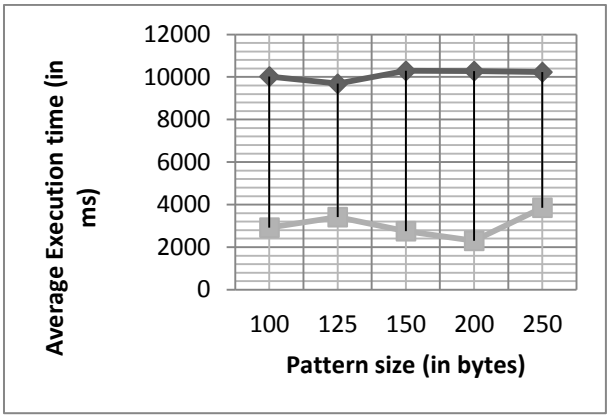

Figure 7: Average Execution time vs. Pattern size

6. CONCLUSION

From the graphs we can conclude that our method employed with the BM algorithm performs better than the corresponding KMP version when divide and conquer is used. The sequential nature of conventional KMP and conventional BM prevents us to use processors that are lying idle in the system. The algorithm suggested in this paper overcomes this problem by dividing the text recursively into smaller texts which are independent of each other. These independent units can be parallely executed to achieve better performance. Moreover one can use a pool of threads to simulate the parallelism. The only problem with our algorithm is that it will take more time to complete execution if the pattern does not occur in the text. This is due to the overhead of computations used in GA.

7. REFERENCES

[1] Banerjee Sagnik, Chakrabarti Tamal and Sinha Devadatta (2012), A Genetic Algorithm Based Pattern Matcher, International Journal of Scientific & Engineering Research, Volume 3, Issue 11

[2] Baeza-Yates. R. A. String Searching Algorithms Revisited. Lecture Notes in Computer Science, 382:75–96, 1989.

[3] Boyer R. and Moore J.S. A Fast String Searching Algorithm. Comm. of the ACM, 20:762–772, 1977.

[4] Colussi. L. Fastest Pattern Matching in Strings. Journal of Algorithms, 16:163–189, March 1994.

[5] Goldberg, D.E. (2011): Genetic Algorithms in Search, Optimization and Machine Learning, Pearson.

[6] Knuth D., Morris J. and Pratt V. Fast Pattern Matching in Strings, SIAM Journal of Computer Science, pp323 – 350, 1977

[8] Mount David W., Bioinformatics – Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2001.

[9] 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

[10] 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.

[11] 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.

[12] Smith-Keary. P. Molecular Genetics. Macmillan Education Ltd, London, 1991.