Journal of Computational Intelligence in Bioinformatics
ISSN: 0973-385X Volume 2 Number 2 (2009) pp. 147–158

http://www.ripublication.com/jcib.htm

A Fast Exact Pattern Matching Algorithm for Biological Sequences

Sanchita Paul[1]*, Mangesh K. Rajak[2] and Gadadhar Sahoo[3]

[1,2]Department of Computer Science and Engineering, Birla Institute of Technology, Mesra, Ranchi, Jharkhand, 835215, India
**Corresponding Author E-mail: sanchita07@gmail.com*
[3]Department of Information Technology, Birla Institute of Technology, Mesra, Ranchi, Jharkhand, 835215, India,
E-mail: gsahoo@bitmesra.ac.in

Abstract

Pattern matching is a pivotal theme in computer research because of its relevance to various applications such as web search engines, computational biology, virus scan software, network security and text processing. Pattern matching emerges as a power full tool in locating nucleotides or amino acid in the biological sequence databases. Presently, several pattern-matching algorithms are available; however the efficiency of the various algorithms depends on faster and exact identification of the pattern in the text. In this paper we have proposed a new exact pattern-matching algorithm for biological sequences. The experimental results show that the new algorithm is faster than other algorithms being compared here for small alphabet set and long patterns. The proposed algorithm is quite applicable for pattern matching in biological sequences.

Keywords: Pattern Matching, Exact pattern matching algorithm.

Introduction

Pattern matching is one of the most important research subjects that have been studied in computer science. Over the years, pattern-matching algorithms have been extensively applied in various computer applications, for example, in retrieval of information, information security, and searching nucleotide or amino acid sequence patterns in biological sequence databases.

Pattern matching problem can be defined as finding one or more usually all the occurrence of a given pattern ($P=p_0p_1\ldots p_{m-1}$) of length m in a text ($T=t_0t_1\ldots t_{n-1}$) of length n, which are built over a finite alphabet set Σ of size σ. All pattern-matching

algorithms scan the text with the help of a window, which is equal to the length of the pattern. The first process is to align the left ends of the window and then compare the corresponding characters of the window and the pattern. After a whole matches or a mismatches of the pattern, the text window is shifted in the forward direction until the right end of the window reaches the end of the text. The algorithms vary in the order in which character comparisons are made and the distance by which the window is shifted on the text after each attempt. The efficiency of an algorithm lies in two phases: the preprocessing phase and the searching phase. The characters in the pattern are preprocessed in the preprocessing phase and this information is very much used in the searching phase in order to reduce the total number of character comparisons, which in turn minimizes the overall execution time.

In this paper, a new algorithm is presented. It turns out that the proposed algorithm, achieves good results especially in the case of a small alphabet set and long patterns. The rest of the paper is organized as follows the review of several efficient algorithms in practice, the proposed algorithm BMEPM (Boundary and Middle Element Pattern Matching Algorithm) in detail, a comparison study of the proposed method has been made with some of the faster existing algorithms followed by conclusion.

Previous Work

Generally, pattern-matching algorithms scan the text T with the help of the sliding window of which the size is equal to m. After a whole match of the pattern or a mismatch, the window is shifted along the T according to the heuristics of each algorithm. Pattern matching algorithms can be categorized as single and multiple based on their functionalities. Here we are concerned with the former.

The simplest method of exact matching is the Brute Force algorithm. The premise is simple: the first letter of pattern P is lined up with the first letter of text T, and the letters of the aligned region are compared until all of P is found to match the corresponding T region or a mismatched letter is found, in which case P is shifted one letter to the right and the process is repeated.

The algorithm developed by **Knuth et al.** [1], seeks to improve the length of the pattern shift by utilizing information already gathered from searching a string. To accomplish this goal, the algorithm preprocesses the pattern and creates a finite state machine (or automaton). Typically, the information is portrayed in tabular form with "KNP Next" values assigned to each character in the pattern based on the number of spaces the machine moves the pattern if a mismatch is found. The algorithm then uses the defined finite state machine to process the string.

Boyer-Moore (BM) algorithm [2], is one of the most well-known and efficient pattern matching algorithms. BM utilizes two heuristics, bad character and good suffix, to reduce the number of comparisons. The maximum shift computed by the two heuristics is considered after each attempt during the searching phase.

Sunday designed an algorithm namely **Quick Search (QS) algorithm** [3], which scans the characters of the window in any order and computes its shifts with the

occurrence shift of the character of the text T immediately after the right end of the window.

Horspool algorithm [4], uses a shift value by finding the bad character shift for the right most character of the window. This algorithm compares the rightmost character in the pattern first and then compares the leftmost character, and subsequently all other character from the second position to the (m-1) th position are compared.

In algorithm defined by **Raita** [5], the order of comparison is modified to attain maximum efficiency. Here the rightmost character of the pattern and that of the window are compared, and on an exact match the leftmost character of the pattern and the window are compared. If they match, it compares the middle character of both the pattern and the text window and then the characters from the second to the last but one position of the pattern and the window are compared.

Smith Algorithm [6], is derived from Horspool and Quick Search algorithms. It uses their bad character shift functions to compute shift values.

Colussi Algorithm [7], is an enhancement of Knuth-Morris-Pratt algorithm. In this case the pattern position is divided into two disjoint subsets and is scanned from left to right whereas the other from right to left.

Method

Generally, for the better performance, one needs to implement efficient way of preprocessing the pattern and text to get a better shift value. Secondly, good methodology should be used in the searching phase. Keeping this in mind we proposed here an algorithm that deals with the following two phases:

(1) The preprocessing phase.
(2) The searching phase.

The preprocessing phase: The preprocessing of our algorithm is to construct the Text Element Position (TEP) table and Pattern Element Position (PEP) table of size σ in sorted order where σ is the size of the alphabet set. In these two tables: we stored the position of occurrences of the each element of the alphabet set in the respective text and pattern. Now from PEP-table we can know the first, middle and the last elements and the corresponding size of the pattern.

The searching phase: The preprocessing phase goes hand in hand with the searching phase to improve overall efficiency of the algorithm by calculating larger shift value. The searching phase is accomplished with the following two stages:

Stage 1: For searching a pattern, using the TEP-table which already have been constructed in sorted order during pre-processing phase, we can know the positions where the 1st element of pattern has occurred in the text using binary search. Now we take these positions one at a time, say it is ‘i’ and add the length of the pattern (l) that

is we get the new position (i+l). Now at position (i+l-1) we will find whether the last element of the pattern has occurred or not. If yes, then we compare the middle element of the pattern window with the middle element of the text window, now if they found as same then we align the pattern with the text and the window will be positioned starting at (i) to (i+l-1) and move to stage 2. Otherwise we discard the position and take the next position and repeat the process.

Stage 2: After alignment sequential comparisons is done from the second element to the last but one element until a complete match or mismatches occur. If the entire characters match, then the corresponding position of the window on the text is displayed and for next shift returns back to stage 1. And in case of a mismatch the algorithm directly returns back to the stage 1. Both the above two stages of the searching phase are repeated until the window is positioned beyond the position (n-m+1). The proposed algorithm has been depicted by a flowchart in Fig. 1.

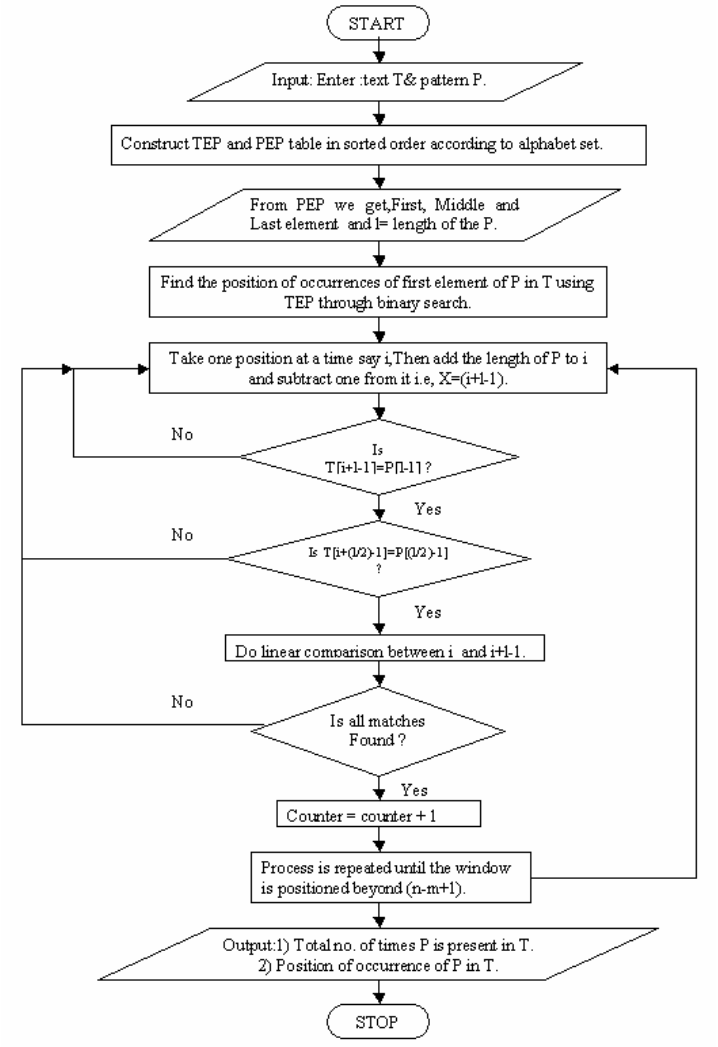


Figure1: Flowchart of the Algorithm.

Working Example

Preprocessing Phase

TEPT STRUCTURE:

Text: A T T T C A G T C A A T C A G T A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	T	T	T	C	A	G	T	C	C	A	T	C	A	G	G	A

TEPT Table->

Alphabet	**A**	**C**	**G**	**T**
Position	**0, 5, 9, 10, 13, 16.**	**4, 8, 12.**	**6, 14, 15.**	**1, 2, 3, 7, 11, 15.**

Table1: Text Elelment Position Table.

PEPT STRUCTURE:

Pattern: T C A G T C

0	1	2	3	4	5
T	C	A	G	T	C

PEPT Table->

Alphabet	**A**	**C**	**G**	**T**
Position	**2**	**1,5**	**3**	**0,4**

Table 2: Pattern Element Position tables.

1st element = T, Middle element = G, Last element= C, Size of the pattern= 6

Searching Phase

Searching Phase

Attempt 1:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	T	T	T	C	A	G	T	C	C	A	T	C	A	G	G	A

0	1	2	3	4	5
T	C	A	G	T	C

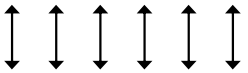
Attempt 2:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	T	T	T	C	A	G	T	C	C	A	T	C	A	G	G	A

0	1	2	3	4	5
T	C	A	G	T	C

Attempt 3:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	T	T	T	C	A	G	T	C	C	A	T	C	A	G	G	A



0	1	2	3	4	5
T	C	A	G	T	C

Attempt 4:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	T	T	T	C	A	G	T	C	C	A	T	C	A	G	G	A

0	1	2	3	4	5
T	C	A	G	T	C

Attempt 5:

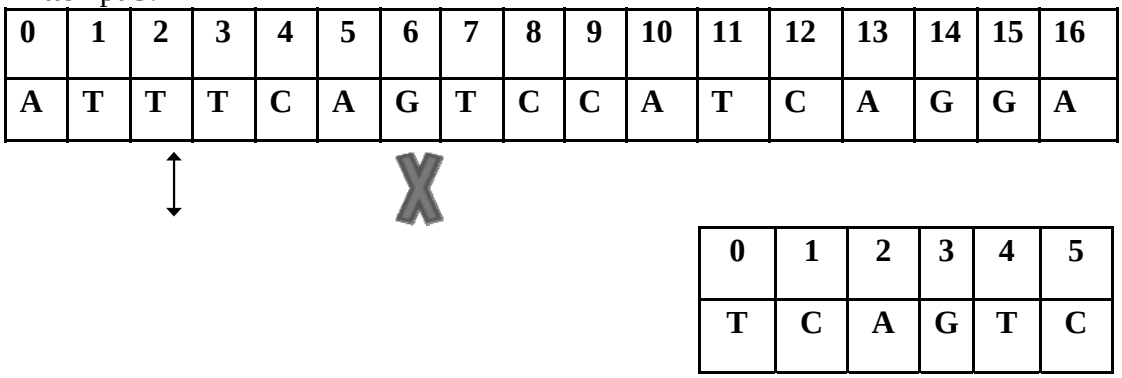


Figure 2: BMEPM Algorithm

Analysis of the proposed algorithm

In the preprocessing phase, we generate the tables TEP-table and PEP-table at the same time. Therefore, its time complexity and space complexity can respectively be defined as O (n+m) and O (n* σ+m*σ) where n is length of text, m is length of pattern and σ is size of alphabet set. In the searching phase, our method is to scan the sliding window from left to right. When a mismatch occurs in Pj-1, we look up TEP-table to decide our shift by employing the binary search method to find the next occurrences of the 1st element of pattern in the text. Therefore the time complexity in this phase is O(r*m) where, r is the number of times 1st element has occurred in the text and m is the length of the pattern.

Comparison Study and Experiments

In this paper, we will show the experiments of our approach and some other existing algorithms and based on these experiments our algorithm has been compared with the other algorithms. The experiments run on a Core 2 Duo with 1.73 GHz of RAM 512 MB under Linux. We compared the following algorithms.

- Brute Force
- KMP algorithm
- Boyer and Moore algorithm
- Horspool algorithm
- Quick Search algorithm
- Raita algorithm
- Smith algorithm
- Colussi algorithm
- Our algorithm (BMEPM)

For our experiment, we have considered a DNA sequence with size 1024 and used it as the text string T. We repeatedly generated a substring of seven different lengths randomly and used it as the pattern string P. We have used seven different lengths of

P and tested our algorithm along with other algorithms.

We have recorded the average number of character comparisons in the searching phase for all the algorithms and the result is shown in the Tables3-9. It can be seen that our algorithm needs much smaller number of character comparisons than that needed by the other algorithms being compared.

Here, we show the experiments with different size of patterns.

Experiment 1

Table 3: The average number of attempts and comparisons performed when pattern length = 4.

For Text Length (n = 1024) and Pattern Length (m = 4)

Algorithms	Character Comparison	Attempts
Brute Force	1376	1021
Boyer Moore	510	381
KMP	1100	829
Quick Search	504	368
Horspool	508	381
Raita	514	381
Smith	426	281
Colussi	871	659
BMEPM	**307**	**254**

Experiment 2

Table 4: The average number of attempts and comparisons performed when pattern length=5.

For Pattern Length (m = 5)

Algorithms	Character Comparison	Attempts
Brute Force	1388	1020
Boyer Moore	382	270
KMP	1085	809
Quick Search	381	263
Horspool	457	322
Raita	310	195
Smith	345	220
Colussi	715	538
BMEPM	**306**	**231**

Experiment 3

Table 5: The average number of attempts and comparisons performed when pattern length =8.

For Pattern Length (m = 8)

Algorithms	Character Comparison	Attempts
Brute Force	1409	1017
Boyer Moore	429	296
KMP	1020	760
Quick Search	492	364
Horspool	469	343
Raita	475	343
Smith	352	259
Colussi	971	733
BMEPM	**341**	**254**

Experiment 4

Table 6: The average number of attempts and comparisons performed when pattern length =12

For Pattern Length (m = 12)

Algorithm	Character Comparison	Attempts
Brute Force	1390	1013
Boyer Moore	368	253
KMP	1064	769
Quick Search	386	296
Horspool	412	316
Raita	413	316
Smith	309	225
Colussi	1014	773
BMEPM	**309**	**242**

Experiment 5

Table 7: The average number of attempts and comparisons performed when pattern length =16.

For Pattern Length (m = 16)

Algorithm	Character Comparison	Attempts
Brute Force	1349	1009
Boyer Moore	361	235
KMP	1062	785
Quick Search	488	368
Horspool	433	306
Raita	428	306
Smith	338	242
Colussi	952	693
BMEPM	**338**	**244**

Experiment 6

Table 8: The average number of attempts and comparisons performed when pattern length =32.

For Pattern Length (m = 32)

Algorithm	Character Comparison	Attempts
Brute Force	1376	993
Boyer Moore	358	260
KMP	1003	710
Quick Search	526	393
Horspool	490	366
Raita	489	363
Smith	380	282
Colussi	980	722
BMEPM	**319**	**237**

Experiment 7

Table 9: The average number of attempts and comparisons performed when pattern length =99.

For Pattern Length (m = 99)

Algorithm	Character Comparison	Attempts
Brute Force	1332	926
Boyer Moore	372	188
KMP	1123	754
Quick Search	477	276
Horspool	546	338
Raita	543	338
Smith	413	236
Colussi	927	620
BMEPM	**353**	**228**

Discussion & Conclusion

In this paper, we presented a pattern matching algorithm which is based on the boundary and the middle element of the pattern. Experiments show that the proposed algorithm requires much smaller number of character comparisons than that required by the other algorithms being compared. So our algorithm has better performance. Therefore, it is feasible that this method can effectively be used in applications related to exact pattern matching in biological sequence databases.

References

[1] Knuth, D. E., Morris, J. H. and Pratt, V. R, (1977), Fast pattern matching in strings, SIAM Journal on Computing, Vol. 6, No. 1, pp. 323-350.

[2] R. S. Boyer, J. S. Moore, (1977), “A fast string searching algorithm”, Communications of ACM, 20(10): 762-772.

[3] D. M. Sunday, (1990), “A very fast substring search algorithm”, Communications of the ACM, 33(8):132-142.

[4] R. N. Horspool, (1980), “Practical fast searching in strings”, Software - Practice & Experience, 10(6):501-506.

[5] RAITA T., 1992, Tuning the Boyer-Moore-Horspool string searching algorithm, Software - Practice & Experience, 22(10):879-884

[6] Smith, P.D., (1991), Experiments with a very fast substring search algorithm, Software - Practice & Experience, Vol. 21, No. 10, pp. 1065-1074.

[7] Colussi, L, (1991), Correctness and efficiency of the pattern matching algorithms, Information and Computation, Vol. 95, No. 2, pp. 225-251.

[8] C. Charras, T. Lecroq, Handbook of exact string matching algorithms, http://wwwigm. univ-lv.fr/~lecroq/string/

Author Biography

Sanchita Paul received the B.E degree from the Burdwan University in 2004 and received the M.E degree from the Birla Institute of Technology, Mesra, India, in 2006. She is currently working as a Lecturer in Birla Institute of Technology, Mesra, Department of Computer Science and Engineering, India. Her research interests are Parallel Computing, DNA Computing and Bioinformatics.

Mangesh Kumar Rajak received the B-Tech., degree in Information Technology from the West Bengal University of Technology, in 2007, and currently pursuing M.E degree in Software Engineering from Birla Institute of Technology, Mesra, India.

Gadadhar Sahoo received MSc in Mathematics from Utkal University, Bhubaneswar in 1980 and PhD in Computational Mathematics from Indian Institute of Technology Kharagpur, India in 1987. He is now with the Department of Information Technology, Birla Institute of Technology (Deemed University), Mesra, Ranchi, INDIA in the capacity of Professor and Head. His research interests are Parallel Computing, DNA Computing and Distributed Systems.