An Enhanced Version of Pattern Matching Algorithm using Bitwise XOR Operation

K. P. Ambika
Dept. of Biotechnology
MS University, Tirunelveli, India.

U. Ramesh
Dept. of Molecular Biology
School of Biological Sciences, MK University, Madurai, India.

K. Saravanan
Dept. of Zoology
Nehru Memorial College
Puthanampatti, Trichy, India.

J. Hencil Peter
Playfast Technologies,
UBI Techpark, Singapore.

ABSTRACT
In this study, a new algorithm for the traditional pattern matching problem has been proposed. This algorithm is a modified version of KMP algorithm and using bitwise XOR operation to process two characters (or bytes) in parallel, to speed up the pattern matching process. An additional loop to avoid the undesirable comparison(s) also been introduced and let the algorithm to initiate, and continue only the essential comparisons from the required location. As the new algorithm uses the principle of Finite automata which is used by KMP algorithm and Bitwise XOR operation to speed up the character match, it shows some reasonable performance improvement. Also this new algorithm is easy to implement as it doesn't require any additional/complex data structure(s) and suitable for DNA sequence search.

Keywords
KMP Algorithm, Pattern Matching. Exact Pattern Matching.

1. INTRODUCTION
A text string T of length n and a pattern string P of length m, with a finite character set $\sum$ with size is equal to σ, pattern matching problem is to find all the occurrences of the given pattern string P in a text string T. There are many applications of pattern matching problems exist in computer science subject, namely text editor, web search engine, image analysis, speech recognition, DNA sequence search in bio-informatics, etc.

In general, pattern matching algorithms can be classified into two categories:

(i) Exact pattern matching algorithm(s).
(ii) Approximate pattern matching algorithm(s).

As the name implies, exact pattern matching algorithm will look for the same sequence of pattern string in the text string. The Brute force algorithm, Knuth-Morris-Pratt Algorithm [3] and Boyer-Moore algorithm [4] are the traditional exact pattern matching algorithms.

Similarly there are many solutions are proposed for the approximate pattern matching problems. For instance, Landau and vishkin proposed two solutions [11] and [12], which are using suffix tree and lowest common ancestor algorithm. Also another algorithm proposed by Galil and Giancarlo [10] is to find the pattern match with k ($k \leq m \leq n$) mismatches.

Automata also play an important role in string matching algorithms. The idea behind automata based solution for pattern search is, prepossess the pattern string before initiate the actual search. So if the P is the sub word of T, compute the shift location to skip the matched (sub word) characters comparison and reinitiate the search from the unmatched string index. Automata based solutions [6], [7], [8], [9] have been applied on most of the algorithms to speed up the performance of pattern search.

This article is organized as follows. In section 2, pattern matching algorithms have been discussed, and followed by section 2, in section 3; newly proposed solution has been explained with the help of sample DNA sequence. Upon explaining the proposed algorithm, Section 4 and 5 address the experimental study and further customization respectively. Eventually section 6 concludes with the advantages of proposed algorithm and further enhancements.

2. RELATED WORK
The exact pattern matching algorithms are used in many practical applications to solve simple as well as complex problems. Also most of the pattern matching algorithm will have two phases, namely pre-processing phase and searching phases. Usually pre-processing phase will pre-process the pattern string P in order to speed up the search and Search phase will use the pre-processed data to locate the pattern string P in text string T efficiently. The simple pattern matching algorithm is Brute force algorithm which doesn't have the pre-processing phase but it checks all the positions in the text string T (between 0 and n-m) to find whether an occurrence of pattern string start with or not. If there is any mismatch occurs during the pattern search, it shifts the pattern by one position towards right and continues the search. So the time complexity of this algorithm is O (mn).

Knuth-Morris-Pratt algorithm [3] performs the comparisons from left to right order but it shifts the pattern very intelligently than the brute force algorithm. This algorithm has two phases, KMP algorithm's Failure function (also it is referred as KMPNext function) will pre-process the pattern to find the prefixes of the pattern with in the pattern itself. i.e it computes the size of largest prefix of pattern string P[0..j-1] which is also the suffix of P[1..j-1] where j is the current mismatch position of pattern string P. So this information will be used in the search phase to shift the pattern elegantly when mismatch occurs. This algorithm requires O (m) time complexity during the pre-processing phase and searching phase requires O (n) time complexity. Hence the overall time complexity of this algorithm is O (n + m).

Boyer Moore[4] is another efficient algorithm when the size of alphabets (σ) are large. This algorithm scans the pattern characters from right to left and compare with the text string. If there is any mismatch occurs, it uses two functions namely, good-suffix shift and bad-character shift, to shift the current

window towards right. The tables required for these two functions would be constructed during the pre-processing phase and used in the searching phase. The worst case time complexity of this algorithm is O (nm).

Karp-Rabin [5] algorithm uses hashing function for the pattern match. Instead of checking each character in text string T for the pattern match, it checks the contents of the window (length must be equal to the given pattern string) "looks like" the pattern string. This algorithm takes O(m) time during the pre-processing and takes O(nm) in searching phase.

The proposed algorithm uses the Bitwise XOR operation to improve the performance of the pattern match process. Initially bitwise techniques have been applied in Shift-Or [1] algorithm. Though this algorithm is an efficient algorithm, it can process the pattern only if the pattern length is no longer than the memory-word size of the system. This algorithm does take $O(m + \sigma)$ time complexity during the pre-processing phase and takes O(n) time complexity during the search phase.

Navarro and Raffinot [14] used nondeterministic backwards directed acyclic word graph in a bit-parallel approach and proposed BNDM algorithm. As this algorithm uses the shift mechanism in a negligent manner, the overall performance of the algorithm is good.

Peltola and Tarhio [15], and Holub and Durian [13] simplified the inner while loop of BNDM algorithm and improved its initial performance. Eventually they proposed good performance algorithms namely SBNDM and SBNDM2 respectively which are based on BNDM.

Pattern matching problems are further modified to support DNA sequence search which are represented using encoded two bits. For instance, Fed algorithm [19] combines a multi-pattern version of the Quick-Search algorithm [20] and a simplified version of the Commentz-Walter algorithm [17]. Simone and Tierry[18] proposed an efficient algorithm for exact pattern matching in encoded DNA sequences and on binary strings. This algorithm combines a multi-pattern version of the Bndm algorithm [16] and a simplified version of the Commentz-Walter algorithm [17].

3. PROPOSED SOLUTON

A new algorithm has been proposed which is based on the popular Knuth-Morris-Pratt (KMP) pattern matching algorithm [3]. The new algorithm initially read the text string as well as pattern string from the file and initializes the in-memory buffers T and P of text string and pattern string respectively. These buffers can be accessed using its index as well as base address (and offset). There are few important changes have been made in this new algorithm, especially to improve the performance:

(i) Bitwise XOR operation for comparing two characters with single operational cost has been used in this algorithm. The Shift-And [2] and Shift-Or [1] pattern matching algorithms are already using bitwise operations efficiently in order to improve the performance. In this algorithm, bitwise XOR operation has been used to perform the pattern match for two bytes concurrently. i.e. Single ascii character needs 1 byte space memory and a buffer with size of a WORD is sufficient to hold the binary values of two characters (i.e. two bytes).

(ii) Added an additional loop to skip the undesirable comparisons which is required to move the text string's index to the correct location on the right side, for the subsequent comparison(s). Usually, the existing algorithms start comparing each characters (or bytes) which go through the complete main cycle though it contains the mismatch characters in the beginning, but the mismatched characters can be skipped until the first letter's match occurs. So an additional loop is essential, as and when the algorithm re-initiates the Pattern search from the starting character of pattern string (i.e. P's current index j = 0).

As the intention of this study is to propose the good solution for pattern search on DNA sequence, KMP algorithm has been chosen for the customization. Because, other algorithms like Boyer-Moore [4], are well suitable only if the alphabets size σ is huge and KMP performs well when the size of the Alphabets are small. As the intention is to deal with only four English letters ∑= {A, G, T, C}, using KMP algorithm approach gives better result. Also this algorithm has been intended to process two characters (or bytes) in single operation, so it is essential to make sure that T and P have minimum of two characters from the current string array location. Thus a '0' has been appended to text string T and pattern string P. The figure 1 shows the sample text string T and pattern string P, and the same being used to explain the new algorithm.

Text T:

G	G	T	C	A	G	G	A	G	T	C	A	G	T	C	A	A	A	G	T	0

Pattern P:

A	G	T	C	A	0

Fig 1. Text string T and Pattern string P with '0' appended for the safe XOR operation.

The Exclusive OR (or XOR) operation on two bits result single true value (1) if the opposite bits are not equal to each other otherwise it results false value (0). The $\oplus$ operator has been used to refer the XOR operation on two bits.

Table 1. XOR Operation – Truth Table

Input		Output
A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

The decimal and binary representation for the DNA sequences, ∑= {A, G, T, C}, have been shown in the Table 2.

Table 2. Decimal and Binary representation of DNA alphabets.

Character	Decimal	Binary
A	65	1000001
G	71	1000111
T	84	1010100
C	67	1000011

For instance, the result of XOR operation between "AA" and "AG" are shown in the following Table 3.

Table 3. The result of XOR Operation on "AA" and "AB".

	Character		Binary	
S1	A	A	1000001	1000001
S2	A	G	1000001	1000111
S1 ⊕ S2			0000000	0000110

In short, the XOR operation gives '0' result when text string characters and pattern string characters are matched and return non zero if both are non-identical. The actual binary representation of given text string T and pattern string P would look like Figure 2 in memory:

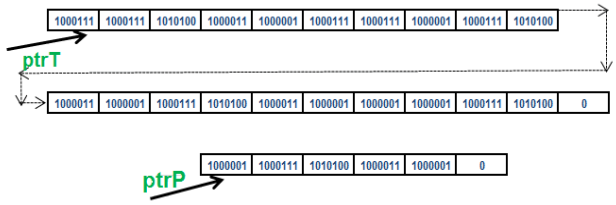

Fig 2. Binary representation of T and P in memory.

This algorithm starts with pre-processing the input pattern string P and construct the table with length of m, with the integer values of the longest prefix[0..j] which is also the suffix of P[1..j-1], where j is the mismatched character position in P.

Table 4. Failure Index table for the pattern string P.

j	0	1	2	3	4
P(j)	A	G	T	C	A
F(j)	0	0	0	0	1

The Table 4 shows the failure indexes of the pattern string P and these values will be used when mismatch occurs. So, F is an integer array of size m will have the prefix size for the safe shift when mismatch occurs.

After the Failure Index table construction, algorithm will initiate the pattern match and start with the first two bytes from both the strings T and P respectively. The bitwise XOR operation on the two consecutive bytes of T and P will let the algorithm to make either one of the following decisions:

Case 1:

Result of xor operation is 0 indicate that both the characters are matched. If the current pattern index is two byte away from the pattern string length P, algorithm will conclude that the pattern match found and starting index of pattern P in T will be added into the output array (store the pattern match locations) and current index of P will be set to 0 (i.e j = 0). Otherwise current index of pattern will be incremented twice. In either case, current index of T (i.e i) will be incremented twice as the algorithm has successfully compared the two consecutive bytes.

Case 2:

Result of xor operation is not 0 but the current (or first) index's character from T and P matched. This case indicate that only the single character is matched and still the algorithm needs to check the pattern match if the current index of P is only one byte away from the length of pattern string P. If so, the pattern match found and starting index of pattern P in T will be added into the output array and current index of P will be set to 0. Otherwise, index of P will be set to the P's index value of Failure Index array. Since only one character match found, index of T gets incremented by 1.

Case 3:

Result of xor operation is not 0 and the current index characters from T and P are not matched, and current index of P > 0. When this case get satisfied, only index of P must be updated as there could be valid character skips are possible. So the j-1th index value of Failure Index array will be assigned to j.

Case 4:

If none of the above cases (case 1, case 2 and case 3) are matched, P's current index j will be equals to 0. Hence only T's index i must be incremented only once as none of the two consecutive characters are matched. Furthermore, the algorithm skips the T's current index to the next right location where T's current index byte and P's first index byte must match. This is the additional loop, will be executed until it finds the first character of P in T from the current location or last possible location to be searched.

The outer loop of this algorithm will continue until the last character of T touched, if the pattern match found in the last or it decides the remaining characters of T from the current location doesn't seem to be a pattern. The Pattern search using the new algorithm has been explained as follows using an example text and pattern strings T and P respectively:

Step 1:

The algorithm starts with comparing first two consecutive bytes (highlighted) of T with P, shown in the Figure 3 as follows:

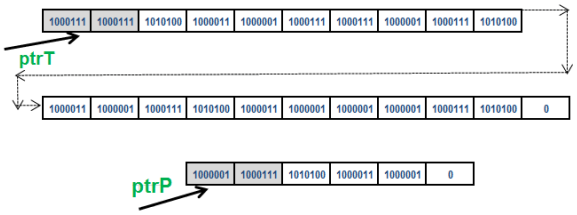

Fig 3. First step of algorithm compares two consecutive bytes.

In this case, none of the bytes matched with each other and P's starting index is 0. So Case 4 will get executed and inner loop will move T's index on the right side where T [i] = P [0] or i will be assigned with the length of T if the remaining characters length in T is less than pattern length. After the right move of ptrT, it will point the new location, shown in Figure 4 as the pattern string exists on the right side.

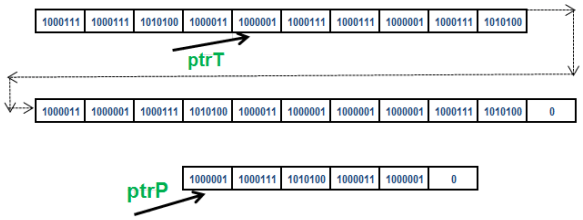

Fig 4. Text string T's current index moved to the correct index on the right side.

Step 2:

As the text string index and pattern string index are pointing the correct next match location on the respective strings, XOR operation applied on the two consecutive highlighted bytes shown in Figure 5.

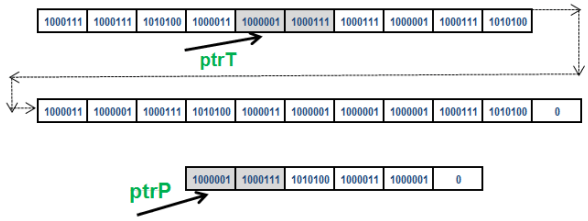

Fig 5. XOR operation applied on current two consecutive bytes between T and P.

Step 3:

As the result of previous XOR operation in Step 2 returns 0, two pointers of T ad P are further incremented twice, shown in Figure 6.

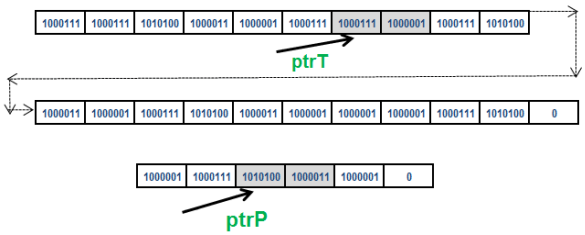

Fig 6. XOR operation on the subsequent bytes of previous Step (2).

Step 4:

In Step3, XOR operation's result is non-zero value and none of the characters were matched. However current index of P is greater than 0, hence case 3 gets executed. So only P's index get modified with the value of FailureIndex table value, shown in Figure 7.

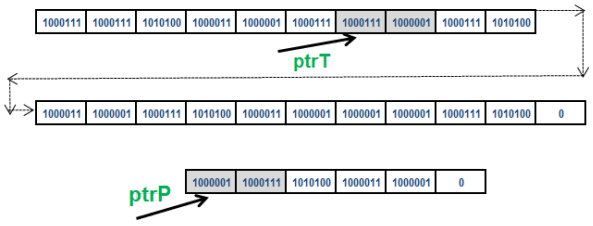

Fig 7. P's index is reassigned to the starting location

After re-initializing the P's index with starting location and XOR operation on the current two consecutive bytes of P and T still gives non-zero value and this time case 4 gets executed. Because none of the characters are matched and current starting index of P is zero. So, Case 4 will increment the index of T until the T's current iterating character matches P's first character. Eventually the inner loop will stop moving the iterator soon after it reaches the matching character in T which is shown in Figure 8.

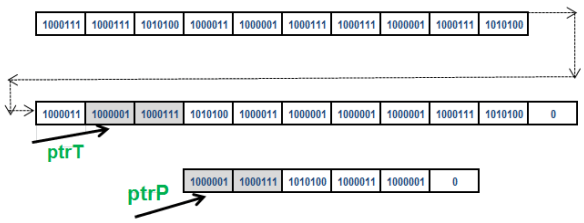

Fig 8. Modified T's Index to point the right side character of T where P's first character matches with T's current character.

Step 5:

In Step 4, two consecutive characters were matched and Case 1 increments the indexes of two strings by 2, and the new comparison will start from the following index shown in Figure 9.

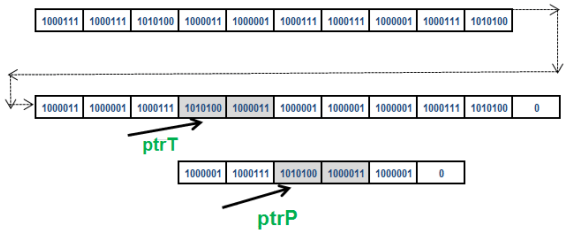

Fig 9. Incremented indexes of P and T by 2.

Step 6:

As the two bytes matched in the previous Step 5, again Case 1 will get executed and both the strings indexes were further incremented by 2 and new pointer locations are shown in the following Figure 10.

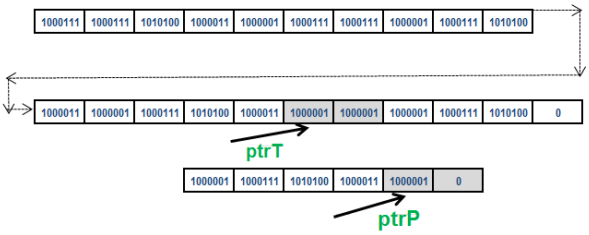

Fig 10. Latest two bytes of P are being compared with two bytes of T.

Figure 10 clearly shows that combined XOR operation for two bytes gives non-zero value. However first byte matches and its XOR value is 0. So Case 2 will get executed where the current P's index is equal to the length of the pattern string. Hence it reports the pattern match.

Algorithm

Input:

(i) Text - given text string where the pattern string will be searched in.

(ii) Pattern - given pattern string which will be searched in the text string.

Output:

(i) oArray[] – an output array has the indexes of pattern found locations in text string. If the given pattern string doesn't exist in text string, first index of oArray will be assigned with -1 (no match).

Precondition:

(i) length of the pattern string must be ≤ length of the text string.

Let iTextLength be the length of the text string.

Let iPatternLength be the length of the pattern string.

Let wXORVal be the word type variable used to store the result of XOR operation.

Let FI be an array of integers store the length of safest shift when the mismatch occur.

Let FailureIndexes be the function which computes the size of largest prefix of Pattern[0..j-1] which is also the suffix of Pattern Pattern[1..j-1] where j is the current mismatch position of pattern string.

```
i ← j ← 0
oArray[0] = -1
Count = -1
FI ← Compute FailureIndexes(Pattern)

While ( i  <  iTextLength ) Begin
   wXORVal ← Text[i..i+1] ⊕ Pattern[j..j+1]
   If ( 0 = wXORVal) Then
         If ( j  ≥  iPatternLength - 2 ) Then
            Count ← Count + 1
            oArray[Count] ← i - j + 1
            i ←  i – j +  iPatternLength
            j ← 0
         Else
            i ← i + 2
            j ← j + 2
         End If
  Else If ( 0 = ( wXORVal & 0x00FF )) Then
         If ( j = iPatternLength -1 ) Then
             Count ← Count + 1
             oArray[Count]  ←  i – j + 1
             j ← 0
         Else
             j ← FI [  j  ]
         End If
         i ← i + 1
   Else If  (j > 0 )
         j ← FI[  j - 1 ]
   Else
          i ←  i + 1
         While( Text[i] != Pattern[0] &&
                ( i + iPatternLength  ≤ iTextLength )  )
         Begin
             i ← i + 1
         End While
         If ( (i + iPatternLength) > iTextLength) Then
            i ← iTextLength
         End If
   End IF
End While
```

In this algorithm, Text [i..i+1] refers two consecutive characters from the current character (i.e. current and next character). As this algorithm process two bytes simultaneously, it reads two consecutive characters (or bytes) from both text string as well as pattern string. Also the newly introduced small inner loop's iterations are not being counted as it is used to just skip the non-matched characters and doesn't involve any complete cycle of comparisons (all the four cases).

4. EXPERIMENTAL RESULT

The newly proposed algorithm is referred as "Enhanced KMP" algorithm. The existing and newly proposed algorithms are implemented in MS Visual C++ 2010 on Windows 7 (64 bit) Operating System. The Hardware configuration is:

Intel(R) Core(TM) i5 CPU 760@2.80 GHz processor and 6 GB RAM. Also the DNA sequences from National Center for Biotechnology Information (NCBI) [21] have been used to test the performance of the newly proposed algorithm. Furthermore, this algorithm uses WORD type pointer to easily iterate the text string T as well as pattern string P. This is really essential to access two consecutive bytes in parallel and apply XOR operation between two values.

Table 5. Running time of Pattern matching algorithms (in milli-seconds).

Algorithm	Running Time		
	Experiment 1	Experiment 2	Experiment 3
Brute Force	0.008407	0.010056	0.009774
Boyer-Moore	0.004471	0.003356	0.004803
KMP	0.003013	0.003058	0.003148
Enhanced KMP	0.002788	0.002211	0.001384

Running time comparison, Table 5 shows that newly proposed Enhanced KMP algorithm's performance is better than the existing algorithms.

Table 6. Iterations required for Pattern Matching algorithms.

Algorithm	Number of Iterations		
	Experiment 1	Experiment 2	Experiment 3
Brute Force	1644	1803	2994
Boyer-Moore	733	1009	1266
KMP	824	1221	1326
Enhanced KMP	529	546	850

As an additional loop has been introduced in the new algorithm, it could reduce some outer loop's iterations and eventually newly proposed algorithms takes very few iterations to process the pattern.

The basic KMP algorithm requires O (n+ m) time for the pattern search process. As this algorithm compares two characters simultaneously, it can reduce the effort by half in the best case scenario. However it acts like KMP algorithm in the worst case scenario, other than the time gaining due to the bitwise comparison. So this algorithm also requires O (n + m) time complexity.

5. FURTHER CUSTOMIZATION

In a fixed-length encoded DNA sequence, each base is represented by a couple of bits [18]. So the DNA sequence letters {A, G, T, and C} can be mapped to {00, 01, 10, and 11} in order to save the memory and improve the processing speed.

In this algorithm, only two consecutive characters (or bytes) have been processed simultaneously. As two bits are sufficient to represent the DNA base, same bitwise XOR operation can be used to compare multiple bases using single operational cost. However, finding the first mismatched binary location logic must be proposed if there is any mismatch during the comparison, in order to improve this algorithm further.

6. CONCLUSION

Since pattern matching algorithms are very essential in the current scenario, an enhanced version of algorithm has been proposed. The proposed algorithm is based on the existing KMP algorithm and uses XOR operation to process two bytes in parallel. Also it uses an additional loop to skip the undesirable characters efficiently and takes only few iterations to find the pattern in the given text string, and eventually it gives better performance. Further research will address the problem of searching pattern in an encoded DNA sequence using XOR operation and new efficient solution for the first non-matched encoded character from the XOR operation will be proposed.

7. REFERENCES

[1] Baeza-Yates, R. A. and Gonnet, G. H. 1992. A new approach to text searching. Communications of the ACM 35, 10, 74–82.

[2] Bálint Dömölki, An algorithm for syntactical analysis, Computational Linguistics 3, Hungarian Academy of Science pp. 29–46, 1964.

[3] Knuth, Donald; Morris, James H., jr; Pratt, Vaughan (1977). "Fast pattern matching in strings". SIAM Journal on Computing 6 (2): 323–350.

[4] Boyer, Robert S.; Moore, J Strother (October 1977). "A Fast String Searching Algorithm.". Communications of the ACM (New York, NY, USA: Association for Computing Machinery) 20 (10): 762–772.

[5] Karp, R.M., Rabin, M.O., 1987, Efficient randomized pattern-matching algorithms, IBM Journal on Research Development 31(2):249-260.

[6] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel: Linear size finite automata for the set of all subwords of a word: an outline of results. Bull. Eur. Assoc. Theor. Comput. Sci., 21 1983, pp. 12–20.

[7] M. Crochemore: Optimal factor tranducers, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., vol. 12 of NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1985, pp. 31–44.

[8] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas: The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci., 40(1) 1985, pp. 31–55.

[9] C. Allauzen, M. Crochemore, and M. Raffinot: Factor oracle: a new structure for pattern matching, in SOFSEM'99, J. Pavelka, G. Tel, and M. Bartosek, eds., LNCS 1725, Milovy, Czech Republic, 1999, Springer-Verlag, Berlin, pp. 291–306.

[10] Galil, Z., and Giancarlo, R. Improved string matching with k mismatches. SIGACT News 17 (1986), 52-54.

[11] Landau, G. M., and Vishkin, U. Fast string matching with k differences. J. Comput. System Sci. 37 (1988), 63-78.

[12] Landau, G. M., and Vishkin, U. Fast parallel and serial approximate string matching. Journal of Algorithms 10 (1989).

[13] J. Holub and B.Durian. Fast variants of bit parallel approach to suffix automata. Unpublished Lecture, University of Haifa, April 2005.

[14] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithms, 5(4):1-36, 2000.

[15] H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In LNCS 2857, Proceedings of SPIRE'2003, pages 80-94, 2003.

[16] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In CPM, volume 1448 of LNCS, pages 14–33. Springer-Verlag, 1998.

[17] B. Commentz-Walter. A string matching algorithm fast on the average. In ICALP, volume 71 of LNCS, pages 118–132, 1979.

[18] Simone Faro, Thierry Lecroq: An Efficient Matching Algorithm for Encoded DNA Sequences and Binary Strings. CPM 2009: 106-115.

[19] J. W. Kim, E. Kim, and K. Park. Fast matching method for DNA sequences. In Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, volume 4614 of LNCS, pages 271–281, 2007.

[20] D. M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, 1990.

[21] Human DNA Sequences downloaded from, ftp://ftp.ncbi.nlm.nih.gov

AUTHOR'S PROFILE

K. P. Ambika is a Research Scholar in Manonmaniam Sundaranar (MS) University, Tirunelveli, India. She has completed her M.Sc degree from MS University, Tirunelveli and M.Phil degree from Bharadhidasan University, Tiruchirappalli, specialized in Biotechnology. Now she is doing her Ph.D. in Biotechnology and working on the pattern matching algorithms to predict various diseases from the DNA sequence.

Dr. U. Ramesh is working as assistant Professor in Molecular Biology Department, Madurai Kamaraj University, Madurai, India. He earned his M.Sc (Environmental Biotechnology) Ph.D (Zoology) from Manaonmanium Sundaranar University, Tirunelveli, India. He has published many research papers in various National and International Journals.

Dr. K. Saravanan is working as assistant Professor of Zoology, Nehru Memorial College (Autonomous), Puthanampatti-621 007, Tiruchirappalli (District), Tamilnadu, India. He completed his M.Sc (Wildlife Biology) and Ph.D (Zoology) at AVC College Mayiladuthruai, and he has also completed M.Sc Bioinformatics at Annamalai University, Chidambaram. He has published 25 scientific articles in reputed journals in the various fields of biological sciences.

Dr. J. Hencil Peter completed his M.C.A. and Ph.D. degrees, specialized in Computer Science, from St. Xavier's College (affiliated to MS University) Palayamkottai, India. His area of interest is clustering algorithms, cryptography and Performance Tuning. He is currently working as R&D Specialist in Playfast Technologies, Singapore and has published research papers on clustering algorithms in various National and International Journals.