Advances in Computational Sciences and Technology
ISSN 0973-6107 Volume 10, Number 8 (2017) pp. 2707-2720
© Research India Publications
http://www.ripublication.com

Survey of Exact String Matching Algorithm for Detecting Patterns in Protein Sequence

Jiji. N
Associate Professor,
Department of CSE, Younus College of Engg. & Tech.,
Kollam, Kerala, India.

Dr. T Mahalakshmi
Principal, Sree Narayana Institute of Technology,
Kollam, Kerala, India.

Abstract

Extracting and detecting normal/anomaly patterns from the massive amount of online data are gaining more attention in the world of Big Data Analysis. Last two decades, researchers proposed a number of searching algorithms for pattern matching from large volume of data. Most of these algorithms are based on string matching concept. In this paper, we have made a survey of string matching algorithms for pattern matching in protein sequence. The exact string matching algorithms are taken for this survey. And only protein sequence datasets are considered for experiment evaluation.

Keyword: String matching, pattern matching, gene database

I. INTRODUCTION

String matching is a process of finding a particular string pattern from a large volume of text. String matching detects a particular string pattern from the stored data. Nowadays, most of the applications are using string matching concept for data retrieval or pattern matching from massive amount of data. Formal definition is given below,

Definition 1: Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are vectors of elements of Σ. The pattern is a combination of alphabets from the Σ

and the searching pattern from$\Sigma = \{A, \dots, Z|a, \dots, z|0, \dots 9\}$. Other applications may use *binary alphabet* (Σ = {0,1}) or *DNA alphabet* (Σ = {A,C,G,T}) in bioinformatics.

Definition 2: Find one, or more generally, all the occurrences of a pattern $x = [x_0 x_1 x_2 \dots x_{m-1}];\ x_i \in \Sigma; i = 0,1,2, \dots m-1$, in a text of $y = [y_0 y_1 y_2 \dots y_{n-1}]$; $y_j \in \Sigma; j = 0,1,2, \dots, n-1$

There are two techniques of string matching ,

1) Exact String Matching : For given two strings T and P and wants to find all substrings of T that are equal to P. Formally, it calculates all indices i such that $T[i+s] = P[s]$ for each $0 \leq s \leq |P| - 1$. The following algorithms are used to find the exact substring matching, Needleman Wunsch (NW) [16], Smith Waterman(SW) [15], Knuth Morris Pratt (KMP) [7], Dynamic Programming, Boyer Moore Horspool (BMH) [13].
2) Approximate String Matching: Approximate string matching (fuzzy string searching) is the technique of finding strings that match a pattern approximately (rather than exactly). The problem of approximate string matching is typically divided into two sub-problems: finding approximate substring matches inside a given string and finding dictionary strings that match the pattern approximately (Fuzzy string searching, Rabin Karp, Brute Force).

Various string matching algorithms were proposed by the researchers over the period of time for solving the string matching problems. The following algorithms are widely used in various string matching applications, wide window pattern matching, approximate string matching, polymorphic string matching, string matching with minimum mismatches, prefix matching, suffix matching, similarity measure, longest common subsequence (dynamic programming algorithm), BMH, Brute Force, KMP, Quick search, Rabin Karp [12].

We have used Gene Database for analyzing the similarity measurements by using various kinds of string matching algorithms such as Boyer Moore (BM) algorithm [13], NW algorithm, SW algorithm, Hamming Distance, Levenshtein Distance, Aho-Corasick (AC) algorithm [12], KMP algorithm, Rabin Karp algorithm [14], CommentZ-walter (CZW) algorithm. We have compared the proposed two methods with these algorithms.

The remaining sections are organized as follows; section 2 provides a survey on basic string matching algorithms with applications. Section 3 provides detailed view about the proposed two string matching algorithms with examples. In section 4, we have provided the result and discussion for the proposed algorithm with related string matching algorithm. Section 5 gives the conclusion with remarks.

II. RELATED WORK

In this section, we have given an extensive survey of exact string matching/searching algorithms with simple explanations, outline of algorithm and time complexity. The string searching methods are classified into two main groups as mentioned early, exact

string matching and approximate string matching. Here we have considered only the exact string matching algorithms.

Brute Force algorithm

The brute force string matching algorithm is a classical matching scheme and this algorithm requires no preprocessing phase. The brute force algorithm consists in checking, at all positions in the text between 0 and *n-m*, whether an occurrence of the pattern starts there or not. The extracted patterns are compared one by one character. The searching window is shifted exactly one position from right to left. The searching may start from any order (left to right/right to left). The searching phase time complexity is $O(mn)$ and minimum $2n$ expected text character comparisons. This following algorithm illustrates the steps in brute force algorithm,

```
Brute_Force (String Text[ ], String Pattern[ ], int m, int n)
For j = 0 to n − m
Begin
   For i = 0 to m − 1
   Begin
      If Text[j + i] == Pattern[i]
           Continue;
     Else    Break;
     return i;
   End
return 0
End
```

Figure 1: Brute Force String matching algorithm

Deterministic Finite Automaton algorithm [17]

The basic principle behind this method is to search the given pattern by using finite state automaton. Each character from the pattern has a state and each match sends the automaton into a new state. If all the characters in the pattern have been matched then the automaton enters the accepting state. Otherwise, the automaton will return to a suitable state according to the current state and the input character such that this returned state reflects the maximum advantage we can take from the previous matching. This algorithm takes O(n) time since each character is examined once. The finite automaton based string matching technique is very efficient and it examines each character in the text exactly once and reports all the valid shifts in O(n) time

```
FINITE-AUTOMATON-MATCHER(Text, δ,m)

1. n ← length[Text]
2. q ← 0
3. For i ← 1 to n
4. do
5.     q ← δ(q, Text[i])
6.     If q == m then
7.         print "Pattern occurs with shift i − m
```

Figure 2: Finite-Automation String Matching Algorithm

Karp-Rabin algorithm [14]

Hashing provides a simple method to avoid a quadratic number of character comparisons in most practical situations. Rabin-Karp string searching algorithm calculates a numerical (hash) value for the pattern *p*, and for each *m*-character substring of text *t*. Then it compares the numerical values instead of comparing the actual symbols. If any match is found, it compares the pattern with the substring by naive approach. Otherwise it shifts to next substring of *t* to compare with *p*. This algorithm illustrate the basic steps in the Karp-Rabin algorithm. The time complexity for the Karp-Rabin algorithm is $O(n + m)$

```
Rabin-Karp ( String Text [ ], String pattern [ ])

Hash_pattern = HASH(pattern[ ])
  For i ← 1 to n − m + 1
      hs = HASH(Text[i ... i + m − 1])
          If (hs == Hash_pattern)
              If(Text[i ... i + m − 1] ==
pattern[1 ... m])
                  return i
  return not found
```

Figure 3: Karp-Rabin String Matching Algorithm

Morris-Pratt algorithm [18]

Knuth, Morris and Pratt discovered first linear time string-matching algorithm by analysis of the naïve algorithm. It keeps the information that naive approach exhausted gathered during the scan of the text. This method avoids this exhaust of information and it achieves a running time of$O(m + n)$. The implementation of Knuth-Morris-Pratt algorithm [7] is efficient because it minimizes the total number of comparisons of the pattern against the input string

```
Morris_Pratt (String Text [ ], String pattern [ ])
 1. n ← length[Text]
 2. m ← length[pattern]
 3. a ← Compute Prefix function
 4. q ← 0
 5. For i ← 0 to n do
 6.     While (q > 0  and pattern[q + 1], Text[i])  do
 7.           q ← a[q]
 8.            If [pattern[q + 1] == S[i]) then
 9.                q ← q + 1
10.            End If
11.            If (q == m) then
12.                q ← a[q]
13.            End If
14.     End While
15. End For
```

Figure 4: Morris-Pratt String Matching Algorithm

Colussi algorithm [19]

Colussi string matching algorithm is a refined form of the Knuth, Morris and Pratt string matching algorithm. This algorithm partitions the set of pattern positions into two disjoint subsets and the positions in the first set are scanned from left to right and when no mismatch occurs the positions of the second subset are scanned from right to left. The preprocessing phase needs $O(m)$ time complexity and space complexity and searching phase in $O(n)$ time complexity. The Colussi string matching algorithm takes $\frac{3}{2}n$ text character comparisons in the worst case.

Boyer-Moore algorithm [13]

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of this algorithm is implemented in text editors for the searching and substitution of words in the text. This algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the *good-suffix shift* (also called matching shift) and the *bad-character shift* (also called the occurrence shift).

The preprocessing phase needs $O(m + n)$ time and space complexity and the searching phase needs in $O(mn)$ time complexity. This algorithm needs $3n$ text character comparisons in the worst case when searching for a non periodic pattern.

Boyer_Moore ($String\ Text$ [ ], $String\ pattern$ [ ])
1. $i \leftarrow m - 1$ 2. $j \leftarrow n - 1$ 3. Repeat 4. $If\ (pattern[j] == Text[i])$ then 5. $If\ j == 0\ \ then$ 6. $return\ i$ 7. $Else$ 8. $i \leftarrow i - 1$ 9. $j \leftarrow j - 1$ 10. $Else$ 11. $i \leftarrow i + m - Min(j, 1 + last[T[i]])$ 12. $j \leftarrow m - 1$ 13. $Until\ i > n - 1$ 14. $return$ "$no\ match$"

Figure 5: Boyer-Moore String Matching Algorithm

Turbo-BM algorithm [20]

The Turbo-BM algorithm is a version of the Boyer-Moore algorithm. It needs no extra preprocessing and requires only a constant extra space with respect to the original Boyer-Moore algorithm. It consists in remembering the factor of the text that matched a suffix of the pattern during the last attempt (and only if a good-suffix shift was performed). This method has two advantages, it is possible to jump over this factor and it can enable to perform a turbo-shift.

Tuned Boyer-Moore Algorithm [21]

The Tuned Boyer-Moore is a simplified version of the Boyer-Moore algorithm which is very fast in practice. The most costly part of a string-matching algorithm is to check whether the character of the pattern match the character of the window. To avoid doing this part too often, it is possible to unrolled several shifts before actually comparing the characters. The comparisons between pattern and text characters during each attempt can be done in any order. This algorithm has a quadratic worst-case time complexity but a very good practical behavior. This algorithm requires O(mn) for the searching phase time complexity

Reverse Colussi algorithm [22]

The Reverse Colussi string matching algorithm is a refined algorithm of the Boyer-Moore string matching algorithm. This algorithm partitions the set of pattern positions into two disjoint subsets and the character comparisons are done using a specific order given by a table. The preprocessing phase requires $O(m^2)$ time and searching phase

requires $O(n)$ time complexity. This algorithm needs minimum $2n$ text character comparisons in the worst case.

Apostolico-Giancarlo algorithm [23]

The Boyer-Moore string matching algorithm is difficult to analyze because after each attempt it forgets all the characters it has already matched. Apostolico and Giancarlo designed an algorithm which remembers the length of the longest suffix of the pattern ending at the right position of the window at the end of each attempt. This information is stored in a table *skip*.

Let us assume that during an attempt at a position less than j the algorithm has matched a suffix of x of length k at position $i + j$ with $0 < i < m$ then $skip[i + j]$ is equal to k. Let $suff[i]$, for $0 \leq i < m$ be equal to the length of the longest suffix of x ending at the position i in x

The complexity in space and time of the preprocessing phase of the Apostolico-Giancarlo algorithm is $O(m + n)$ same as the Boyer-Moore algorithm. During the search phase only the last m information of the table *skip* are needed at each attempt so the size of the table *skip* can be reduced to $O(m)$. The Apostolico-Giancarlo algorithm performs in the worst case at most $O(\frac{3}{2}n)$ text character comparisons

Smith-Waterman algorithm [15]

This algorithm computes the shift with the text character just next the rightmost text character of the window gives sometimes shorter shift than using the rightmost text character of the window. Smith takes the maximum between the two values. The preprocessing phase of the Smith algorithm consists in computing the bad-character shift function (Boyer-Moore algorithm) and the Quick Search bad-character shift function (Quick Search algorithm). The preprocessing phase requires $O(m + \sigma)$ time and $O(\sigma)$ space complexity. The searching phase of the Smith algorithm has a quadratic worst case time complexity

Needleman–Wunsch algorithm [16]

The Needleman-Wunsch string matching algorithm essentially divides a large problem (e.g. the full sequence) into a series of smaller problems and uses the solutions to the smaller problems to reconstruct a solution to the larger problem. It is also sometimes referred to as the optimal matching algorithm and the global alignment technique. This works under the principle of dynamic programming. The Needleman–Wunsch algorithm is still widely used for optimal global alignment, particularly when the quality of the global alignment is of the utmost importance. The processing time for searching a pattern from the given text is$O(mn)$.

Raita algorithm [24]

Raita designed an algorithm, it first compares the last character of the pattern with the rightmost text character of the window and if they match it then compares the first character of the pattern with the leftmost text character of the window, if they match it then compares the middle character of the pattern with the middle text character of the window. And finally if they match it actually compares the other characters from the second to the last but one, possibly comparing again the middle character.

Raita observed that this algorithm works well in practice when searching patterns in english texts. Smith made some more experiments and concluded that this phenomenon may rather be due to compiler effects.

The preprocessing phase of the Raita algorithm consists of computing the bad-character shift function (Boyer-Moore). It can be done in $O(m + n)$ time and $O(n)$ space complexity. The searching phase of the Raita algorithm has a quadratic worst case time complexity.

Reverse Factor algorithm [25]

The smallest suffix automaton of a word w is a Deterministic Finite Automaton $S(w) = (Q, q_0, T, E)$. The language accepted by $S(w)$ is $\left(S(w)\right) = \{u \in \Sigma^* \; exists \; v \; in \; \Sigma^*$ such that $w = vu\}$. The preprocessing phase of the Reverse Factor algorithm consists in computing the smallest suffix automaton for the reverse pattern x^R. It is linear in time and space in the length of the pattern.

During the searching phase, the Reverse Factor algorithm parses the characters of the window from right to left with the automaton $S(x^R)$, starting with state q_0. It goes until there is no more transition defined for the current character of the window from the current state of the automaton. At this moment it is easy to know what is the length of the longest prefix of the pattern which has been matched: it corresponds to the length of the path taken in $S(x^R)$ from the start state q_0 to the last final state encountered. Knowing the length of this longest prefix, it is trivial to compute the right shift to perform. The Reverse Factor algorithm has a quadratic worst case time complexity but it is optimal in average. It performs $O(n.\frac{\log_\sigma(m)}{m})$ inspections of text characters on the average reaching the best bound.

Berry-Ravindran algorithm [26]

Berry and Ravindran designed an algorithm which performs the shifts by considering the bad-character shift (Boyer-Moore algorithm) for the two consecutive text characters immediately to the right of the window.

The preprocessing phase of the algorithm consists in computing for each pair of characters (*a*, *b*) with *a*, *b* in Σ the rightmost occurrence of *ab* in *axb*. For $a, b \in \Sigma$

$$brBc[a,b] = Min \begin{Bmatrix} 1 & if\ x[m-1] = a \\ m-i+1 & if\ x[i]x[i+1] = b \\ m+1 & if\ x[0] = b \\ m+2 & otherwise \end{Bmatrix}$$

The preprocessing phase requires $O(m + \sigma^2)$ space and time complexity. The searching phase of the Berry-Ravindran algorithm has a $O(mn)$ time complexity.

Aho–Corasick algorithm [12]

It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all strings simultaneously. The complexity of the algorithm is linear in the length of the strings plus the length of the searched text plus the number of output matches.

Alpha Skip Search algorithm [27]

The preprocessing phase of the Alpha Skip Search algorithm consists of building a tree $Text(x)$ of all the factors of the length $l = \log_\sigma m$ occurring in the word *x*. The leaves of $Text(x)$ represent all the factors of length *l* of *x*. There is then one bucket for each leaf of $Text(x)$ in which is stored the list of positions where the factor, associated to the leaf, occurs in *x*.

The worst case time of this preprocessing phase is linear if the alphabet size is considered to be a constant. The searching phase consists in looking into the buckets of the text factors $y[j \ldots j+l-1]\ \forall\, j = k.(m-l+1)-1$ with the integer k in the interval$y[1, \left\lfloor \frac{n-l}{m} \right\rfloor]$. The worst case time complexity of the searching phase is quadratic but the expected number of text character comparisons is $O(\log_\sigma(m).(\frac{n}{m-\log_\sigma(m)})$. This algorithm requires preprocessing phase $O(m)$ time and space complexity and searching phase in $O(mn)$ time complexity.

III. RESULT AND DISCUSSIONS

We have used Gene dataset for experimental analysis. The gene database file is created from Genbank Accession No: JN222368 which belongs to Marine sponge. The size of the gene database is 3481 characters. In the testing environment, we have considered the searching pattern size as 34 characters for gene database. The following table and figure provides the accuracy and execution time comparison,

Table 1: Proposed algorithm Accuracy for string matching with related algorithms

Algorithm	Preprocessing	String Matching	Accuracy	Execution Time
Brute Force	Not Applicable	$O(mn)$	66.7%	$\approx 85ms$
Deterministic Finite Automaton	$O(mk)$	$O(n)$	72%	$\approx 65ms$
Rabin-Karp	$O(n)$	$O(mn)$	70%	$\approx 72ms$
Morris-Pratt	$O(m)$	$O(n+m)$	65%	$\approx 68ms$
Colussi	$O(m)$	$O(n)$	74%	$\approx 58ms$
Boyer-Moore	$O(m+n)$	$O(mn)$	83%	$\approx 84ms$
Turbo-BM	$O(m+n)$	$O(mn)$	82.52%	$\approx 86ms$
Tuned Boyer-Moore	$O(m+n)$	$O(mn)$	82.1%	$\approx 88ms$
Reverse Colussi	$O(m^2)$	$O(n)$	79%	$\approx 57ms$
Apostolico-Giancarlo	$O(m+n)$	$O(n)$	74%	$\approx 61ms$
Smith-Waterman	$O(m+n)$	$O(mn)$	71.4%	$\approx 81ms$
Needleman–Wunsch	Not Applicable	$O(mn)$	60%	$\approx 85ms$
Raita	$O(m+n)$	$O(mn)$	76%	$\approx 82ms$
Reverse Factor	$O(m)$	$O(mn)$	75.4%	$\approx 82ms$
Berry-Ravindran	$O(m+n^2)$	$O(m+n)$	77%	$\approx 74ms$
Aho–Corasick	$O(m+n)$	$O(m+n)$	79.7%	$\approx 70ms$
Alpha Skip Search	$O(m)$	$O(mn)$	78.5%	$\approx 83ms$

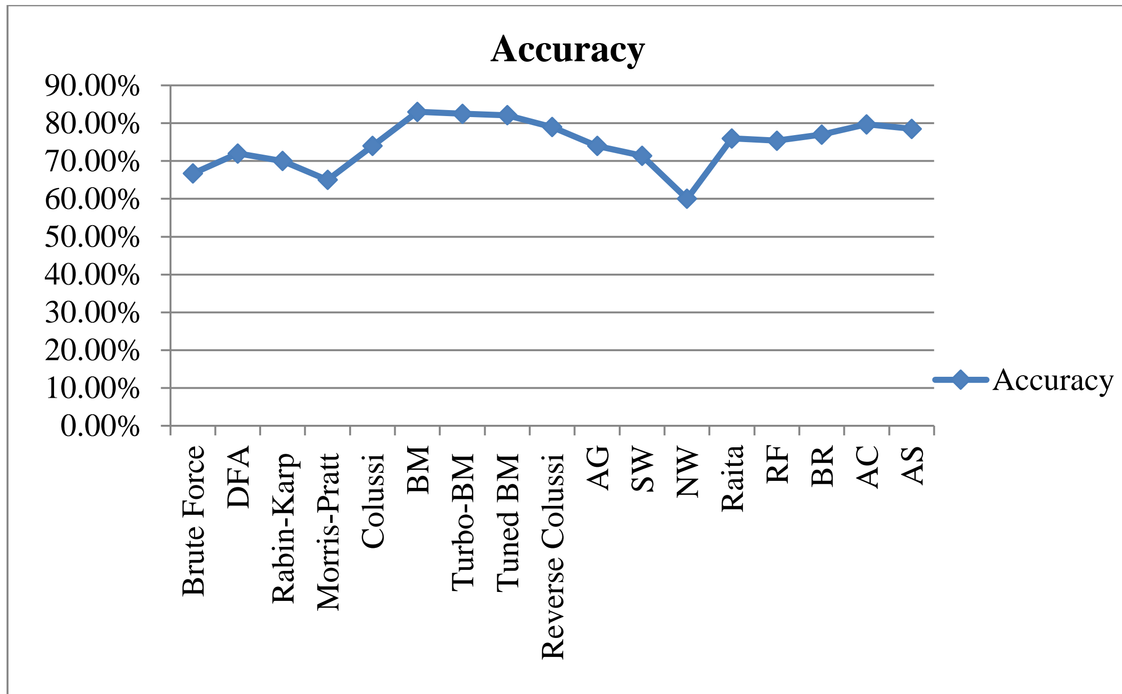

Figure 1: Accuracy comparison of String matching algorithms for Gene Dataset

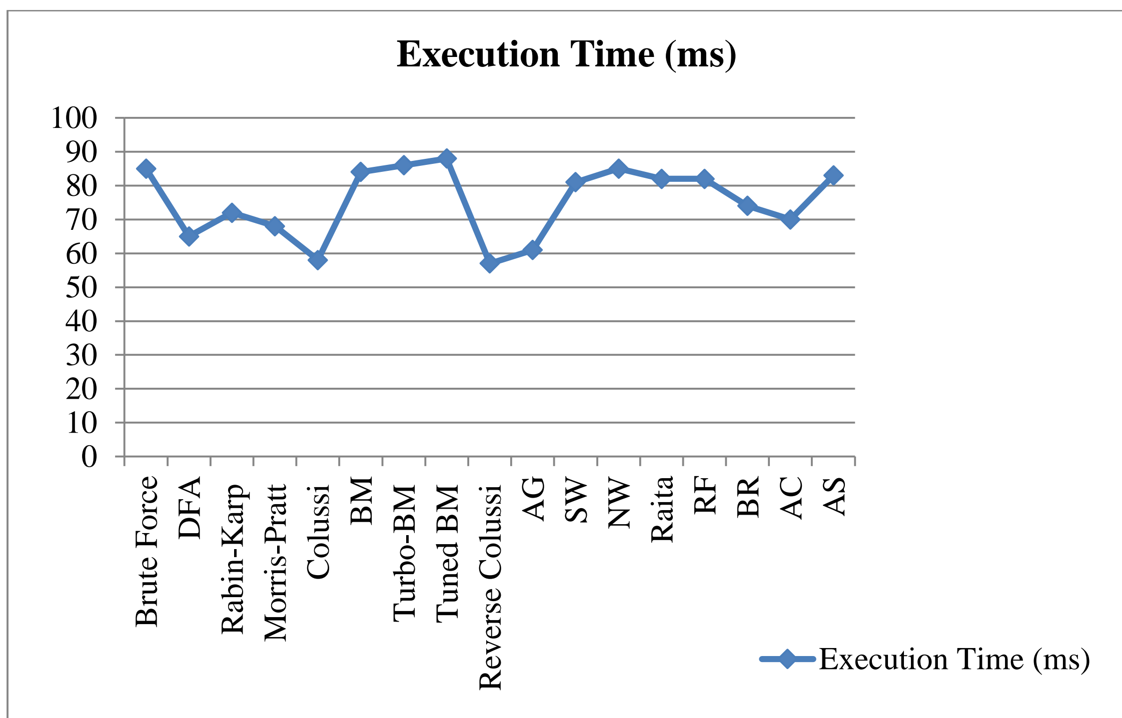

Figure 2: Execution time comparison of String matching algorithms for Gene Dataset

IV. CONCLUSION

String matching algorithms are the most important research area in the field of content retrieval and pattern searching. Most of the string matching algorithms are designed for specific applications and the accuracy differs based on the dataset. In this paper, we have made an extensive survey of exact string matching algorithms and basic working principles. We have taken 18 different string matching algorithms for comparison. We have considered the Gene Dataset for the experiment evaluation with 3481 characters and the searching pattern size is same for all the related algorithms. Boyer-Moore [13] string matching algorithm provides an accuracy of 83% and the execution time is $\approx 84ms$. Reverse Colussi [22] string matching algorithm provides an accuracy of 79% and the execution time is $\approx 57ms$. In future, researcher have a scope in the field of designing an efficient algorithm for string matching/searching algorithm for Gene Dataset, to reduce the execution time and to increase the accuracy.

REFERENCE

[1] Singla N., Garg D., “String Matching Algorithms and their Applicability in various Applications,” International Journal of Soft Computing and Engineering, ISSN: 2231-2307, Volume-I, Issue-6, January 2012

[2] Vidanagamachchi S.M., Dewasurendra S.D., Ragal B.G., Niranjan M., “Comment Z Walter: Any Better than Aho-Corasick for Peptide Sequence Identification,” International Journal of Research in Computer Science, eISSN:2249-8265, Vol 2, Issue 6(2012) PP 33-37

[3] http://www.ebi.ac.uk/Tools/psa/emboss_water/nucleotide.html

[4] Gomaa N.H., Fahmy A.A., “Short Answer Grading using String Similarity and Corpus-Based Similarity,” International Journal of Advanced Computer Science and Applications, Vol 3,No.11, 2012

[5] Jain P., Pandey S., “Comparative Study on Text Pattern Matching for Heterogeneous System,” International Journal of Computer Science and Engineering Technology, ISSN: 2229-3345, Vol.3 No.11 Nov 2012

[6] Alsmadi I., Nuser M., “String Matching Evaluation Methods for DNA Comparisons,” International Journal of Advanced Science and Technology, Vol.47, 2012

[7] Knuth D.E., Morris J.H., and Pratt V.R., “Fast Pattern Matching in Strings,” Journal of Computing, Vol.6, No.2, 1977

[8] Hussain I., Kausar S., Hussain L., and Asifkhan M., “Improved Approach for Exact Pattern Matching,” International Journal of Computer Science Issues, Vol.10, Issue 3, No.1, 2013

[9] Amir A., Lewenstein M., and Porat E., “Faster Algorithms for String Matching with K-Mismatches,” Journal of Algorithms 50(2004) 257-275

[10] Yeh M, Cheng K.T, "A String Matching Approach for Visual Retrieval and Classification," in proceeding of the ACM SIGMOD 978-1-60558-312-9/08/10

[11] Boytsov, Leonid, "Indexing methods for approximate dictionary searching: Comparative analysis," Journal of ACM. Vol. 16, No.1, pp.1–91, 2011 *doi:10.1145/1963190.1963191*

[12] Aho, Alfred V., Corasick, Margaret J, 1975, "Efficient string matching: An aid to bibliographic search," Communications of the ACM, Vol. 18, No.6, pp.333–340, 1975.

[13] Boyer, Robert S., Moore, J Strother. "A Fast String Searching Algorithm," Comm. ACM. New York, NY, USA, 1977, Association for Computing Machinery, Vol. 20, No.10, pp. 762–772

[14] Karp, Richard M., Rabin, Michael O, "Efficient randomized pattern-matching algorithms,"1987, IBM Journal of Research and Development, Vol. 31, No.2, pp. 249–260

[15] Smith, Temple F. & Waterman, Michael S, "Identification of Common Molecular Subsequences," 1981, Journal of Molecular Biology. Vol. 147, pp.195–197

[16] Needleman, Saul B. & Wunsch, Christian D, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," 1970, Journal of Molecular Biology, Vol. 48, No.3, pp.443–453

[17] Crochemore, M., Hancart, C., 1997. Automata for Matching Patterns, in *Handbook of Formal Languages*, Volume 2, Linear Modeling: Background and Application, G. Rozenberg and A. Salomaa ed., Chapter 9, pp 399-462, Springer-Verlag, Berlin

[18] Morris (Jr) J.H., Pratt V.R., 1970, *A linear pattern-matching algorithm*, Technical Report 40, University of California, Berkeley

[19] Colussi L., 1991, Correctness and efficiency of the pattern matching algorithms, *Information and Computation* 95(2):225-251

[20] Crochemore, M., Czumaj A., Gasieniec L., Jarominek S., Lecroq T., Plandowski W., Rytter W., 1992, Deux méthodes pour accélérer l'algorithme de Boyer-Moore, in *Théorie des Automates et Applications, Actes des 2e Journées Franco-Belges*, D. Krob ed., Rouen, France, 1991, pp 45-63, PUR 176, Rouen, France

[21] Hume A. and Sunday D.M., 1991. Fast string searching. *Software - Practice & Experience* 21(11):1221-1248

[22] Colussi L., 1994, Fastest pattern matching in strings, *Journal of Algorithms*. 16(2):163-189

[23] Apostolico A., Giancarlo R., 1986, The Boyer-Moore-Galil string searching strategies revisited, *SIAM Journal on Computing* 15(1):98-105

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

[25] Lecroq T., 1992, A variation on the Boyer-Moore algorithm, *Theoretical Computer Science* 92(1):119—144

[26] Berry, T., Ravindran, S., 1999, A fast string matching algorithm and experimental results, in *Proceedings of the Prague Stringology Club Workshop`99*, J. Holub and M. Simánek ed., Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, 1999, pp 16-26

[27] Charras C., Lecroq T., Pehoushek J.D., 1998, A very fast string matching algorithm for small alphabets and long patterns, in *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching* , M. Farach-Colton ed., Piscataway, New Jersey, Lecture Notes in Computer Science 1448, pp 55-64, Springer-Verlag, Berlin