ISSN: 0974-2115
www.jchps.com Journal of Chemical and Pharmaceutical Sciences

Recursive_ LCS_Pattern: A Recursive Pattern Matching algorithm to identify Longest Common Sequences in DNA Sequences

[1]B. Devika Rubi*, [2]Dr. L. Arockiam
[1]Research Scholar, Research and Development Centre, Bharathiar University, Coimbatore
[2]Associate Professor, Department of Computer Science, St Joseph's College, Tiruchirappalli
*Corresponding author: E-Mail: deviraja@gmail.com

ABSTRACT

Datamining relies on pattern matching algorithms to locate patterns using a variety of technologies from simple keyword matching to rule based expert systems. One of the key bioinformatics application of pattern recognition is pattern matching. Multiple Sequence alignment is the process to align three or more biological sequences. It is useful in finding conserved regulatory patterns in nucleotide sequences and in identifying structural and functional domains in protein families, but it is much more challenging. The existing algorithms generate all possible patterns and then search for the longest pattern. This is time consuming which requires high memory and large amount of data transfer between main memory and cache memory. This paper proposes an efficient pattern matching algorithm to identify longest common sequences/patterns (LCS) in DNA sequences using pattern recursive methodology. The proposed algorithm recursively uses the existing length_2 pattern to identify existing length_3 pattern and so on. This process reduces the search spaces comparatively lower than enumerative combinatorics.

KEY WORDS: Sequence Homology, Recursive Pattern Matching, LCS, Multiple Sequence Alignment, Enumerative Combinatorics.

1. INTRODUCTION

Datamining is the part of knowledge discovery which deals with the process of identifying hidden patterns in data (Han & Kamber, 2006). Datamining relies on pattern matching algorithms to locate patterns using a variety of technologies from simple keyword matching to rule based expert systems. One of the key bioinformatics application of pattern recognition is pattern matching. Automated pattern matching is the ability of a program to compare, identify known patterns and to determine the degree of similarity.

Sequence alignment (Durbin, 1998) is the way of arranging sequences based on their similarity. In bioinformatics, sequence alignment infers homology (common ancestry) and their functions. For Example, it is generally accepted that if two sequences are in alignment then in part or all of the pattern of nucleotides or polypeptides match then they are similar and may be homologous. Another heuristic is that if the sequence of a protein with a known structure and function then the molecules may share structure and function (Mushegian, 2011).

Multiple Sequence alignment (Wang, 2011) is the process to align three or more biological sequences. It is useful in finding conserved regulatory patterns in nucleotide sequences and in identifying structural and functional domains in protein families, but it is much more challenging (R.A.C, 2007). In actual multiple sequence alignment each bio sequence need to align several millions characters. Making manual gap insertions/deletions in the sequences and other non-computational methods like wet-lab sequence analysis etc. are infeasible (Bailey, 2001). Thus Multiple Alignment is an active research in bioinformatics because of the computational challenges involved.

This paper proposes an efficient pattern matching algorithm to identify longest common sequences/patterns (LCS) in DNA sequences using pattern recursive methodology. This paper is organized as given below. Section 2 discusses problem definition and existing algorithms. Section 3 proposes a new algorithm called Recursive_LCS_Pattern() to identify LCS. Section 4 discusses about the illustration and implementation of the proposed algorithm Recursive_LCS_Pattern(). Section 5 provides the conclusion.

2. PROBLEM DEFINITION AND EXISTING ALGORITHMS

2.1. Problem Definition: Let S1 = $a_1 a_2 a_3....a_m$ and S2 = $b_1 b_2 ...... b_n$ are the two sequences. And 'Z' is the Longest Common Subsequence (LCS) between S1 and S2, which is defined as $z_1 z_2. ....z_k$ (Hirschberg, 1975) and (Hakata & Imai, 1998).

Table.1.Sample DNA sequences

Sequence / Position#	1	2	3	4	5	6	7	8	9	10
S1	C	T	G	C	T	C	A	C	G	C
S2	C	A	A	C	T	C	T	C	A	C

Sample DNA sequences have been provided in Table 1 and their LCS Z is "C T C A C".

2.2. Existing Frequent Pattern Matching Algorithms: Sequential pattern mining (Agrawal & Srikant, 1995) is the mining of frequently occurring ordered events or subsequences as patterns. Sequential pattern mining is the knowledge-discovery process which encompasses data storage and access. It also includes scaling of algorithms to very large data sets and interpreting results.

MCFS algorithm (Karim, 2012) proposes suffix tree based approach for mining the maximal contiguous frequent subsequence from DNA sequence datasets. This algorithm introduces a combined main memory and disk-based approach for mining maximal contiguous frequent patterns from large DNA sequence databases that cannot fit into the main memory. This algorithm finds the set of contiguous frequent subsequences and then finds the set of maximal contiguous frequent patterns by checking the properties of maximalist. It creates projected database for each frequent contiguous pattern. This leads to large memory and time consuming process for tree traversal to find the set of contiguous frequent pattern.

Rashid (2012) describes about interesting patterns that is the patterns which are not frequent may still be informative in computational biology and bioinformatics. This algorithm uses two new user defined thresholds called minimum information gain threshold and minimum confidence threshold which are based on probability occurrence of characters in database. This algorithm gives a new view to pattern discovery and its measurement. Repeated scanning of DB is avoided. The search space is reduced with respect to the information gain threshold and confidence threshold. The construction of spanning tree for fixed length and repeated spanning tree traversal for finding the interesting patterns are time and space consuming process.

EBR (Suleiman, 2014) algorithm uses the same searching process used in Enhanced Two Sliding Windows algorithm (ETSW). The searching process uses two sliding windows and the comparisons between the pattern and the text are made from both sides simultaneously. EBR made enhancements on Berry-Ravindran (BR) algorithm (Berry & Ravindran, 2001); while BR uses two consecutive characters of the text immediately following the pattern window to determine the amount of shift, EBR uses three consecutive characters which maximizes the shift and the efficiency of the searching process.

DNA base calling (Timp, 2012) is difficult in Nanopore sequencing (Timp, 2010) third generation DNA sequencing due to the limitation in resolution of signal/noise ratio.

The major limitations of all existing frequent patterns algorithms are (1) Multiple scanning of database to find all base patterns which satisfy minimum support (2) Number of intermediate tables are created to store the base patterns and their positions (3) Construction cost of the spanning trees for the base patterns (4) Repeated traversal of the spanning tree (5) High run time and space complexity.

3. PROPOSED RECURSIVE_LCS_PATTERN ALGORITHM

LCS is the process to identify the longest common pattern between two or more sequences. DNA sequences are the linear arrangement of four nucleotides (A, G, T, C) in any order. The LCS for DNA sequences consist of only repetition of four characters A, G, T, C. As per Enumerative Combinatorics if “m” is the length of the pattern and “n” is the number of characters used to form the sequence, then number of possible permutations with repetitive characters are m^n. In DNA sequences “n=4” is fixed. There will be 16 (2^4) len_2 patterns, 64 (3^4) len_3 patterns, 256 (4^4) len_4 patterns and so on. As length of pattern increases number of possible patterns also increases. Hence this problem becomes an NP-Hard (Non Polynomial Deterministic time) problem. The existing algorithms generate all possible patterns and then search for the longest pattern. This is time consuming which requires high memory and large amount of data transfer between main memory and cache memory.

The proposed Recursive_LCS_Pattern algorithm considers only the existing patterns. It contains two processes namely (i). Pre-processing the DNA sequences into 16 Len_2 patterns (2). These 16 Len_2 patterns are recursively used to identify the required LCS. For an instance LCS for the sample DNA sequences given in Table 1 is “CTCAC”, which is of length_5. The length_2 patterns CT and TC are combined to produce CTC. Length_3 patterns CTC and TCA will form length_4 pattern CTCA. Subsequently length_4 pattern CTCA will be combined with length_4 TCAC to produce the required length_5 LCS “CTCAC”. Thus the length_2 patterns are recursively used to produce length_3 patterns, subsequently length_3 patterns are combined to produce length_4 and so on. This process is repeated till the required LCS is identified.

The pseudo code for proposed algorithm Recursive_LCS_Pattern () is given in Fig.1.

Fig.1.Pseudeo code for Recursive_LCS_Pattern()

```
// Recursive_LCS_Pattern algorithm to find MLCS
Procedure Recursive_LCS_Pattern()
{
// s[m] are the array of m sequences of length_n
int k = 1;
// Split the given sequences into length_2 pattern
// m represents sequence length, k represents each sequence
// x, y represents adjacent two characters of sequences s[k]
// x1, y1 represents adjacent two characters of sequences s[k+1]
```

```
While (k < = m)
{
    For i = 1 to n-1
    {
        x = s[k].charAt(i);
        y = s[k].charAt(i+1);
        x1 = s[k+1].charAt(i);
        y1 = s[k+1].charAt(i+1);
        call length_2_pattern(x, y, i, s[k]);
        call length_2_pattern(x1, y1, i, s[k+1]);
     } // End for
          k = k + 2;
} // End while
Procedure length_2_pattern(char x, char y, int i, int z)
{
    pat2 = x.concat(y);
       switch (x)
            {
              Case "A" :
                     { if (pat2 = "AA")
                        {
                           seqId_AA[i1] = z;
                           positionID_AA[i1] = i;
                           count_pattern2[1]= count_pattern2[1] + 1;
                         }
                       break;
                     }

             Case "T" :
              ………………………………………
               Case "G" :
                      { if (pat2 = "GG")
                        {
                           seqId_GG[i1] = z;
                           positionID_GG[i1] = i;
                           count_pattern2[16]= count_pattern2[16]+1;
                         }
                       break;
                      }
                   } End Switch case
} // End length_2 pattern procedure
Procedure length_3_AA(seqID_AA[], positionID_AA[], count_pattern2[i])
{
// k = count_pattern2[i]
// Combine pattern_AA and pattern_AT

     For j = 1 to k
     {
          // Verification for the occurrence of pattern_AT
          If (count_pattern2[2] > 0)
          {
               For j1 = 1 to count_pattern2[2]
                     { if (positionID_AA[j] = = positionID_AT[j1] + 1)
                        {
                              seqId_AAT[i1] = j;
                              positionID_AAT[i1] = j1;
                              count_pattern3[2]= count_pattern[2] + 1;
```

```
                    }

                  } // End of  inner-if
             } // End of  inner-for
         } // End of outer-if
       } // End of outer-for

} // End length_3_AA procedure

} // End of procedure Recurisve_LCS_Pattern()
```

The pseudeo code of Recurisve_LCS_Pattern() shows that the given sequence are splitted into length_2 patterns which are passed as an input to length_3 patterns. This process is repeated until the required LCS is found.

4. ILLUSTRATION AND ILLUSTRATION

4.1. Illustration of Recursive_LCS_Pattern algorithm: In this section, the proposed Recursive_LCS_Pattern algorithm has been illustrated for the sample DNA sequences in Table 1. The length_2 patterns generated by Recursive_LCS_Pattern algorithm shown in Table 2.

Table.2.Length_2 patterns generated by the Recursive_LCS_Pattern algorithm

Len_2 Pattern	(SeqID, PosID)	# Comparisons	Size	Selected/Rejected
AA	(20,2)	4	2	Rejected
AT	-			Rejected
AC	(10,7), (20,3), (20,9)	4	6	Selected
AG	-			Rejected
TA	-			Rejected
TT	-			Rejected
TC	(10,5), (20,5), (20,7)	4	6	Selected
TG	(10,2)	4	2	Rejected
CA	(10,6), (20,1), (20,8)	4	6	Selected
CT	(10,1), (10,4), (20,4), (20,6)	4	8	Selected
CC	-			Rejected
CG	(10,8)	4	2	Rejected
GA	-			Rejected
GT	-			Rejected
GC	(10,3), (10,9)	4	4	Rejected
GG	-			Rejected

Some of the length_2 patterns in Table 2 are rejected due to either total number of patterns is 1 or not common for both the sequences. For eg. Pattern AA, TG, CG are occurring only at seqID_10 and Pattern GC is occurred only at seq_ID_10.

Similarly the generated, length_4 patterns and length_5 patterns by the algorithms are shown in Table 3 and Table 4.

Table.3.Length_4 patterns generated by the Recursive_LCS_Pattern algorithm

Len_4 Pattern	(SeqID, PosID)	# Comparisons	Size	Selected /Rejected
TCAC	(10,5), (10,6),(10,7), (20,7), (20,8), (20,9)	8	12	Selected
CTCA	(10,4), (10,5), (10,6), (20,4), (20,5), (20,6), (20,7)	12	14	Selected

Table.4.Length_5 patterns generated by the Recursive_LCS_Pattern algorithm

Len_5 Pattern	(SeqID, PosID)	# Comparisons	Array Size	Selected /Rejected
CTCAC	(10,4), (10,5), (10,6), (10,7) (20,4), (20,5), (20,6), (20,7), (20,8), (20,9)	21	20	Selected

Tables 2, 3 and 4 show that number of comparisons needed to get the patterns and the memory space needed to store the selected recursive patterns.

4.2. Implementation details and Results: This algorithm has been implemented using Java on a Windows 10 machine with i7 Intel processor 2.33 GHZ, 16 GB RAM. Table 5 shows that number of comparisons and memory space needed for length_2, length_3, length_4 and length_5 patterns.

Table.5. Number of Comparisons and memory space needed for each length pattern

Pattern Length	# Comparisons	Array Size
2	32	36
3	33	36
4	20	26
5	21	20

The graphical representation of runtime results of Recursive_LCS_Pattern algorithm for the given sequences in Table 1 are shown in Fig 2 and 3.

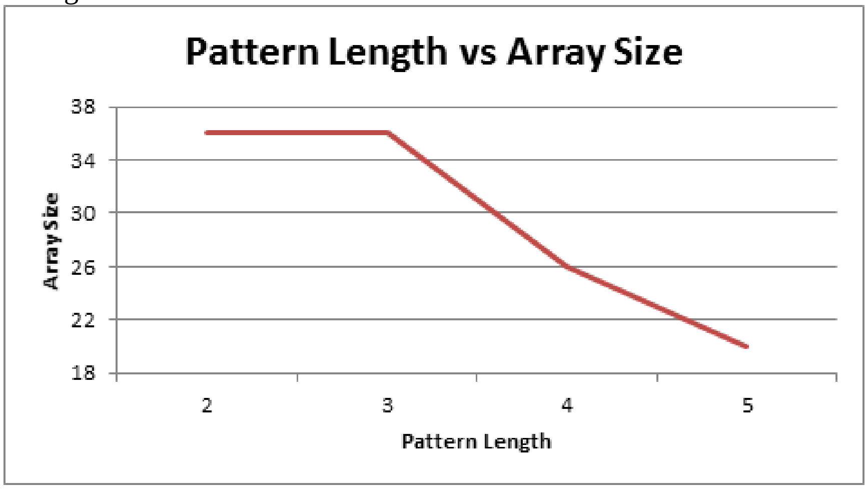

Fig.2.Recursive_LCS_Pattern algorithm runtime results for memory

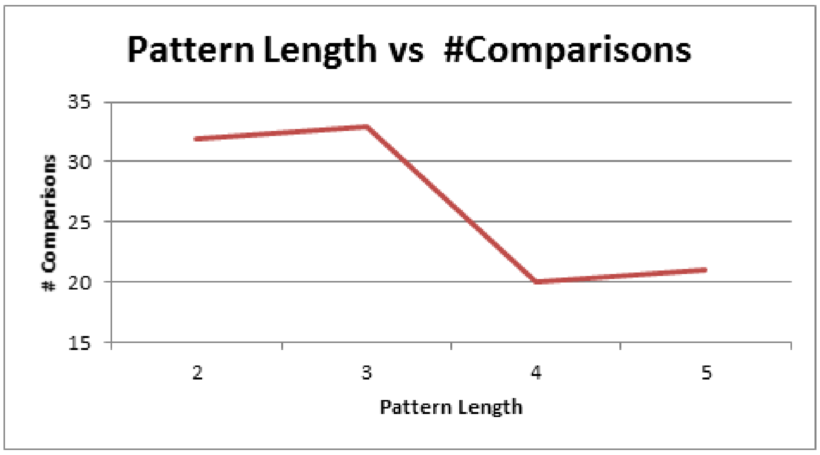

Fig.3.Recursive_LCS_Pattern algorithm runtime results for time

The Recursive_LCS_Pattern algorithm runtime graphical results show that the as the pattern length increases both memory and number of comparisons are reduced. Hence the Recursive_LCS_Pattern algorithm works efficiently for the LCS identification.

5. CONCLUSION

This paper proposes a recursive pattern matching algorithm called Recursive_LCS_Pattern to identify LCS. It recursively uses the existing length_2 pattern to identify existing length_3 pattern and so on. This process reduces the search spaces comparatively lower than enumerative combinatorics. This technique leads lesser number of pattern comparisons and less memory space which leads to linear time and space complexity to identify longest common patterns in DNA sequences.

REFERENCES

Agrawal R, Srikant R, 1995. Mining Sequential Patterns, ICDE'95, 1995, 3 - 14.

Bailey J, Segmental duplications: Orgnization and impact within the current human genome project assembly, Genome Res., 11, 2001, 1005-1017.

Berry T, Ravindran S, A Fast String Matching Algorithm and Experimental Results, Prague, Proceedings of the Prague Stringology Club Workshop, 2001.

Durbin R, Eddy S.R, Krogh A, Mitchison G, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, Cambridge University, 1998.

Hakata K, Imai H, Algorithms for the Longest Common Subsequence Problem for Multiple Longest Common Subsequence Problem for Multiple Strings Based on Geometric Maxima, Optimization Methods and Software, 10, 1998, 233 - 260.

Han J, Kamber M, Data Mining - Concepts and Techniques, 2nd ed., Sanfrancisco: Morgan Kaufmann Publishers, 2006.

Hirschberg D, A Linear Space Algorithm for Computing Maximal Common Subsequences, Comm. ACM, 18(6), 1975, 341 - 343.

Karim M, M.M, R.J, B.S.C, An efficient approach to Mining Maximal Contiguous Frequent Pattern from Large DNA Sequence Databases, Genomics & Informatics, 10(1), 2012, 51 - 57.

Mushegian A, Grand Challenges in Bioinformatics and Computational Biology, Frontiers in Genetics, 2011.

R.A.C, Le H, Ramachandran V, Efficient Cache-Oblivious String Algorithms for Bioinformatics, Austin: Dept. of Computer Science, Univ. of Texas at Austin, 2007.

Rashid M, Karim M, Jeong B, Choi H, Efficient Mining of Interesting Patterns in Large Biological sequences, Genomics & Informatics, 10(1), 2012, 44 - 50.

Suleiman D, Enhanced Berry Ravindran Pattern Matching Algorithm (EBR), Life Science Journal, 11(7), 2014, 395 - 402.

Timp W, Nanopore Sequencing: Electrical Measurements of the Code of Life, IEEE Transactions on Nano technology, 9(3), 2010, 281-294.

Timp W, Comer J, Aksimentiev A, DNA Base-Calling from a Nanopore Using a Viterbi Algorithm, Biophysical Journal, 102, 2012, 37 - 39.

Wang Q, Korkin D, Shang Y, A Fast multiple longest common subsequence (MLCS) Algorithm, IEEE transactions on knowledge and data engineering, 23(3), 2011.