DESIGN OF A NEW DETERMINISTIC ALGORITHM FOR FINDING COMMON DNA SUBSEQUENCE

Bigyan Bhar[1], Nabendu Chaki[2]

[1]Department of Computer Science & Engineering, University Calcutta, India
bigyan.bhar@gmail.com
[2]Department of Computer Science & Engineering, University Calcutta, India
nabendu@ieee.org

ABSTRACT

Computational methods have become especially important since the advent of genome projects, whose objective is to decode the entire DNA sequence. Sequence motifs are short, recurring patterns in DNA that are presumed to have a biological function. These motifs are often responsible for similarity or dissimilarity in biological features and their DNA patterns. In this paper, we start with a database containing the set of DNA patterns. Our aim is to search motifs that occur in the same order where as other characters or gaps might occur between the motifs. We generalize it as a common sub-sequence problem from the computational aspect. The exponential nature of the problem is rooted in its definition in the sense that the solution set itself is expected to be exponential in size. In this work, a new deterministic subsequence matching algorithm for a set of DNA strings has been proposed from the computational perspective. The proposed deterministic algorithm yields the exhaustive set of common sub-sequences that many of its commercial counterparts cannot guarantee.

KEYWORDS

Motif identification, sub sequence, Genome, Dynamic programming, NP Hard problem.

1. INTRODUCTION

Emergence of molecular genetics has changed the look of biology. Among the most exciting advances is large-scale DNA sequencing efforts such as the Human Genome Project [1], [2]. These are producing an enormous amount of data [3]. It is estimated that the volume of DNA and protein sequence data is currently doubling every 22 months [4]. Analysis of this huge amount of data is the one of the most challenging and innovative field of research. Demands for sophisticated analyses of biological sequences are driving forward the newly-created and explosively expanding research area of computational molecular biology, or bioinformatics. The collections of biological data are subject to organize, classify, relate, and mine the selected patterns from the whole data set. DNA structures are made experimentally in biology, but these structures are stored in one dimensional pattern form of large alphabetical sequences to make them suitable for mining and computing purpose. This provides strong motivation for developing computational methods that can infer biological information from sequence alone. Computational methods have become especially important since the advent of genome projects, whose objective is to decode this entire DNA sequence and to find the location and ordering of genetic markers along the length of the chromosome. These genetic markers can be used, for example, to trace the inheritance of chromosomes in families and thereby to find disease genes.

Basically DNA sequencing involves the collection of 4 peaks (A, G, C, or T). DNA and motifs are generally long character strings, where character represents amino acid sequence. Sequence motifs are short, recurring patterns in DNA that are presumed to have a biological function. Often they indicate sequence specific bindings sites for proteins such as nucleuses and

transcription features (TF). These motifs which are amino acid sequence patterns have their importance for biological significance. These sequences are responsible for similarity or dissimilarity in biological features and their DNA patterns. At this point, let's define a couple of basic terms that have been used frequently in the context of subsequence computation throughout the article.

A **subsequence** S_1 is a sequence that can be derived from another sequence S by deleting some elements from S without changing the order of the elements, e.g., BCDF is a subsequence of ABCDEFGH.

When a sequence S_1 is a subsequence of another sequence S, then sequence S is called the **super-sequence** of sequence S_1, e.g., ABCDEFGH is a super-sequence of BCDF.

Two sequences S_1 and S_2 are said to be **independent sequences** with respect to each other, iff S_1 is not a subsequence of S_2 and S_2 is not a subsequence of S_1. e.g., BCDF and ACFG are independent sequences.

Several Sophisticated statistical and machine-training techniques have been used in more recent protein structure prediction programs, and the success rate has increased [4, 5, 9]. A recent advance in this now active field of research is to organize proteins into groups or families on the basis of sequence similarity, and to find consensus patterns of amino acid domains characteristic of these families using the statistical methods [4]. However, computational biology has still a huge need for finding out the "sequence similarity" between DNA sequences.

The longest common subsequence (LCS) [8] solution is there to find the longest subsequence common to all sequences in a set of sequences (often just two). However, the solution inherently has higher complexity, as the number of sub-sequences is exponential in the worst case, even for only two input strings. There is another technique of using bitwise operators for the computation of the longest common subsequence which would have resulted in a "machine word length" times improvement [10] in speed. However, the increase in speed does not improve the time complexity for the solution.

At first glance, deciding that two biological sequences are similar and no different from deciding that two text strings are similar. However, simple substring finding algorithms fall short of accommodating the kind of flexibility in pattern matching required for biological sequences. To meet this need several algorithms have been discovered, most important class of those are called "edit distances" [6] first introduced by V. I. Levenshtein in 1966. In 1970 the Needleman–Wunsch algorithm [7] was developed which performs a global alignment on two sequences. It is commonly used in bioinformatics to align protein or nucleotide sequences. The Needleman–Wunsch algorithm was the first application of dynamic programming to biological sequence comparison.

The objective of this paper is *to design, implement and analyze the performance of a new deterministic algorithm that outputs the set of all sub-sequences of length above a predefined threshold value that are common to each string in a set of n DNA strings.*

We start with a database containing the set of DNA patterns and a collection of motifs without going into the details of DNA, motif structures and their characteristics. Our aim is to search motifs that occur in the same order (sequence) but not necessarily in contiguous manner in a desired number of the DNA patterns. In other words, the sequence of occurrence of motifs is same but there might occur other characters or gaps between two motifs. We generalize it as a common sub-sequence problem from the computational aspect.

However, let us understand that stated in this form, the solution set can be exponential in nature. So any attempt at solving this problem deterministically cannot have a polynomial time complexity. In fact, the problem is inherently NP-Hard [14] in nature. In case of an arbitrary number of input sequences, the problem of finding common sub-sequences is NP-hard. Thus most of the commercial tools available for use by molecular biologists use non-deterministic approaches. This in effect means that the proposed deterministic algorithm gives the set of exhaustive common sub-sequences that many of its commercial counterparts cannot guarantee.

2. LONGEST COMMON SUBSEQUENCE FOR TWO STRINGS

We have used the existing techniques for finding Longest Common Sub-sequece (LCS) as a building block for the proposed solution. In this section, the LCS methodology is presented for the sake of completeness. The two strings LCS problem can be broken down into smaller, overlapping sub-problems and these smaller problems may again be broken and so on. This may be solved using dynamic programming. The method is illustrated by the following algorithm:

```
Function LCSLength (X[1..m], Y[1..n])
Begin
    C = array(0..m, 0..n)
    for i := 0..m C[i,0] = 0;
    for j := 0..n  C[0,j] = 0;
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1;
            else
                C[i,j] := max(C[i,j-1], C[i-1,j]);
            endif
    return C[m, n];
End
```

The function LCSLength (X[1…m], Y[1…n]) computes the **subsequence matrix** C[0…m][0…n] and returns the length of the longest common subsequence that appears in cell C[m][n]. The complexity as evident from the algorithm is O(mn). Next we have the function that can read out only one instance of the longest common substring.

```
Function backTrace(C[0..m, 0..n], X[1..m], Y[1..n], i, j)
Begin
    if i = 0 or j = 0
        return();
    else
        if X[i-1] = Y[j-1]
            return backTrace(C, X, Y, i-1, j-1) + X[i-1];
        else
            if C[i,j-1] > C[i-1,j]
                return backTrace(C, X, Y, i, j-1);
            else
                return backTrace(C, X, Y, i-1, j);
            endif
        endif
    endif
End
```

This backTrace function is called with the arguments i=m and j=n initially to obtain one instance of the longest common subsequence. The complexity is Θ(k), where k = max(m, n).

```
Function backTraceAll(C[0..m, 0..n], X[1..m], Y[1..n], i, j)
Begin
    if i = 0 or j = 0
        return;
    else
        if X[i] = Y[j]
            return (Z+X[i] for all Z in backTraceAll(C,X,Y,i-1,j-1));
        else
            R := {};
            if C[i,j-1] ≥ C[i-1,j]
                R := backTraceAll(C, X, Y, i, j-1);
            endif
            if C[i-1,j] ≥ C[i,j-1]
                R := R U backTraceAll(C, X, Y, i-1, j);
            endif
        endif
    endif
    return R;
End
```

The backTraceAll function reads out all the longest common subsequences between strings X and Y, if more than one of them exists. The complexity is of the exponential order. The algorithm that we have developed for finding out All Common Substrings of two strings is modeled on backTraceAll.

		2	1	7	3	5	4	8	9	3	6
	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1
3	0	0	1	1	2	2	2	2	2	2	2
5	0	0	1	1	2	3	3	3	3	3	3
2	0	1	1	1	2	3	3	3	3	3	3
4	0	1	1	1	2	3	4	4	4	4	4
3	0	1	1	1	2	3	4	4	4	5	5
6	0	1	1	1	2	3	4	4	4	5	6
7	0	1	1	2	2	3	4	4	4	5	6
8	0	1	1	2	2	3	4	5	5	5	6
9	0	1	1	2	2	3	4	5	6	6	6

Figure 1: Matrix C[][] for longest common subsequence operation

The following is an example of the running of Longest Common Subsequence algorithm on strings: S_1 = 2173548936 and S_2 = 1352436789. In these strings, each digit represents a different alphabet. The matrix C[][] generated by LCSLength is as follows with the paths for backTraceAll.

In this case the longest common subsequences are not unique and thus backTraceAll would be

needed to generate the entire set of solutions. Figure 2 illustrates the two solutions. Our work on finding all the common sub-sequences of two strings is largely based on this algorithm. In fact we make use of the matrix C[][] from this algorithm and then proceed to build upon that.

		2	1	7	3	5	4	8	9	3	6
	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1
3	0	0	1	1	2	2	2	2	2	2	2
5	0	0	1	1	2	3	3	3	3	3	3
2	0	1	1	1	2	3	3	3	3	3	3
4	0	1	1	1	2	3	4	4	4	4	4
3	0	1	1	1	2	3	4	4	4	5	5
6	0	1	1	1	2	3	4	4	4	5	6
7	0	1	1	2	2	3	4	4	4	5	6
8	0	1	1	2	2	3	4	5	5	5	6
9	0	1	1	2	2	3	4	5	6	6	6

Figure 2: Longest Common Subsequences found after running backTraceAll

Another result that we would use in the proposed solution is that from the longest increasing subsequence problem. This has been defined as to find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous.

For example, in the sequence: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, a longest increasing subsequence is 0, 2, 6, 9, 13, 15. This subsequence has length six. However, the longest increasing subsequence in this example is not unique, as illustrated by 0, 4, 6, 9, 11, 15 which is another increasing subsequence of equal length in the same input sequence.

The solution may be obtained by computing the longest common subsequence between the original string and the string of the same alphabets sorted in non-decreasing order. The longest increasing subsequence problem is solvable in time O(nlog(n)) [11], where n is the length of the input sequence. The longest increasing subsequence problem is closely related to the longest common subsequence problem, which has a quadratic time dynamic programming solution.

3. The Proposed Algorithm for Generating All Common Sub-sequences for Any Number of Strings

We use a divide and conquer technique to handle this problem. Let S denote the input set strings and cardinality of S be n. We divide this set into ⌈n/2⌉ sets, each containing a pair of strings (in case n is odd then the last set would contain a single string).

Next we compute the set of independent super-sequences that contain every common subsequence present in the two strings for each such set. In case of a single string, we just copy the string. Next we treat each such independent super-sequence as strings and compute the independent super- sequences across such sets in a hierarchical fashion until we arrive at the final set. Assuming that we know the value of the minimum length of sub-sequences to find, we can ignore any subsequence of length less than the threshold value. The sub-sequences of length greater than or equal to that of the threshold value are automatically promoted to the next level iff the set does not contain a super-sequence of it.

Lastly we use a lexical order subset generation algorithm [12, 13] to compute and enumerate all the common sub-sequences appearing in all strings of the set above the threshold value. This last phase of the process is exponential in nature owing to the generation of combinations.

3.1. Algorithm for generating Independent Super-sequences

We begin with the Longest Common Subsequence matrix described in section 2. From this matrix we compute the independent sub-sequences starting from the rightmost column and bottommost row. We start from all such points where the last letter of either sequence match with letters of the other sequence. The procedure for reading out the independent super-sequences is different from the longest common subsequence algorithm.

```
Input:      String   str1, str2

Output:     Set <String> Super-sequences

Variables:  String   s1, s2  /* to store normalized form of str1 and str2 */
            Matrix <char>  M  /* string the LCS character matrix */

Begin
   (s1, s2) = normalize (str1, str2)
      /* s1 and s2 will be computed from str1 and str2 respectively by removing those
      characters that occur in only in any one string. */
   M = compute_LCS_matrix ( s1, s2 );
      /* the LCS matrix will be computed and stored in M */
   Scan the matrix M row-wise, and for each match in position (i, j);
   begin
      Create a vertex that contains the following information:
         int x=i, y=j;
            /* position of the match */
         Set <vertex> link = Φ
            /* set of vertices having a directed edge from the present vertex */
         char ch;
            /* character which caused the match */
         cutoff = 0;
      For each column k (0 <k <j) starting from k=j-1 to k=1
      begin
         Identify the lowest jump at position (l, k) (if it exists) in the column k, such that
         cutoff < l < i and no jump in that column is placed in the range between (l+1, k) and
         (i-1, k);
         Find out the vertex V at with x=l and y=k and put it in the vertex set link;
         cutoff = l;
      end
   end

   /* Now we have obtained a Directed Acyclic Graph (DAG) D in which the vertices
   containing the matches positioned at the rightmost column or bottommost row have 0 in-
   degree */

   S = Φ;
   for all vertices a: a ε D Λ In-degree(a)=0
      traverse the directed path P from a to b: b ϵ D Λ Out-degree(b)=0;
      derive a super-sequence by reversing the directed path P;
```

/* From each vertex in DAG with in-degree 0, traverse the DAG from that vertex to all the reachable vertices of out-degree 0. Each such path from a start to an end vertex will trace out one single super-sequence in reverse order. */
end.

3.2. Illustrative Example

		2	1	7	3	5	4	8	9	3	6
	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1
3	0	0	1	1	2	2	2	2	2	2	2
5	0	0	1	1	2	3	3	3	3	3	3
2	0	1	1	1	2	3	3	3	3	3	3
4	0	1	1	1	2	3	4	4	4	4	4
3	0	1	1	1	2	3	4	4	4	5	5
6	0	1	1	1	2	3	4	4	4	5	6
7	0	1	1	2	2	3	4	4	4	5	6
8	0	1	1	2	2	3	4	5	5	5	6
9	0	1	1	2	2	3	4	5	6	6	6

Figure 3: Matrix C[][] with the matches encircled

To demonstrate the aforementioned algorithm for generating independent super-sequences, we go back to the original example in section 2 for S_1 = 2173548936 and S_2 = 1352436789. Let the threshold value be 4, any subsequence of length less than that is not required.

In Figure 3, the circled numbers are the valid jumps, which indicate the positions where the vertical and horizontal strings has a letter in common, The value of such a cell indicates the length of the longest common substring that can be obtained from that character to the first character of respective strings.

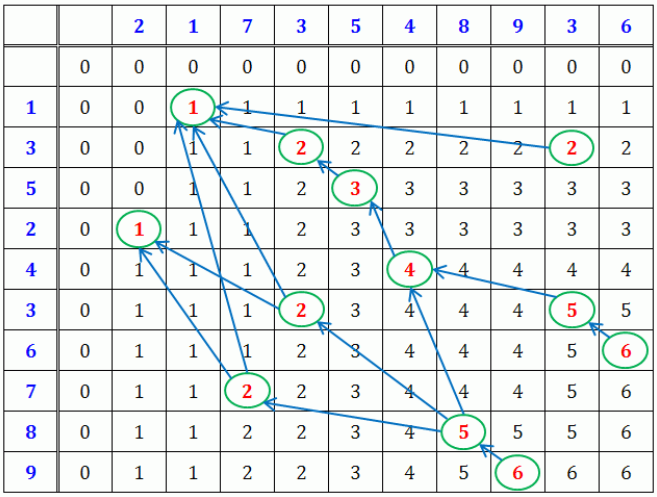

Figure 4: DAG drawn on the matrix C[][]

Figure 4 shows the Directed Acyclic Graph (DAG), drawn on the matrix C[][] itself, connecting all the valid jumps to one another. Figure 5 shows the same DAG, drawn in a different way to

aid the comprehension of the underlying meaning. In fact, this is the DAG that we will be using for our computation of the super-sequences.

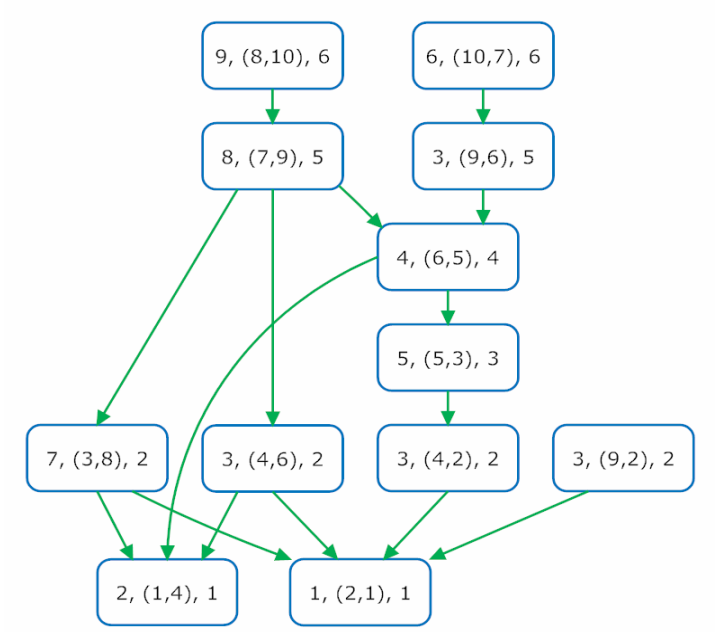

Figure 5: Directed Acyclic Graph from the C[][] matrix

Using the DAG, we can now compute the super sequences. Each path from a source node (in degree = 0) to a destination path (out degree = 0) represents a unique super-sequence. The set of all super-sequences obtained from the DAG contains all the sub-sequences common to strings S_1 and S_2.

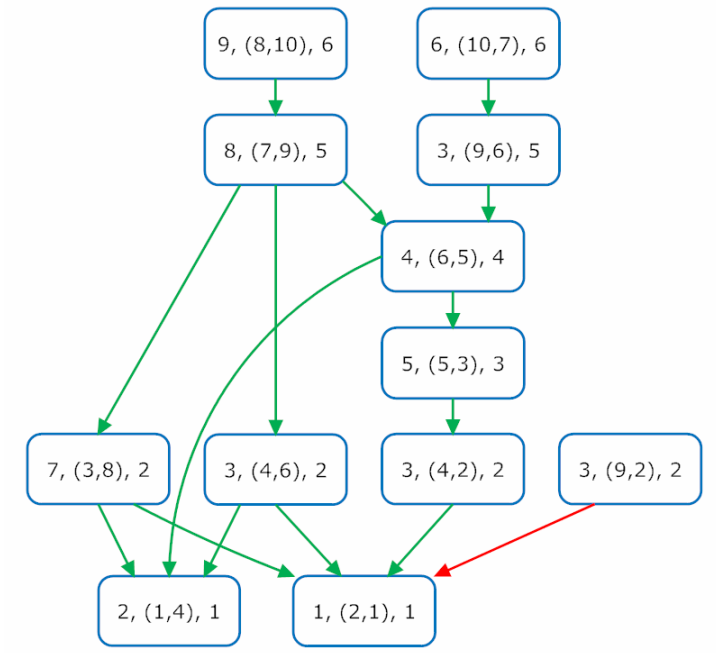

Figure 6: Final DAG (red paths are invalid)

As we know that the threshold value is 4, thus we can edit the DAG to contain only those paths of length ≥ 4. So we get a DAG represented in Figure 6. In fact while generating the DAG, this is automatically done. However this does not affect the complexity of the algorithm.

From this final DAG in Figure 6, we obtain the set of super-sequences by tracing out the alphabets in reverse order to which they occur in any path between a source and destination node: 135489, 135436, 2789, 2389, 2489, 2436, 1789, and 1389. It may be noted that we have a total of four sequences of length 4 here and two of length 6. So the total number of common sub-sequences between the strings S_1 and S_2 would be:

To get this entire set, the exhaustive set of all possible sub-sequences of length greater than or equal to the threshold value are to be generated from the super-sequences obtained so far. However, we do this only for the final step of the hierarchy and not at any lower levels to substantially reduce the amount of computation.

4. Proof of Algorithm for Generating Super-sequences

			S_2	
			ch[1]	ch[2]
		0	0	0
S_1	ch[1]	0	0/1	0/1
	ch[2]	0	0/1	0/1/2

Figure 7: Possible values in a 2x2 subsequence matrix

We will make use of induction to prove the correctness of our algorithm. First we will establish an induction base by proving that our algorithm works for all possible 2x2 subsequence matrices. To do this, we will have to first look into the different values that can constitute the 2x2 subsequence matrix [figure 7].

So we see that there are three cells with two possible values and one cell with one three possible values. Thus if we generate the exhaustive listing, there will be $2^3 \times 3^1 = 24$ such matrices. Fortunately, many of them are invalid in nature like the one shown in figure 8.

X			
	0	0	0
	0	1	1
	0	1	0

Figure 8: Invalid configuration of a 2x2 subsequence matrix

In total there are only 9 valid combinations of values for the 2x2 subsequence matrix. They are enlisted in figure 9 along with sample strings (of length 2) that cause them to appear. From the patterns, it is obvious that our algorithm works for all of them. This entire set of valid 2x2 subsequence matrices constitutes the induction base.

The induction hypothesis n this case is "Assuming that the algorithm works for any matrix of

size nxm, it also works for matrices of size (n+1)xm". In other words it means that "if the algorithm works correctly for strings S_1 and S_2 then it will also work for pairs S'_1, S_2 and S_1, S'_2 where S'_1, S'_2 represent the original strings with one character appended at the end.

1		2	2			**4**		2	3			**7**		2	1
	0	0	0				0	0	0				0	0	0
1	0	0	0			1	0	0	0			1	0	0	1
1	0	0	0			2	0	1	0			2	0	1	1
2		3	1			**5**		2	2			**8**		1	3
	0	0	0				0	0	0				0	0	0
1	0	0	1			1	0	0	0			1	0	1	1
2	0	0	0			2	0	1	1			2	0	1	1
3		3	2			**6**		2	1			**9**		1	2
	0	0	0				0	0	0				0	0	0
1	0	0	0			1	0	0	1			1	0	1	1
2	0	0	1			1	0	0	1			2	0	1	2

Figure 9: Valid cell values for 2x2 subsequence matrix

Without loss of generality let us assume that S_1 is the vertical string and S_2 is the horizontal one in the subsequence matrix. Let us further assume that only S_2 is appended by one character, making it S'_2. So the new matrix is same as the old matrix, with only a single new column added to the right side of it (figure 10).

			String S'_2					
			[1]	[2]	...	...	[m]	[m+1]
		0	0	0	0	0	0	0
String S_1	[1]	0	...	...	...	...	...	...
	[2]	0	...	...	...	...	...	...
	...	0	...	...	...	...	...	...
	...	0	...	...	...	...	...	...
	[n]	0	...	...	...	...	...	...

Figure 10: Modified subsequence matrix (grey cells are newly added)

As we are normalizing the strings before processing, there is bound to be some valid jump in the n^{th} row of the old matrix (non grey part). In the DAG for the new matrix, these will be the source nodes. For all other valid jumps that are present below (i.e. having higher y value that) the lowest valid jump in $(m+1)^{th}$ column would no longer be source nodes. Instead they will become the child nodes of the new valid jumps in column (m+1).

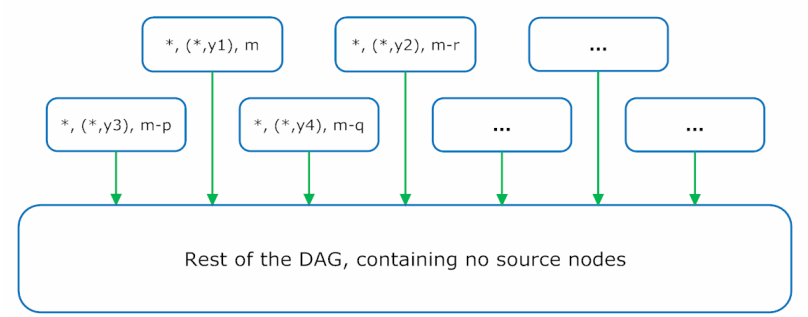

Figure 11: A generic DAG from nxm subsequence matrix

Thus, basically all the source nodes in the old DAG which had y values lower than the highest y value of the valid jumps in the $(m+1)^{th}$ column will become child nodes of the valid jumps in $(m+1)^{th}$ column. Rest of the nodes will stay in the same status as they were in the old DAG.

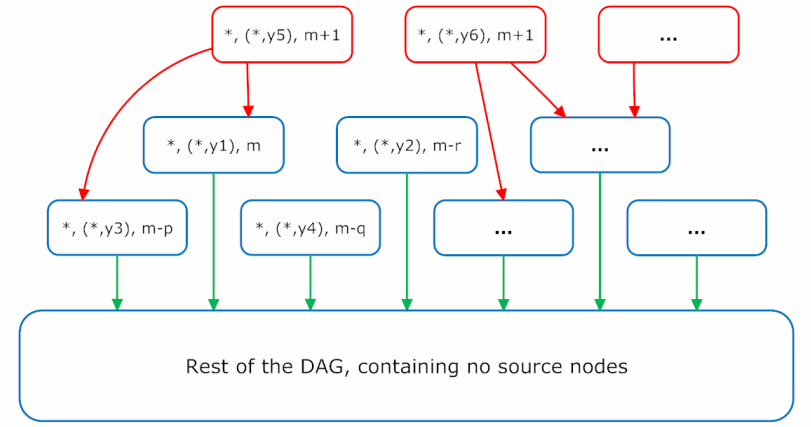

Figure 12: A generic DAG from nx(m+1) subsequence matrix

Figure 11 shows the old DAG and Figure 12 shows the newly constructed DAG as a modification of the old DAG. The new parts are colored red. Thus, it is established that source vertices of the old DAG will have either of the two outcomes in the newly constructed DAG:

Case 1: Continue to be source nodes

Case 2: Become the child nodes of the valid jumps appearing in column (m+1)

So the super-sequences that existed from the old source vertices are all preserved either in the same form (as in case 1) or as sub-sequences (as in case 2). Now what remains to be proved is that our DAG also contains the exhaustive set of sub-sequences that have been generated due to the inclusion of the alphabet appended at the end of string S_2.

Showing the validity of the sub-sequences is easy. We look into figure 12 in the parts drawn in red. The new nodes correspond to the valid jumps in $(m+1)^{th}$ column i.e. they correspond to the last character appended at the end of string S_2. If P be such a super-sequence and let C be the last character in S'_2 then the sequence Q, defined by P = concatenation (Q ,C), is a valid super-sequence for the strings S_1 and S_2. The character C is placed after the position where P ends in strings S_1. That is why there was a match of that character with the $(m+1)^{th}$ character of

string

S_2. Hence we prove that P is indeed a valid super-sequence of strings S_1 and S'_2.

To prove that our DAG contains the exhaustive set of sub-sequences for strings S_1 and S'_2, let us assume that there exist such a subsequence R. There can now be two cases:

Case 1: R is be a subsequence of strings S_1 and S_2.

Case 2: R is be a subsequence of strings S_1 and S'_2 and not of strings S_1 and S_2.

Case 1 cannot be true as we have already shown that all the sub-sequences of R must be a subsequence of strings S_1 and S_2 are also contained in the DAG. Thus only case 2 can be true; this implies that R ends with the last character of S_2. This again cannot be true, as all the sub-sequences ending at the last character of S_2 will end at either of the newly inserted nodes. Hence we prove that such a sub-sequence R cannot exist.

Thus we have proved the induction hypothesis to be true and so using the law of induction, it follows that our algorithm is correct.

5. Time Complexity of the Proposed Solution

5.1. Time Complexity for Generating Super-sequences

We can break down the algorithm into the following stages:

Stage 1: Generating the subsequence matrix

Stage 2: Computing the Directed Acyclic Graph

Stage 3: Traversing the DAG for super-sequences

Stage 1 has a complexity of $\Theta(mxn)$ as we have to generate a subsequence matrix of size mxn and computation of value for each cell takes a constant amount of time. Stage 2 and 3 has the same complexity of O(p), where $p = O(k(k-1)(k-2) \ldots (k-m+1)) = O({}^{k}P_m)$, where m is length of longest common subsequence and k is minimum of the lengths of S_1 and S_2 . The reason for this is evident from the following observation: in top level of DAG, we can have at most k number of children. In next level nodes can have (k-1) children and so on till level m, which is the stopping point.

However, the worst case complexity is O(k!) where k = min(m, n). This is derived from the fact that a node at position (x, y) can have at most min(x-1, y-1) number of child nodes. Thus the worst case complexity of the proposed algorithm is O(k!) where k = min(m, n). This case occurs when the subsequence has no gap and degenerates to substring finding problem.

5.2 Time Complexity of the entire Algorithm

A problem of this nature which ultimately involves k-subset generation is inherently NP-Hard [11] and resultantly there cannot be any polynomial time complexity algorithm for the solution. The last stage of the problem involves the generation of k-subsets where k = {t, t+1, t+2 … L}, where t is the threshold value and L is the length of the largest super-sequence produced. Any such subset generation problem has a complexity of exponential order.

Leaving out the last part, if we want to estimate the complexity of the part of generating super-sequences hierarchically, then the complexity for each problem is O(k!) k = minimum of the length of two sequences. If the initial number of strings were N, then the number of stages

would be $\log_2 N$. Thus the worst case complexity of the algorithm is

$$\left(\cdots \text{upto } \log_2 N \text{ terms} \right)$$

$$= O(n!).2N.\left[1 - \left(\frac{1}{2}\right)^{\log_2 N}\right]$$

(where n is the maximum length of the input strings). Clearly the complexity is exponential is nature.

6. CONCLUSIONS

The exponential nature of the problem is rooted in its definition in the sense that the solution set itself is expected to be exponential in size. Thus, any attempt at solving this problem deterministically cannot have a polynomial time complexity. However, the proposed deterministic algorithm gives the set of exhaustive common sub-sequences that many of its commercial counterparts cannot guarantee.

Although the all common independent super-sequence generation algorithm proposed by us has a worst case exponential time complexity, but if the threshold value (t) and the length of the largest common subsequence (c) are sufficiently close then the complexity is a polynomial of O(c-t). Thus this algorithm has a practical scope of application where the value (c-t) is small.

A major advantage of the proposed algorithm is that it is inherently parallel in nature. In contrast to some of the existing dynamic programming based solution, the proposed algorithm works on a divide and conquer mechanism. Breaking down the original problem into sub-problems of similar nature enables us to use a parallel architecture to achieve a significant time gain. The granularity of the sub-problem is quite low to ensure a sufficient time improvement during parallel implementation.

ACKNOWLEDGEMENTS

The authors acknowledge and appreciate *Dr. Sudip Kundu, Department of Bio Physics and Molecular Biology, University of Calcutta, India* for his contribution and advices in formation of the problem and also in understanding its significance. We also thank Ms. Devlina Das, a fresh Graduate of the *Department of Computer Science & Engineering, University of Calcutta, India* for her constant support in working with the student co-author of this article Bigyan Bhar during their final year project in closely related fields.

REFERENCES

[1] Necla Grant Cooper; The human genome project: Deciphering the blueprint of heredity; University Science Press, 1994. ISBN 0-935702-29-6

[2] T. J. Hudson, E. S. Lander, et. al.; An STS based Map of the Human Genome. Science, Vol. 270. no. 5244, pp. 1945 - 1954, 1995. ISSN 0036-8075.

[3] R. Durbin, S. Eddy, A. Krogh, G. Mitchison; Biological sequence analysis - Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, 2002. ISBN: 0521629713.

[4] Miller W, Scbwartz S, Hardison RC. A point of contact between computer science and molecular biology. IEEE Computational Science & Engineering. Vol. 1(1). pp. 69-78, 1994.

[5] N. Horspool. Practical Fast Searching in Strings. Software Practice and Experience, Vol 10(6), pp. 501 - 506, 1980.

[6] Vladimir I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, English translation in Soviet Physics Doklady, Vol. 10(8), pp:707-710, 1966

[7] Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, Vol. 48 (3), pp: 443-53, 1970.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms (second ed.). MIT Press and McGraw-Hill. pp. 350–355. ISBN 0-262-53196-8.

[9] Hirschberg, D. S. A linear space algorithm for computing maximal common subsequences. Communications of the ACM Vol. 18 (6), pp: 341–343. 1975.

[10] L. Allison, T.I. Dix, A Bit-String Longest-Common-Subsequence Algorithm. Information Processing Letter., Vol.23, pp.305-310, 1986.

[11] Schensted, C. Longest Increasing and Decreasing Subsequences, Canadian Journal of Mathematics 13: 179–191, 1961, ISSN 0008-414X

[12] Reingold E. M., Nievergelt J., Deo N., Combinatorial Algorithms. Prentice-Hall, Inc., New Jersey, 1977.

[13] Donald. L. Kreher, Douglas. R. Stinson, Combinatorial Algorithms: Generation, Enumeration and Search, CRC Press LTC, Boca Raton, Florida, 1998.

[14] Garey M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NP completeness, W. H. Freeman, 1979

Authors

Bigyan Bhar is a presently a Masters student at the Indian Institute of Science, Bangalore, India. He completed his B.Tech. in Computer Science and Engineering from University of Calcutta in 2009 and B.Sc. in Computer Science from St. Xavier's College in 2006. The current work is part of his B. Tech. dissertation work under the supervision of the other co-author of this article.

Nabendu Chaki is a faculty member in the Department of Computer Science & Engineering, University of Calcutta, Kolkata, India. He received his Ph.D. in 2000 from Jadavpur University, India. Dr. Chaki has published more than 50 referred research papers and a couple of text books. His areas of research interests include Distributed Systems, Software Engineering, and Bio Informatics. Dr. Chaki has also served as a Research Assistant Professor in the Ph.D. program in Software Engineering in U.S. Naval Postgraduate School, Monterey, CA. He is a visiting faculty member for many Universities including the University of Ca'Foscari, Venice, Italy. Besides being in the editorial board of a few International Journals, Dr. Chaki has also served in the committees of several international conferences.