A New Algorithm to Find Longest Common Sub-sequence's

Arindam Kotal[#], K.K.Senapati[*]

Abstract— In the study of pattern matching, string matching is an important topic and also an important component in the application of computer science. The Longest Common Subsequence (LCS) problem is to find a subsequence which is common to at least two or more given sequences. The sub- sequence which has the largest length is the longest common subsequence. This paper describes a new algorithm to find the longest common subsequence between two strings. In this paper we introduce two pruning operations which improve the speed up of the algorithm. The proposed algorithm emphasizes on the optimization of the running time & improvement of time complexity against the existing well-known improved dynamic programming algorithm for LCS. The further significance of this algorithm is that this algorithm can also be applied in multiple sequences. The comparison between the performance of the proposed algorithm with pruning and without pruning operations has also been analysed.

Keywords — Algorithm ,Dynamic programming, DNA Sequences, Longest Common Subsequence, Pruning Operations, Multiple sequence, String Matching.

IJSER

1 Introduction

Pattern matching or String matching is one of the oldest & fundamental areas in the application of computer science. Solution to the problem plays an important role in many areas of science & information processing. Sequence comparison is an important tool in molecular biology as it can be used to compare two or more given sequences. This application is particularly useful when studying the relationships of similar type of gene products. Any sequence can be represented as sequence of symbol over a fixed alphabet Σ. In biological sequence this sequence is a sequence of nucleotides. A protein sequence is a sequences of 20 characters (amino acid) & DNA sequence (gene) is combination of sequences of four characters Σ=(A,C,G,T) & RNA sequence is a combination of four characters Σ=(A,C,G,U) [15].

Let S & T be two sequences over some fixed alphabet Σ. The sequence T can be a subsequence of S or vice versa if T can be obtained by deleting some letters/characters/nucleotides from S, without violating the order of sequence of letters in S. The length of the subsequence is the number of letter in it. A longest common subsequence is a subsequence which has the longest length among the possible sub sequences & is the common one between two sequences. When a new biological sequence is discovered, this sequence has to be verified as a new sequence or homologous. It helps to find the exact DNA sequences matching, in retrieval of information like parenthood, genetic disorder, medicine, etc.

#Arindam Kotal is currently pursuing Masters' degree program in Computer Sc. &Eengineering in BIT Mesra, India, PH-08051124869. E-mail: arindam.kotal@gmail.com
**K.K.Senapati is an Assistant Professor in the depertment of Computer Sc. & Enginnring in BIT Mesra, India, PH-09431768348. E-mail: kksenpati@bitmesra.ac.in.*

Our paper is organized as follows. The section 2 discusses some related LCS algorithms. The new algorithm including the pruning steps followed by suitable example is explained in section 3. Then in section 4 describes the pseudo code of the proposed algorithm. The analysis & comparison part is given in section 5 and the paper is concluded with a discussion & future scope in section 6.

2 Related Works

For two DNA sequences, let X & Y each of length m and n. The popular algorithms are:

In 1974 Wagner and Fischer [1] published an algorithm which solves for Levenshtein distance with dynamic programming. The Levenshtein distance is an important concept to find out edit distance between two sequences. Levenshtein distance is a count of how many substitutions, insertions, and deletions are required to change one string to another string. To break down the Levenshtein distance problem, Wagner and Fischer used a matrix solution. Given two strings, place one string down the left side of the matrix, Place the other across the top of the matrix. The last cell of the matrix was cumulative distance between the two strings. It was also the length of the LCS between the two given sequences. This algorithm did not give the LCS. In this algorithm as each character of left string was needed to compare against each character of top string only once, the time complexity of the algorithm was O (mn).

In 1975 Hirschberg [2] employs a divide-and-conquer strategy, which consists of recursively breaking the problem into smaller independent sub problems, solving the sub problems directly and combining their solution to solve the whole problem. So, this model follow the divide conquer model. It was actually the combination of divide conquer & dynamic method. In the first phase it searches the matrix from the forward direction but in the second phase it searches the matrix from reverse direction. Although the space complexity of the algorithm is reduced to O (m) but the time complexity of the algorithm was O (mn). Later in 1977 Hirschberg [3] gives another approach based on the dominant matches. The complexity of this algorithm is O (rn+nlogn) where r is the total number of ordered pairs of positions at which the two strings match.

In 1977 J.W. Hunt and T.G. Szymanski's [4] approach to extracting an LCS from two strings is equivalent to determining the longest monotonically increasing path in the graph composed of nodes (i; j) such that $x_i = y_j$. Whereas previous methods required quadratic time in all cases, their algorithm requires O ((r + n) log n) time for equal length strings. A small amount of pre-processing vastly improved the performance of the algorithm. The main source of inefficiency in this algorithm is the inner loop on which repeatedly searches for the elements of the Y sequence which match X[i]. Link List techniques avoids this problem.

Naktasu ET all [5] gave quite different solution. In this case, the LCS is found by checking systematically, how long common subsequence can be found for Y and the substring X[C...i]. That is, starting from X[i], we select consecutive symbols from X and traverse Y from ri ght to left searching for matches until all of Y has been used up. These results are then combined cleverly to obtain the LCS. The search is terminated as soon as we know that the suffix of X [l...i] cannot give us any better results. Because of this, the method executes fast if the LCS is long; its theoretical time complexity is O (n (m - r**)).**But this algorithm is only suitable for similar texts.

In 1987 Apostolico, A. & Guerra, C.[7], They introduced an alternate date structures to support the forming of the LCS: close vector is a compact representation of the closest table defining for each symbol of ∑ its nearest occurrence in Y after a given position j. The time complexity of their algorithm is O (mlogn+dlog (2mn/d)) where d is the minimum distance of the next closest common elements.

In 1990 Wu ET all [8] minimized the edit distance problem to compressed edit distance. The compressed edit distance was sticker to the close main diagonal. Here instead of calculating the true edit distance, they are concentrated to reduce the number of deletion. Thus, the algorithm executes quickly when the shorter input string is a substring of the other. This algorithm has the time complexity of O (n (m-r)).

In 1995 Rick, C [9] published an algorithm which simple & efficient compared to the previous discussed algorithm. The algorithm was based on advancing from contour to contour. Contour is a region where the matching values are found & it was bounded by broken lines [10]. The algorithm is based on the well-known paradigm of computing dominant matches and was obtained through a kind of dualization. Its time complexity was O (min {(rm), r (n-r)}).

The most well-known algorithm to find LCS between two strings is the improved dynamic programming algorithm [13].

A DNA sequence is viewed as a linear sequence of $a_1, a_2, a_3 \ldots a_m$ of nucleotide. The sequence is known as primary structure. Each a_i is identified with one of the four nucleotides i.e. A, T, G, C. The LCS problem has an optimal substructure property.
Let, A= ($a_1, a_2, a_3 \ldots a_m$) and B= ($b_1, b_2, b_3 \ldots, b_n$) be sequences. Let Z= ($z_1, z_2, z_3 \ldots, z_i$) be any LCS of A & B. This property had the three following cases:

1. If $a_m = b_n$, then $z_i = a_m = b_n$ and Z_{i-1} is an LCS of A_{m-1} and B_{n-1}.
2. If $a_m \neq b_n$, then $z_i \neq a_m$ implies that Z is an LCS of A_{m-1} and B.
3. If $a_m \neq b_n$, then $z_i \neq b_n$ implies that Z is an LCS of A and B_{n-1}.

The optimal substructure of the LCS problem tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences.

From this optimal substructure property we can find an LCS of two sequences. If $a_m = b_n$, we must find an LCS of A_{m-1} and B_{n-1}. If $a_m \neq b_n$, we have to solve two sub problems: finding an LCS of A_{m-1} and B and finding an LCS of A and B_{n-1}. The longest LCS between this two sub problems is the LCS of two sequences.

Let us define C [i, j] to be the length of the LCS of the sequences A_i and B_j. If i=0 or j=0 i.e. one of the sequence has length 0, then, the LCS become zero length also. The optimal substructure gives the recursive formula:

$$C\,[i, j] = \begin{cases} 0 \quad \textit{if } i = 0 \textit{ or } j = 0 \\ C[i-1, j-1] + 1 \textit{ if } i, j > 0 \textit{ and } a_i = b_j \\ \max(C[i, j-1], C[i-1, j] \textit{ if } \quad i, j > 0 \textit{ and } a_i \neq b_j \end{cases}$$

3 Our method

All the methods mentioned above are based on matrix. Our method uses a heuristic approach for determining the LCS of two sequences. Our approach is to find out the relative positions of all the matching pairs. This is done in the preprocessing phase. Only once the proposed method is searched the two given DNA sequences & collect all the possible matching pairs. As all the matching pairs are not necessary to find the LCS, we delete some matching pairs by only identifying those pairs. This is done by the introduction of two pruning operations. The method can be easily understood theoretically & experimentally.

We define Σ as a sequence of four nucleotides Adenine (A), Thiamine (T), Guanine (G), and Cytosine(C).

The proposed method based on the following steps:

Step1: Find out all the possible matching pairs between the two sequences.

Step2: Among the matching pairs, choose those pairs which have contain the same elements. Prune the pairs which have repeated elements & the 1st element of the set should be greater than the 1st element of the other set.

Step3: Among the existing matching pairs find out the sum of each pairs & sorted them in increasing order. Sum is calculated as follows:

$X = S_1 + S_2$ Where S_1 and S_2 refers to the sequence being used.

Step4: Identify the pairs which are having the same sum, find out the difference between the elements of the same sum & select the pairs which are having minimum differences & prune others. The difference is calculated as follows:

$Y = | S_1 - S_2 |$

Step5: From the remaining matching pairs find out the pairs which are having the minimum sum and select the corresponding character as the character in the LCS of the two sequences.

Step6: Delete those pairs whose elements contain less than or equal values compared to the elements of selected matching pairs. This process continues until all the matching pairs are exhausted.

IJSER

Pruning Operations:

In the process of generating pruning technique, it can be implemented to remove those matching pairs which can't generate the LCS .It leads to reduce the search space & complexity & increase the efficiency.

Pruning Operation1: If on the same level, find out those matching pairs contained the duplicate elements, prone the character corresponding to those duplicate matching pairs whose first elements have greater value compared to the selected pair's first elements.

The reason for pruning character corresponds to duplicate matching pairs is as follows. Let the identical character set is $P_n = (s_1, s_2)$ and this set P_n is repeated as (s_2, s_1). So, (s_1, s_2) or (s_2, s_1) represents the same character. It is unnecessary to repeat the set which contained same elements. So, select one pair. The selection should be done by the comparison of the first elements. If $s_1 < s_2$ then select the character corresponding pair (s_1, s_2) & prune the others. If the process selects the other set then some matching pair's sets are deleted and as a result LCS is not coming. Because of this select the pair whose first element has less value.

Pruning Operation2: Identify those pairs which are having same sum, then select the character correspond to the pair which are having minimum differences between its elements. Prune the character correspond to the other pairs.

Working Example:

For example consider two input sequences:

	1	2	3	4	5	6	7	8	9
S1 =	T	G	C	A	T	A	A	T	T
S2 =	T	A	G	T	G	T	A	T	G

1. Step1: Matching Pairs: 🟥 **Deleted**

p[1]=(1,1)	p[9] =(4,7)	p[17]=(7,7)
p[2]=(1,4)	p[10]=(5,1)	**p[18]=(8,1)**
p[3]=(1,6)	p[11]=(5,4)	p[19]=(8,4)
p[4]=(1,8)	p[12]=(5,6)	p[20]=(8,6)
p[5]=(2,3)	p[13]=(5,8)	p[21]=(8,8)
p[6]=(2,5)	p[14]= (6,2)	p[22]=(9,1)
p[7]=(2,9)	p[15]=(6,7)	p[23]=(9,4)
p[8]=(4,2)	p[16]=(7,2)	p[24]=(9,6)
p [25] =(9,8)		

Step2:

Apply pruning operation1 and find that matching pair's p[4] and p[18] contained the same elements, prune pair p[18].

Step3:

p [1] = X = 1+1 = 2	p [13] = X = 5+8 = 13
p [2] = X = 1+4 = 5	p [14] = X = 6+2 = 8
p [3] = X = 1+6 = 7	p [15] = X = 6+7= 13
p [4] = X = 1+8 = 9	p [16] = X = 7+2 = 9
p [5] = X = 2+3 = 5	p [17] = X = 7+7 =14
p [6] = X = 2+5 = 7	p [18] = X = 8+4 =12
p [7] = X = 2+9 = 11	p [19] = X = 8+6 = 14
p [8] = X = 4+2 = 6	p [20] = X = 8+8 = 16
p [9] = X = 4+7 = 11	p [21] = X = 9+1 = 10
p [10] = X = 5+1 = 6	p [22] = X = 9+4 = 13
p [11] = X= 5+4 = 9	p [23] = X = 9+6 = 15
p [12] = X = 5+6= 11	p [24] = X = 9+8 = 17

Step4, 5, 6:

Here pruning operation2 is also applicable and find the difference between the elements of those pairs which have same sum.

p [3] and p [6] have same sum.

$Y_1 = |1-6| = 5$

$Y_2 = |2-5| = 3$

According to pruning operation2 p [3] is pruned.

p [7], p [9] and p [12] have same sum.

$Y_1 = |2-9| = 7$

$Y_2 = |4-7| = 3$

$Y_3 = |5-6| = 1$

p [7] and p [9] are pruned.

In the same way,
p [8] & p [10], p [10] is pruned.
p [2] & p [5], p [2] is pruned.
p [4], p [11] and p [16], p [4] & p [16] are pruned.
p [13], p [15] and p [22], p [13] & p [22] are pruned.
p [17] & p [19], p [19] is pruned.

Remaining Matching pairs:

p [1] = (1, 1)	p [9] = (7, 7)	🟦 minimum
p [2] = (2, 3)	p [10] = (8, 8)	
p [3] = (2, 5)	**p [11] = (9, 1)**	🟥 delete
p [4] = (4, 2)	p [12] = (9, 6)	
p [5] = (5, 4)	p [13] = (8, 4)	
p [6] = (5, 6)	p[14] = (9,8)	
p [7] = (6, 2)		
p [8] = (6, 7)		

Select the character correspondence to p[1] as its sum is minimum.

S_1 = T G C A T A A T T
S_2 = T A G T G T A T G
LCS= T……………..

As pair p [11] has already visited, that's why it is deleted.
This process is continued till the sequence is exhausted.

2. Remaining Matching Pairs:

	Sum		Sum
p [1] = (2, 3)	**5**	p [8] = (7, 7)	14
p [2] = (2, 5)	**7**	p [9] = (8, 8)	16
p [3] = (4, 2)	**6**	p [10] = (9, 6)	15
p [4] = (5, 4)	9	p [11] = (8, 4)	12
p [5] = (5, 6)	11	p [12] = (9, 8)	17
p [6] = (6, 2)	**8**		
p [7] = (6, 7)	13		

Here p [1] is selected as minimum pair which has minimum sum and select the character correspondence to it. p [2], p [3], p [6] are deleted as their elements are already visited.

S_1 = T G C A T A A T T
S_2 = T A G T G T A T G
LCS= T G…………..

3. **Remaining Matching Pairs:**

p [1] = (5, 4)	**9**
p [2] = (5, 6)	**11**
p [3] = (6, 7)	13
p [4] = (7, 7)	14
p [5] = (8, 8)	16
p [6] = (9, 6)	15
p [7] = (8, 4)	**12**
p [8] = (9, 8)	17

S_1 = T G C A T A A T T
S_2 = T A G T G T A T G
LCS= T G T…………..

4. **Remaining Matching Pairs:**

p [1] = (6, 7)	**13**
p [2] = (7, 7)	**14**
p [3] = (8, 8)	16
p [4] = (9, 6)	**15**
p [5] = (9, 8)	17

S_1 = T G C A T A A T T
S_2 = T A G T G T A T G
LCS= T G T A………..

5. **Remaining Matching Pairs:**

	Sum
p [1] = (8, 8)	**16**
p [2] = (9, 8)	**17**

S_1 = T G C A T A A T T
S_2 = T A G T G T A T G
LCS= T G T A T

Here p [1] is selected as minimum pairs and p [2] is deleted as its one of the elements is already visited. The character corresponding (8, 8) is selected as the character of LCS.

As the sequences are exhausted, the process is terminated.

Length of the LCS is 5.

4 The Algorithm

Pre-processing Phase:
FindMatchPairs (p [ ]) //let p is the set for all matching pairs
Input: I [1...m] & J [1…n] are two DNA sequences

```
1.  m← length(I);
2.  n← length(J);
3.  a=0;
4.  for i=0 to m-1
5.      for j=0 to n-1
6.          if (I[i]==J[j])
7.              P[a]=(I[i],J[j]);
8.              a++;
9.          end if
10.     end for
11. end for
12. return p[a];
```

Pruning Phase:
PruningOperation1 (p, a)
Input: The set of matching pairs i.e. p

```
1. r = 0, count = 0;                //initialize
2. for i=0 to a-1
3.   P[i] = FindMatchPairs (p[a]);
4.    for j=0 to r-1
5.      if ((I[i] == J[j]) && (J[i] == I[j]))
6.          break;
```

IJSER

```
7.      End for
8.    if (j==r)
9.      P[count] = (I[i], J[j]);
10.   end if
11. end for
12. return p [count];
```

PruningOperation2 (P, count)
Input: Remaining set of matching pairs i.e. p

```
1.  for i=0 to count-1
2.    X[i]= sum (I[i],J[i]);
3.  end for
4.  for i=0 to count-1
5.    if (X[i]== X[i+1])
6.      Y[i]= abs (I[i] – J[i]);
7.      Set $Y_{min}$= MIN(Y[i]);
8.    end if
9.  end for
10. return $Y_{min}$
```

Implementation Phase:

Compute LCS ()
Input: The set of remaining match pairs i.e. p [1, 2… n]

```
1. for i=0 to n-1
2.   X[i]= sum(I[i], J[i]);
3.   Set L[i] = MIN(X[i]);
4.   Select the pair corresponding L[i], say A[i], B[i]
5.   Remove the matching pairs.
     // which are <= relative position of A[i] & B[i]
6. End for
7. For i=0 to n-1
8. Store the character corresponding to A[i] & B[i] in an
   array says LCS[i]. //output
9. End for
```

5 Analysis and Result

The aim of the algorithm is to find out the LCS between two large sequences. In addition to this the algorithm also finds out LCS between multiple DNA sequenced with large length, even manually. This is possible because of taking the relative position of each character.
Analysis of the proposed algorithm is based on three phases.
In the pre-processing phase, it will take O (n^2) time to find the identical character sets from the two given DNA sequences. The subsequent phase which is known as the pruning phase also take O (n^2) time. The final phase of computing LCS will take O (n) time because the length of the matching pairs is n. The desired result is achieved by a linear time complexity, compared to a quadratic order complexity [13].

In terms of total time complexity without the distinction of pre-processing phase & pruning phase/operation the time complexity lies between O (n^2) to O (n^3) according to the best & worst case analysis.
The number of identical character set is reduced by the two pruning operations. Pruning techniques removed those Identical character sets which could not able to generate LCS. Thus the speed up is achieved and search space also reduced.

We implemented our algorithm in DEV-C++ 4.9.9.0 on a 2.67GHz processor with 3GB RAM on a windows7 operating system. The DNA sequences data's are taken from the FASTA format [16].

To find out the effect of pruning, we took the identical two sequences with increasing length and we got a decent amount of speedup using the pruning approach. The comparison has been shown in Fig. 1. We also compare our new algorithm with pruning with the improved dynamic programming [13]. The comparison has been shown in Fig. 2.

TABLE I
Pruning effect

Sequence length	Time with pruning	Time without pruning
10	0.001	0.093
20	0.002	0.218
30	0.004	0.359
40	0.005	0.50
50	0.007	0.765
60	0.010	0.967

TABLE II
Comparison with improved dynamic programming

Sequence length	Time with New Algorithm	Time with Improved Dynamic Programming
10	0.001	0.003
20	0.002	0.004
30	0.004	0.005
40	0.005	0.007
50	0.007	0.009
60	0.010	0.012

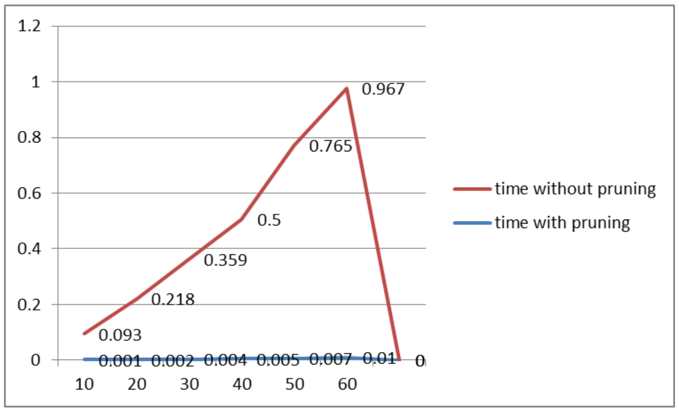

Fig. 1

IJSER

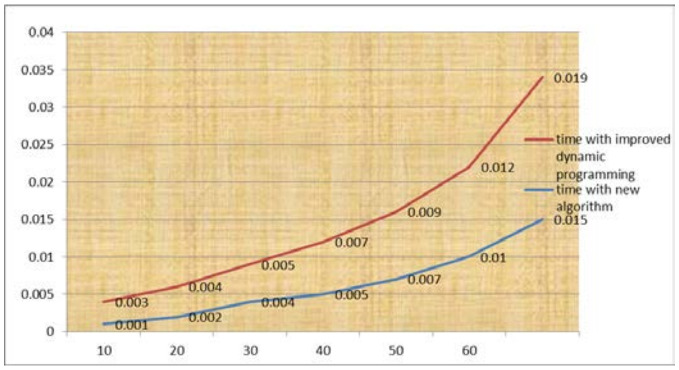

Fig. 2

6 Conclusions and Future Work

This paper has given a simple and novel approach for finding the LCS from sequences of DNA, protein etc. It has been examined that this algorithm has also been able to find LCS from multiple sequences of DNA, protein etc. We feel that there are still some scope to improve the time complexity and performance of our approach through different data structure such as heap for finding minimum sum.

IJSER

References

[1] R. A. Wagner and M. J. Fischer, The string to string correction problem, J.ACM, Vol.21, No. 1, 1974, pp.168-173.

[2] D. S. Hirschberg: A linear space algorithm for computing maximal common subsequences. Commun. ACM, 18(6):341–343, 1975.

[3] D.S. Hirschberg , Algorithms for the Longest Common Subsequence Problem, J.ACM, Vol.24, No.4, October1977, PP. 664-675.

[4] J.W. Hunt & T.G.Szymanski, A Fast Algorithm for Computing Longest Common Subsequences, Comm. ACM, Vol. 20, NO. 5, 1977, pp. 350-353.

[5] N. Nakatsu, Y. Kambayshi & S .Yajima : A Longest Common Subsequence Algorithm Suitable for Similar Texts, Acta Informatica, vol.18, 1982, pp.171-179.

[6] E.W. Myers: An O (ND), Difference Algorithm and its Variations, Algorithmica, Vol. 1. 1986, pp. 251-266.

[7] A.Apostolico & C.Guerra: The Longest Common Sub-sequence Problem Revisited, Algorithmica, No. 2, 1987, pp. 315-336.

[8] Wu, S., Manber, U., Myers, G. & Miller, W.: "An O (NP) Sequence Comparison Algorithm", Inf. Proc. Lett., Vol. 35, September 1990, pp. 317-323.

[9] C. Rick: A New Flexible Algorithm for the Longest Common Subsequence Problem, in Galil, Z. & Ukkonen, E. (eds): Proc. of Combinatorial Pattern Matching, 6th Annual Symposium, Espoo, Finland, July 1995, pp. 340-351. Appeared also as Lectures Notes in Computer Science, vol. 937.

[10] S .Kuo & G.R .Cross, An Improved Algorithm to Find the Length of the Longest Common Subsequence of two Strings, ACM SIGIR Forum, Vol. 23, No. 34, 1989, pp.89-99 .

[11] L. Bergroth, H. Hakonen, and T. Raita, A survey of longest common subsequence algorithms, Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, 39 – 48.

[12] SAM Rizvi, P. Agarwal, A New Index-Based Parallel Algorithm for finding Longest Common Subsequence in Multiple DNA Sequences. International Conference in Cognitive Systems, 2005.

[13] T. H. Corman, R L. Rivest, Charles E.Leiserson, C.Stein, "Dynamic Programming", Introduction to Algorithm, Third Edition, Cambridge, MA: MIT Press, 2010, Ch. 15, sec. 15.4, pp.390-397.

[14] D. Gusfield, "Core Strings Edits, Alignments, and Dynamic Programming", Algorithms on Strings, Trees, and sequences, Cambridge, Cambridge University Press, 1999, Ch. 11, pp. 215-295.

[15] P. A. Pevzner & N. C. Jones.: *An Introduction to Bioinformatics Algorithms*, MIT press, MA, 2004, pp.61-63.

[16] FASTA format - http://www.neb.com/nebecomm/tech_reference/