Application of the *A** Algorithm to Solve the Longest Common Subsequence from Fragments Problem

Yu-Mei Pan and Chia-Tung Lee*
Computer Science and Information Engineering Department
National Chi-Nan University, Nantou, Taiwan
ympan@realtek.com.tw, rctlee@ncnu.edu.tw*

Abstract

Finding longest common subsequence (LCS) is a common problem in Biology informatics. The problem is defined as follows: Given two strings $X=x_1x_2…x_m$ and $Y=y_1y_2…y_n$, find a common subsequence $L=l_1l_2…l_p$ of X and Y such that p is maximized. In this paper, we discuss a variation of the LCS problem – LCS from fragments problem defined as follows: Given two strings X and Y and a set M of fragments which are matching substrings of X and Y, find a LCS from M. A new method using a tree searching strategy, A algorithm, is proposed in this study for the LCS from fragments problem. The method can help us to filter out some fragments which wouldn't appear in solutions, and efficiently find a solution. However, in worst cases, all fragments are needed to be computed in the solving process.*

1 Introduction

In computational biology, there are many important problems such as sequence alignment problem [11, 12], multiple sequence alignment problem [3, 5], longest common subsequence (LCS) problem [6, 7, 8] and string matching problem [1, 4, 9] etc.

In this paper, we discuss a variation of the LCS problem. First, let us introduce the LCS problem defined as follows.

Def. 1-1 Longest common subsequence problem
Given two strings $X=x_1x_2…x_m$ and $Y=y_1y_2…y_n$, find a common subsequence $L=l_1l_2…l_p$ of X and Y such that p is maximized.

The LCS of two strings X and Y is denoted LCS(X, Y). Let |LCS(X, Y)| be the length of LCS(X, Y). For instance, given X=agcgtta and Y=gatgagt, LCS(X, Y) is "aggt" and |LCS(X, Y)|=4. Up to now, many algorithms [7, 8] have been proposed to solve the problem. [7] proposed a divide-and-conquer algorithm to solve the problem in linear space. [8] proposed a dynamic programming approach to solve the problem, and it introduced the sparse computing concept into the dynamic programming method.

We now introduce the LCS from fragments problem. First, given two strings X and Y, a fragment of X and Y is a common substring of X and Y. Given two strings X and Y and a set M of fragments $\{f_1, f_2, …, f_h\}$ of X and Y, a common subsequence CS from M is a common subsequence between X and Y such that CS contains only some fi's or substrings of fi's. An LCS of X and Y from M is one of the common subsequence of X and Y from M which is the longest. The LCS from fragments problem is defined as follows:

Def. 1-2 Longest common subsequence from fragments problem
Given two strings X and Y and a set M of fragments of X and Y, find a longest common subsequence of X and Y from M.

The LCS from fragments problem was first proposed in [2], and a dynamic programming approach was proposed to solve the problem. In the dynamic programming approach, it has to compute all fragments in finding the solution. But, not all fragments appear in the found solution. Due to the reason, a new method is proposed in this paper and the method can ignore the fragments which don't appear in solutions.

In the new method proposed in the paper, the concept of the A* algorithm [10] is used. The method can help us to ignore the fragments which will not appear in solutions, and it can efficiently find a solution for the LCS from fragments problem.

In the following sections, the score measure used in this paper will be introduced in section 2. The A* algorithm to solving the LCS from fragments problem will be discussed in section 3. Experimental results will be presented in section 4. Finally, in section 5, concluding statements will be presented.

2 The Edit Distance

Given two strings X and Y, the edit distance between X and Y is the minimum number of deletions, substitutions and insertions to transform X to Y. For example, let X=aat- -ct and Y=aatgag.

We may align the strings as follows:

$$X = \text{a a t - - c t}$$
$$Y = \text{a a t g a - g}$$

As can be seen, X can be transformed to Y by executing two insertions, one deletion and one substitution. It can also be proved that the edit distance between X and Y is 4.

It is well known that the edit distance finding problem is equivalent to the longest common subsequence finding problem.

2.1 The Edit Graph Shortest Path Problem

In this section, we are interested in the longest common subsequence from fragments problem. In this section, we shall show that this problem can be viewed as a graph searching problem. The graph is defined an edit graph and the problem is to find a shortest path from a certain starting node to a certain terminal node.

Let us illustrate the edit graph through an example. Suppose we are given X=tacat and Y=actat, the edit graph of X and Y is showed in Fig. 1. In the edit graph, for a node p, let $x(p)$ and $y(p)$ to be the x-coordinate and y-coordinate of p, respectively. For instance, for node (2, 3), $x((2, 3))=2$ and $y((2, 3))=3$.

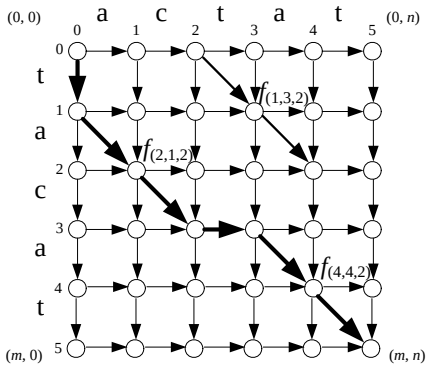

Fig. 1. The shortest path of two strings X=tacat and Y=actat and the set $M=\{f_{(2, 1, 2)}, f_{(1, 3, 2)}, f_{(4, 4, 2)}\}$.

Suppose we further have fragments ta, ac and at. Then we denote these fragments as diagonal paths. We denote a path by (i, j, k) where i, j and k stand for the starting positions in X and Y and the length of the matching substring, respectively. For instance, given two strings X=tacat and Y=actat and $M=\{f_{(2, 1, 2)}, f_{(1, 3, 2)}, f_{(4, 4, 2)}\}$, $f_{(2, 1, 2)}$, $f_{(1, 3, 2)}$ and $f_{(4, 4, 2)}$ represent the three common substrings ac, ta and at, of X and Y, respectively.

In addition, for a fragment f, the node (i-1, j-1) of f is denoted as $start(f)$ and the node (i+k-1, j+k-1) of f is denoted by $end(f)$. For example, consider Fig. 1. $start(f_{(2, 1, 2)})$ is the node (1, 0) and $start(f_{(4, 4, 2)})$ is the node (3, 3).

For each horizontal and vertical edge, we associate it with cost 1 and for each diagonal edge, we associate it with cost 0. Our longest common subsequence from fragments problem can now be viewed as a shortest path finding problem where the path is from (0, 0) to (m, n). In Fig. 1, the black path it the shortest path from (0, 0) to (5, 5).

After obtaining the shortest path, we obtain the solution by only retaining the substring in the fragments. There for the case shown above, the found solution is acat. Now, let us consider a situation. Would all fragments appear in the found shortest path? The answer is "Not necessarily". For example, consider Fig. 1, fragment $f_{(1, 3, 2)}$ doesn't appear in the found shortest path of X and Y and the set M. Thus we proposed a new method using A* algorithm to solve the problem. Through this method, we can filter out the fragments which wouldn't appear in the solution, and it can efficiently find the solution.

3 The *A** Algorithm

In this chapter, we shall introduce the A* algorithm for the shortest path finding problem. The edit distance is used to be the cost measure through this chapter. In the edit graph, for each horizontal and vertical edge, we associate it with cost 1 and for each diagonal edge, and we associate it with cost 0. In the following, we shall use one simple example to informally illustrate the basic idea of the A* algorithm.

Consider Fig. 1. It's the edit graph of two strings X=tacat and Y=actat and a set $M=\{f_{(2, 1, 2)}, f_{(1, 3, 2)}, f_{(4, 4, 2)}\}$. In the execution of the A* algorithm, there are two sets, *found* and *unfound*. At the beginning, the *found* is null and *unfound* stores the two nodes (0, 0) and (m, n) and the three given fragments, $f_{(2, 1, 2)}$, $f_{(1, 3, 2)}$ and $f_{(4, 4, 2)}$.

In the first step, node (0, 0) is selected from *unfound* to *found*, because there is no cost between it and the starting position of the shortest path from (0, 0) to (m, n). Then, expand the selected node (0, 0) to the elements in *unfound* as shown in Fig. 2, and the costs of the paths from (0, 0) to all elements in *unfound* are obtained. For example, the cost of the path from (0, 0) to $f_{(2, 1, 2)}$, $f_{(1, 3, 2)}$, $f_{(4, 4, 2)}$ and (m, n) are 1, 2, 6 and 10, reapectively.

By the A* algorithm, for each element in *unfound*, the cost of the path from it to the ending node (m, n) is estimated. The details of the estimating method will be discussed later. Since the cost of the paths from each element in *unfound*

to the ending node (m, n) can be estimated, the cost of the path from (0, 0) to (m, n) through each element in *unfound* can be gotten. Consider Fig. 2. The cost of the path from (0, 0) to (m, n) through $f_{(2, 1, 2)}$ is 2, the cost of the path from (0, 0) to (m, n) through $f_{(1, 3, 2)}$ is 5, the cost of the path from (0, 0) to (m, n) through $f_{(4, 4, 2)}$ is 6 and the cost of the path from (0, 0) to (m, n) is 10.

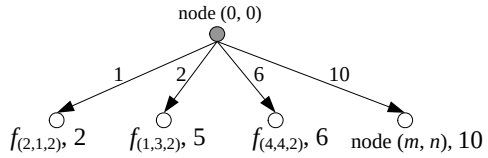

Fig. 2. Node (0, 0) selected to expand of Fig. 1.

Next, in *unfound*, the element whose cost of the path from (0, 0) to (m, n) through it is the lowest, is selected to be the expanding element. In our case, $f_{(2, 1, 2)}$ is selected from *unfound* to *found*, and $f_{(2, 1, 2)}$ is expanded to the elements in *unfound* as shown in Fig. 3-3. Thus the cost of the path from (0, 0) to $f_{(1, 3, 2)}$ through $f_{(2, 1, 2)}$ is 1+3=4, the cost of the path from (0, 0) to $f_{(4, 4, 2)}$ through $f_{(2, 1, 2)}$ is 1+1=2 and the cost of the path from (0, 0) to (m, n) through $f_{(2, 1, 2)}$ is 1+5=6. As well, from each element in *unfound*, the cost of the paths from it to (m, n) can be estimated. Through this estimation, we know that the path from (0, 0) to (m, n) through $f_{(2, 1, 2)}$ and $f_{(1, 3, 2)}$ is with cost 5, the path from (0, 0) to (m, n) through $f_{(2, 1, 2)}$ and $f_{(4, 4, 2)}$ is with cost 2 and the path from (0, 0) to (m, n) through $f_{(2, 1, 2)}$ is with cost 6.

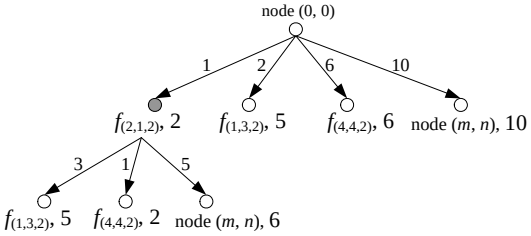

Fig. 3. $f_{(2, 1, 2)}$ selected to expand of Fig. 1.

By the method, if node (m, n) is selected next, the shortest path from (0, 0) to (m, n) is found by A^* algorithm. Fig. 4 shows that the found shortest path is from (0, 0) to (m, n) through two fragments $f_{(2, 1, 2)}$ and $f_{(4, 4, 2)}$, and the path corresponds the shortest path of the edit graph of X , Y and M as shown in Fig. 1.

As shown in Fig. 1. The found shortest path doesn't contain the fragment $f_{(1, 3, 2)}$. The reader can see that $f_{(1, 3, 2)}$ was not contained by the A^* algorithm. This is the advantage of our A^* algorithm. Many fragments may not need to be considered. The details of the A^* algorithm to solve the shortest path finding problem will be discussed in the below.

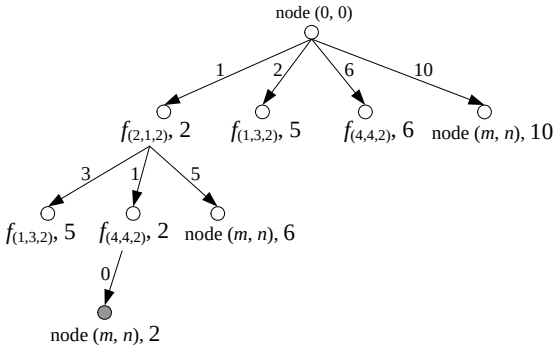

Fig. 4. Node (m, n) selected to expand of Fig. 1.

Consider the partially expanded tree shown in Fig. 5. Let p denote a node. Let $Dist(p)$ denote the length of the shortest path from the root of the tree to a terminal node through p. Let $D(p)$ denote the length of the shortest path from the root of the tree to p. Let $E(p)$ denote the length of the shortest path from p to a terminal node. Then $Dist(p)=D(p)+E(p)$.

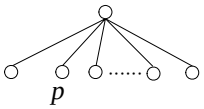

Fig. 5. p is a node of the expanded tree.

For A^* algorithm, there is an estimation scheme which estimates the lower bound of $E(p)$. Let $E^*(p)$ denote the estimated $E(p)$. Then $Dist^*(p)=D(p)+E^*(p)$. The A^* algorithm always selects the node with the lowest $Dist^*(p)$ to expand and stops if the selected node is a terminal node.

To apply the A^* algorithm to our problem, we use the edit graph as the input data. We always expand the nodes related to the fragments. For instance, consider Fig. 6. The nodes to be expanded at the first level are points (3, 2), (2, 4), (8, 2) … (5, 7).

Now, for each node p, we define $E^*(p)$ as follows. Let $S_{row}(p)$ denote the number of the rows in the area from p to (m, n) which do not contain diagonal edges and the number of the columns in the area from p to (m, n) which do not contain diagonal edges is denoted by $S_{col}(p)$. For instance, in Fig. 6, $S_{row}((0, 0))=2$ and $S_{row}((4, 3))=1$. $S_{col}((0, 0))=2$ and $S_{col}((4, 3))=1$. Obviously, from a node p to (m, n) of the edit graph, $E^*(p)$ can be defined as follows: $E^*(p)= S_{row}(p)+S_{col}(p)$.

In addition, we may use some other information to cut down the searching space. In Fig. 6,

consider the fragment $f_{(3, 2, 2)}$. Any solution consisting of this fragment will not contain use the fragment $f_{(2, 4, 2)}$. Therefore, when we consider $f_{(3, 2, 2)}$, we may ignore $f_{(2, 4, 2)}$. We now define the domination relation as follows: For two fragments f and f', f' is dominated by f, if $x(end(f')) > x(end(f))$ and $y(end(f')) > y(end(f))$. In our algorithm, suppose we are considering a fragment f, we shall ignore all of the fragments which are not dominated by f. For instance, in Fig. 6, $f_{(3, 2, 2)}$ dominates $f_{(5, 7, 2)}$, $f_{(7, 4, 2)}$ and $f_{(8, 6, 2)}$. $f_{(2, 4, 2)}$ dominates $f_{(3, 6, 2)}$, $f_{(8, 6, 2)}$ and $f_{(5, 7, 2)}$, and $f_{(7, 4, 2)}$ dominates $f_{(8, 6, 2)}$.

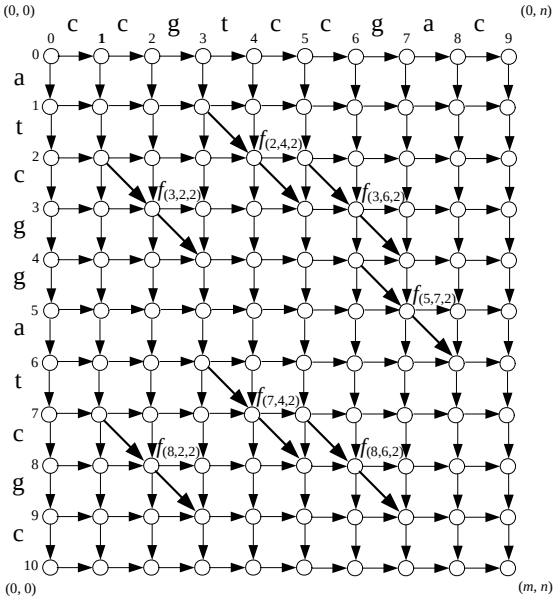

Fig. 6. The edit graph of two strings X=atcggatcgc and Y=ccgtccgac and a set M={$f_{(2, 4, 2)}$, $f_{(3, 2, 2)}$, $f_{(3, 6, 2)}$, $f_{(5, 7, 2)}$, $f_{(7, 4, 2)}$, $f_{(8, 2, 2)}$, $f_{(8, 6, 2)}$}.

In addition, for two fragments f and f', we define $Len(f, f')$ as the number of horizontal and vertical edges between f and f'. Let us consider Fig. 6. $Len(f_{(2, 4, 2)}, f_{(3, 6, 2)})$ is 1. $Len(f_{(2, 4, 2)}, f_{(5, 7, 2)})$ is 2. $Len(f_{(2, 4, 2)}, f_{(8, 6, 2)})$ is 4.

3.1 The Algorithm

The algorithm proposed in this paper consists of two phases, preprocessing and main algorithm. In Step 3 of the main algorithm as below, $E^*(f)$ has to be used in for loop. For each fragment f, since $E^*(f)$ doesn't change, we only compute $E^*(f)$ once for each f. In the preprocessing, we compute $E^*(f)$ for each f. In the main algorithm, a value *Link* is prepared for each fragment f and it will be used in recovering the shortest path step.

Let *Par* be pointer.
Input: Two strings $X=x_1x_2\ldots x_m$ and $Y=y_1y_2\ldots y_n$, and a set M of fragments.
Output: A shortest path from (0, 0) to (m, n)

Step 1. Consider two nodes (0, 0) and (m, n) as fragments $f_{(1, 1, 0)}$ and $f_{(m, n, 0)}$, respectively. Put fragments $f_{(1, 1, 0)}$ and $f_{(m, n, 0)}$ and set M into set *unfound*. Let set *found*=ϕ.

Step 2. For each fragment f in set *unfound* except fragment $f_{(1, 1, 0)}$, let $D(f)=\infty$ and $Dist^*(f)=\infty$. For $f_{(1, 1, 0)}$, let $D(f_{(1, 1, 0)})=0$ and $Dist^*(f_{(1, 1, 0)})=E^*(f_{(1, 1, 0)})$.

Step 3.
Do until $f = f_{(m, n, 0)}$
  Select f from set *unfound* such that $Dist^*(f)$ is the smallest, and remove f from *unfound* to *found*.
  For every f' in *unfound* which is dominated by f do
    Let $\overline{D(f')} = D(f) + Len(f, f')$.
    If $\overline{D(f')} < D(f')$, set $f'.\underline{Link} = f$.
    Let $D(f') = \min\{D(f'), \overline{D(f')}\}$.
    Let $Dist^*(f') = D(f')+E^*(f')$.
  End For
End Do

Step 4. Set $Par = f_{(m, n, 0)}$.
While *Par* is not *null*
  Print (i, j) pair pointed to by *Par*
  Advance *Par*
End While

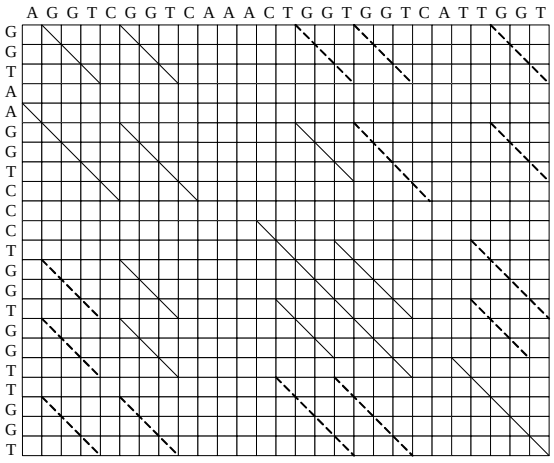

Fig 7. Given fragments distributed over the edit graph where the dotted fragments are not selected by the A^* algorithm.

If the given fragments are distributed over the diagonal line of the edit graph, the A^* algorithm can not filter out many fragments which will not appear in the solution. For instance, in Fig. 7, the fragments are distributed over the diagonal line of

the edit graph. There are few dotted fragments filtered by the *A** algorithm. Furthermore, if the given fragments are distributed over the edit graph, the *A** algorithm can filter out many fragments which will not appear in the solution.

3 Experiments

In this section, we shall present some experimental results processed by the *A** algorithm proposed in this paper and the dynamic programming approach proposed by Baker and Giancarlo[2]. In the experiments, 6 Hepatitis B Virus and 10 gene strings were used. The 6 Hepatitis B Viruses were HT, A4, pAD14, HMA, 8884H and 7768H. The 10 gene strings were CYP3A26, LOC489851, At2g30770, At1g11610, Fmo3, FMO3, P0452F10.9, P0452F10.11, B1131G07.28-2, B1131G07.28-1, which are considered to belong to the same group, CYP3A7, in NCBI Website. Table 1 shows the definition of terms used in the experiments. The experimental results are shown in Table 2. In Fig. 8, we compare the number of computed fragments by using the *A** algorithm and the number of computed fragments by using the dynamic programming approach.

Table 1. The definition of terms used in the experiments.

S_1:	The name of the first string.
S_2:	The name of the second string.
Len_1:	The length of the first string.
Len_2:	The length of the second string.
T:	The threshold of the length of fragments.
M:	The number of fragments whose length are greater than and equal to the threshold *T*.
A*:	The number of fragments which need to be computed by the *A** algorithm.
DP:	The number of fragments which need to be computed by the dynamic programming approach proposed by Baker and Giancarlo.
Rate:	The rate of numbers of fragments between by using the *A** algorithm and the dynamic programming approach.

Table 2. Experimental results

S_1	S_2	Len_1	Len_2	*T*	*M*	*A**	*DP*	*Rate*
HT	A4	663	666	7	74	8	74	10.81%
pAD14	HMA	666	666	7	70	16	70	22.85%
8884H	7768H	666	666	7	39	20	39	51.28%
CYP3A26	LOC489851	1925	1743	8	150	68	150	45.33%
At2g30770	At1g11610	1632	1449	8	107	72	107	67.29%
Fmo3	FMO3	2020	1805	8	140	87	140	62.14%
P0452F10.9	P0452F10.11	1590	1656	9	98	54	98	55.10%
B1131G07.28-2	B1131G07.28-1	1529	1890	9	49	49	49	100%

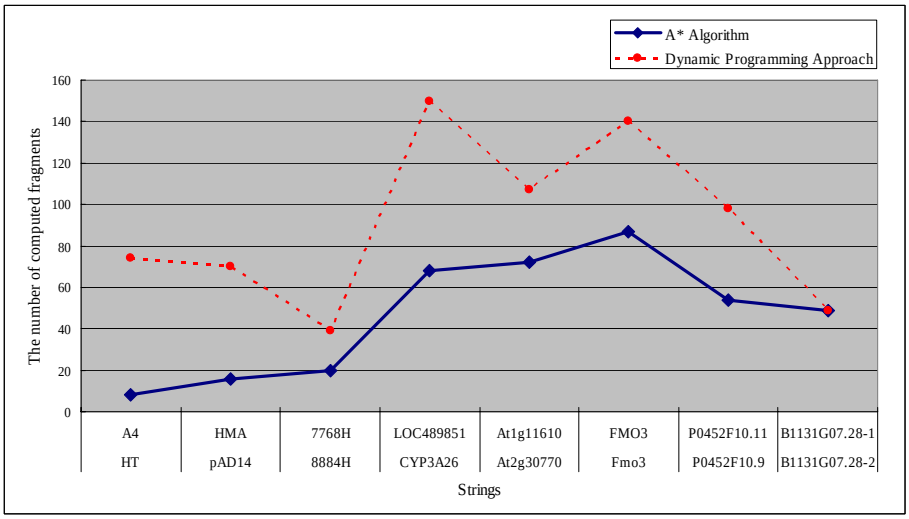

Fig. 8. Comparing the number of computed fragments by using the *A** algorithm and the number of computed fragments by using the dynamic programming approach.

5 Conclusion

In this paper, we proposed the application of the *A** algorithm to solve the longest common subsequence from fragments problem. The *A** algorithm can successfully filter out some fragments which wouldn't appear in solutions, and efficiently find a solution. If given fragments are distributed over the edit graph, the method can ignore a lot fragments. The method is quite efficient as the number of fragments which are needed to be computed is smaller than the dynamic programming approach proposed in Baker and Giancarlo. In general cases, a few fragments are computed in solving the problem. However, in worst cases, all fragments are needed to be computed in solving process.

References

[1] Faster Algorithms for String Matching with *k* Mismatches, Amir, A., Lewenstein, M. and Porat, E., Journal of Algorithms, Vol. 50, 2004, pp. 257-275.

[2] Sparse Dynamic Programming for Longest Common Subsequence from Fragments, Baker, B. S. and Giancarlo, R., Journal of Algorithms, Vol. 42, 2002, pp. 231-254.

[3] Approximation Algorithms for Multiple Sequence Alignment, Bafna, V., Lawler, E. L. and Pevzner, P. A., Theoretical Computer Science, Vol. 182, 1997, pp. 233-244.

[4] A Fast String Searching Algorithm, Boyer, R. S. and Moore, J. S., Communication of the ACM, Vol. 20, 1977, pp. 762-772.

[5] The Multiple Sequence Alignment problem in Biology, Carrillo, H. and Lipman, D. J., SIAM Journal on Applied Mathematics, Vol. 48, 1988, pp. 1073-1082.

[6] Jewels of Stringology, Crochemore, M. and Rytter, W., World Scientific, 2002.

[7] A Linear Space Algorithm for Computing Maximal Common Subsequences, Hirschberg, D. S., Communications of the ACM, Vol. 18, No. 6, 1975, pp. 341-343.

[8] A Fast Algorithm for Computing Longest Common Subsequences, Hunt, J. W. and Szymanski, T. G., Communications of the ACM, Vol. 20, No. 5, 1977, pp. 350-353.

[9] Fast Pattern Matching in Strings, Knuth, D., Morris, J. and Pratt, V., SIAM Journal on Computing, Vol. 6, 1977, pp. 323-350.

[10] Introduction to the Design and Analysis of Algorithms, Lee, R. C. T., Chang, R. C., Tseng, S. S. and Tsai, Y. T., Flag Corporation, Second Edition, ISBN:957-717-777-8, 1991.

[11] Identification of common Molecular Subsequences, Smith, T. F. and Waterman, M. S., Journal of Molecular Biology, Vol. 147, 1981, pp. 195-197.

[12] Alignments Without Low-Scoring Regions, Zhang, Z., Berman, P. and Miller, W., Research in Computational Molecular Biology (RECOMB), Vol. 5, 1998, pp. 294-301.