Fast Algorithms for Finding the Common Subsequence of Multiple Sequences*

Kuo-Si Huang, Chang-Biau Yang†, and Kuo-Tsung Tseng
Department of Computer Science and Engineering,
National Sun Yat-sen University, Kaohsiung, Taiwan
†cbyang@cse.nsysu.edu.tw

Abstract *- The longest common subsequence (LCS) algorithm is a useful method for measuring the identities and for finding similar subsequences in several sequences. Unfortunately, the longest common subsequence problem is NP-hard. In the past years, some algorithms, with several different approaches, have been proposed for finding the LCS of two given sequences. The complexity of these algorithms is about* $O(n^2)$ *in general and worst cases, where n is the length of sequences. When the given sequences are very long, these algorithms will take very long time and thus will become impractical. To overcome the disadvantage of time consuming, some efforts are devoted to the development of heuristic and approximate algorithms for finding the LCS. Such algorithms provide feasible solutions in practical application, such as searching in databases. However, there are few efforts for finding the LCS of more than two sequences. In this paper, we propose two approximate algorithms for finding the LCS of multiple sequences. The time complexity of our algorithms are* $O(\sigma kn)$ *and* $O(\sigma^2 kn + \sigma^3 n)$*, where* σ *is the size of symbol set, k and n are the number and length of input sequences, respectively. In the experimental results, our algorithm finds the common subsequences whose lengths are about* $0.8\times|LCS|$ *in average for two random sequences with uniform distribution. In the rank-identity experimental result, it shows that our methods are suitable in practical application.*

Keywords: computational biology, common subsequence, approximate algorithm.

1. Introduction

For a sequence s, over a finite symbol set Σ, a subsequence of s is the sequence t that can be obtained from s by deleting zero or more (not necessarily contiguous) symbols. The longest common subsequence (LCS) problem of multiple sequences, or *multiple sequences longest common subsequence* (*k-LCS*) problem, where k is the number of input sequences, is defined as follows: Given k ($k \geq 2$) sequences, we want to find the longest sequence t such that t is the common subsequence of the k sequences. That is, given a k-sequence set $S = \{s_1, s_2, \ldots, s_k\}$, sequence t is the *common subsequence* of S if t can be obtained from removing zero or more symbols from each sequences in S. In addition, if t is the longest one of all possible common subsequences of S, t is called the longest common subsequence. If the sequences are in biological domain, the symbol set Σ is fixed, such as Σ = {a, g, c, t} in DNA sequences, Σ = {a, g, c, u} in RNA sequences and Σ contains the twenty types of amino acids in protein sequences.

The LCS problem is a famous classical problem in computer science and molecular biology, and it has been studied extensively over more than 30 years. We can regard the common subsequence as the identical part of sequences. According to the LCS, we can reconstruct the relative alignment. In the past years, some algorithms have been proposed for finding the LCS of two sequences based on three major approaches, dynamic programming, contour, and diagonal approaches [2, 7, 13, 14, 16]. The complexity of most algorithms is $O(n^2)$ in general and worst case, where n is the length of sequences. Until now, the lowest time complexity is $O(n^2/\log n)$ for 2-LCS, which is proposed by Masek and Paterson and based on the "Four Russians" trick [12]. If the given sequences are very long, these algorithms will take very long time and become impractical [1].

Most previous researches focused on the relationship of two sequences, because 2-LCS is in the P (polynomial) class. However, the relationship between multiple sequences is also important. The previous results on the k-LCS are very few, because the problem is NP-hard even on binary alphabets [11]. If the dynamic programming strategy is applied, it needs $O(n^k)$ time and space to solve the problem, where k is the number of given sequences and n is the length of the longest sequence. When the number of input sequences grows, it is not practical to use

* This research work was partially supported by the National Science Council of the Republic of China under contract NSC-90-2213-E-110-015.

dynamic programming to find the optimal k-LCS. Therefore, we have to design new approaches to solve the problem for feasibility.

For solving k-LCS problem in multiple sequences, Hakata and Imai [6] proposed an algorithm with time complexity $O(n\sigma k+D\sigma k(log^{k-3}n+log^{k-2}\sigma))$, where D is the number of dominant matches (contours). Irving and Fraser [8] also proposed two algorithms for k-LCS. One requires $O(kn(n-l)^{k-1})$ time, where l is the length of the LCS. The other requires $O(kl(n-l)^{k-1}+k\sigma n)$ time, where l is the length of the LCS, and σ is the alphabet size. If the number of given sequences is large, these algorithms may be infeasible for solving the problem. One feasible approach is to find some common subsequences that are near to LCS. One simple approximate algorithm to find the common subsequence is the long-run algorithm [9]. It computes the minimum number of occurrence for each symbol in all sequences, and returns the maximum value of occurrence among all symbols. The simple long-rum algorithm is fast and can report a solution with approximate ratio $|\Sigma|$. But the algorithm returns the common subsequence consisting of only single symbol. It seems useless in most practical applications.

Elloumi [4] proposed a heuristic algorithm with time complexity $O(k^2 n^2 log\ n)$. His method is to find the same regions in input sequences, and then assemble these regions as a CS (common sequence). In the data of real sequences with high similarity, Elloumi's algorithm could obtain good results. But it obtains bad result when the sequences are random or in different family and the number of sequences is large. In addition, it is not an approximate algorithm.

Bonizzoni et al. [3] developed an approximate algorithm for k-LCS called Expansion Algorithm (EA). Their algorithm first compresses sequences to streams by the same concept of run length encoding (RLE), then progressively find a common sequence t of all streams by the bottom-up tree merging technique. Finally, expand all alphabets and all the substrings (contiguous subsequence) of common sequence t to find the longest common subsequence for each sequence in S. The time required for the algorithm is $O(k\ n^3 \log n)$, where k is the number of sequence and n is the length of sequences. The algorithm does not always return real LCS, and the sequences cannot always be compressed in real data. To improve the expansion algorithm, Tsai and Hsu proposed a minimum-spanning-tree-based greedy (MSTG) algorithm [15] for replacing the original bottom-up tree merging technique. The performance of EA algorithm is related to the common stream of all sequences.

In this paper, we propose new approximate algorithms for finding the LCS of multiple sequences. The time complexities of our algorithms are $O(\sigma kn)$ and $O(\sigma^2 kn+\sigma^3 n)$, respectively, where σ is the size of symbol set, k and n are the number and the length of input sequences, respectively. In the experimental results, our algorithms find the common subsequences whose lengths are about $0.8\times|\text{LCS}|$ in average for two random DNA sequences. In the rank-identity experimental result, it shows that our methods are suitable in practical application.

This paper is organized as follows. In Section 2, we review the long run algorithm, the expansion algorithm, and the best next algorithm. Our algorithms and the time complexity analyses are described in Section 3. For illustrating the performance of our algorithm, we give the experimental results for several cases in Section 4. Finally, Section 5 concludes this paper.

2. Related Works

2.1 The long run algorithm

The long run algorithm is proposed by Jiang and Li [9], it is a simple and fast approximate algorithm. Suppose sequence t is the approximate LCS of input k-sequence in $S=\{s_1, s_2, \ldots, s_k\}$ on the finite symbol set $\Sigma=\{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. Let $f(s, g)$ denote the total number of symbol g in sequence s. The long run algorithm is as follows.

$$t=\max_{1\le j\le|\Sigma|}\{\min_{1\le i\le|S|}\{f(s_i, g_j)\}\}$$

The time complexity of the long run algorithm is $O(kn)$, where n and k are the length and the number of input sequences, respectively. The long run algorithm is a $|\Sigma|$-approximate algorithm. It is fast and simple to be embedded in other algorithms. So one may say that every heuristic k-LCS algorithm is $|\Sigma|$-approximate if its time complexity is not less than $O(kn)$ by embedding the long run algorithm.

2.2 The expansion algorithm

The expansion algorithm is proposed by Bonizzoni et al. [3]. The main spirit of this algorithm is to compress all sequences to streams first, where a stream is the sequence without consecutive identical alphabets. Then find all short common streams, whose length is not more than 2, and a longest common stream z of all sequences in S. Then one can expand all substrings of stream z and all short common streams to find a common subsequence of S with maximal length, but it may not be the real LCS of S. The long run algorithm is also embedded in the expansion algorithm by finding all short common streams and expanding them.

The time complexity of the expansion algorithm is $O(kn^3\log n)$, where k and n are the size and length of input sequences in S, respectively. In general cases, the performance of the expansion algorithm is related to the efficiency for finding the longest common stream. The result of the expansion algorithm is surely better than the long run algorithm, and it is a $|\Sigma|$-approximate algorithm. However, the expansion

algorithm is still impractical when n is large. To improve the performance for finding the common stream in the expansion algorithm, Tsai and Hsu proposed an algorithm using minimum-spanning-tree-based greedy algorithm (MSTG) [15] to find a better common stream.

2.3 The best next algorithm

The best next heuristic algorithm is a typical example of fast 2-LCS heuristic algorithm [5, 10], and it can be easily extended to become a k-LCS heuristic. Its complexity is $O(\sigma kn)$, where σ is the size of alphabets Σ, k and n are the number and length in S, respectively. Let $p(s, t+g)$ denote the minimal sequence containing subsequence $(t+g)$, the common subsequence t adding symbol g, in sequence s. The best next algorithm is to select a symbol g appending to the common sequence t in each round, where symbol g maximizes $\min_{1\leq i\leq |S|}\{|s_i| - p(s_i, t+g)\}$, that is, the length of the shortest suffix (leaving) sequences is maximal.

Though the best next algorithm may perform well in experiments, it is not a $|\Sigma|$-approximation algorithm. One can find a counter example easily. Given three sequences s_1 = `aaaaaagcccccc`, s_2 = `ccccccgaaaaaa`, and s_3 = `acgacacacgacca` whose lengths are 13, 13, and 14, respectively. The best next heuristic algorithm will lead the result that the common subsequence is `g` with length 1, but the LCS is `aaaaaa` or `gggggg` with length 6. Based on the counter example, we can conclude that the best next algorithm is not c-approximate for any constant c in general. However, the best next heuristic algorithm points out a wrinkle that the problem can be pruned as a small one.

3. Our Algorithms

It is a special case that the symbol set Σ is fixed in the biological sequences. The property helps us to develop $|\Sigma|$-approximate algorithms. We can combine the long-run algorithm and the pruned wrinkle in the best next heuristic algorithm. Our algorithms are given as follows.

Algorithm 1: ELR (Enhanced Long Run)

Input: k sequences $S = \{s_1, s_2, \ldots, s_k\}$ on a finite alphabet set Σ.

Output: A common subsequence t of S.

Step 1: Preprocess all sequences in S and find the long run alphabet g.

Step 2: Append alphabet g to common subsequence t.

Step 3: For each sequence s_i in S, prune the prefix sequence p_i with the minimal length, such that t is the subsequence of the pruned sequence p_i.

Step 4: Find the long run alphabet h of sequences $(s_i - p_i)$ for each i, if there are several long run alphabets and g is one of them, $h = g$; otherwise, randomly select a long run alphabet as h.

Step 5: If h is not empty (null), append the alphabet h to common subsequence t, GOTO Step 3; otherwise, GOTO Step 6.

Step 6: Output sequence t.

Theorem 1. The time complexity of Algorithm ELR is $O(\sigma kn)$, where $\sigma = |\Sigma|$, $k = |S|$, and n is the length of the longest sequence in S, respectively.

Proof. Step 1 executes the long run algorithm, and it requires $O(kn)$. The time complexity of Step 2 is $O(1)$. Step 3 requires $O(k)$ using a programming trick by preprocessing in Step 1. Step 4 requires $O(\sigma k)$ using the programming trick as the preprocessing in Step 1, we need not perform the long run algorithm again. Step 5 will call Step 3 and Step 4 at most $O(n)$ times. It means that the time complexity of Step 3, Step 4 and Step 5 is $O(\sigma kn)$. Step 6 requires $O(1)$ for output the result. So, the time complexity of this algorithm is $O(\sigma\, k\, n)$. □

Theorem 2. Algorithm ELR is $|\Sigma|$-approximate.

Proof. Because Algorithm ELR prunes the problem as smaller one in each round and finds the common subsequence recursively. After pruning the prefix sequences, there may be better long run symbol than the original one. It is easily to show that the length of common subsequence obtained from Algorithm ELR is better than the long run algorithm, so it is also $|\Sigma|$-approximate since the long run algorithm is $|\Sigma|$-approximate. □

One can examine that the solution obtained by Algorithm ELR may not be optimal for the k-LCS problem by the following counter example. Given three sequences s_1 = `agctt`, s_2 = `tagact` and s_3 = `cagtct` whose lengths are 5, 6 and 6, respectively. Algorithm ELR will lead the result that the common subsequence is `tt` with length 2, but the LCS is `agct` with length 4.

For getting better common subsequence, one can examine the above example again. In the example, we have two observations. First, the best next heuristic algorithm is performing well to get the LCS exactly. Second, all three sequences are ended by alphabet `t`. By the optimality principle, after pruned the last alphabet, the LCS of pruned sequences can be found and it is a part of the original LCS. That is, if we pruned the last alphabet `t`, the three sequences will become s'_1 = `agct`, s'_2 = `tagac` and s'_3 = `cagtc` whose lengths are 4, 5 and 5, respectively. The LCS of s'_1, s'_2 and s'_3 is `agc`, which are the first three alphabets of LCS = `agct`. By the two observations, there may be an approximate algorithm whose performance is near to the best next heuristic algorithm.

Before presenting our next algorithm, we need to define some notations which will be used later. Let the vector of maximal available symbols denote as $v_s(g, h)$, which is the number of symbol h prior to (containing g) the last selected symbol g in sequence s, where $g, h \in \Sigma$. The vector of maximal available symbols in S is denoted as $v(g, h) = \min_{s\in S}\{v_s(g, h)\}$. Let t denote the suffix common subsequence, the calculation $S - t$ represents that for each sequence s_i in S, the suffix sequence p_i with minimal length is pruned away, where t is the common subsequence of the pruned sequence p_i, where $1 \leq i \leq k$. Let $d(g)$ represent the degree of symbol g, which means the largest times of $v(g, j)$ will be recursively covered by $v(i, j)$ for symbol $i \neq g$. According to degree $d(g)$, we define the parent symbol $p(g)$ to be the most parent symbol such that $d(g)$ is maximal.

For example, consider three sequences s_1 = `agctt`, s_2 = `tagact` and s_3 = `cagtct`. The vectors of $v(i, j)$ are $v(\texttt{a}, j) = [1, 0, 0, 0]$, $v(\texttt{g}, j) = [1, 1, 0, 0]$, $v(\texttt{c}, j) = [1, 1, 1, 0]$, $v(\texttt{t}, j) = [1, 1, 1, 2]$, the degree $d(j) = [3, 2, 1, 0]$ and $p(j)$ = [`t`, `t`, `t`, `t`], where $j \in$ {`a`, `g`, `c`, `t`}. After pruning symbol t, we have s'_1 = `agct`, s'_2 = `tagac` and s'_3 = `cagtc`. The vectors of $v(i, j)$ are $v(\texttt{a}, j) = [1, 0, 0, 0]$, $v(\texttt{g}, j) = [1, 1, 0, 0]$, $v(\texttt{c}, j) = [1, 1, 1, 0]$, $v(\texttt{t}, j) = [0, 0, 0, 1]$, the degree $d(j) = [2, 1, 0, 0]$ and $p(j)$ = [`c`, `c`, `c`, `t`], where $j \in$ {`a`, `g`, `c`, `t`}. Our next algorithm can help us to find a common subsequence as follows.

Algorithm 2: BNMAS (Best Next for Maximal Available Symbols)

Input: k sequences $S = \{s_1, s_2, \ldots, s_k\}$ on a finite alphabet set Σ.

Output: A common subsequence t of S.

Step 1: Preprocess all sequences in S, the suffix common subsequence t = `null`, and g is the long run symbol.

Step 2: For each symbol, calculate $v(i, j)$ and long run symbol g in $S - t$, where $i, j \in \Sigma$.

Step 3: Calculate the degree $d(i)$ for each symbol i between $v(i, j)$ with topological relations.

Step 4: Find the symbol h with maximal ($d(h)$ + max $_{j\in\Sigma}\{v(h, j)\}$), if several alphabets have the same $\max_{i\in\Sigma}\{d(i) + \max_{j\in\Sigma}\{v(i, j)\}\}$, select the symbol with the maximal degree as h, and g is one of them, $h = g$; otherwise, randomly select a alphabet with $\max_{i\in\Sigma}\{d(i) + \max_{j\in\Sigma}\{v(i, j)\}\}$ as h.

Step 5: If h is not empty (null), append symbol $p(h)$, the parent symbol of h, to common subsequence t, GOTO Step 2; otherwise, GOTO Step 6.

Step 6: Output sequence t.

Theorem 3. The time complexity of Algorithm BNMAS is $O(\sigma^2\, k\, n + \sigma^3\, n)$, where $\sigma = |\Sigma|$, $k = |S|$, and n is the length of the longest sequence in S, respectively.

Proof. Step 1 executes the long run algorithm, it requires $O(kn)$. The time complexity of Step 2 is $O(\sigma^2 k)$, Step 3 requires $O(\sigma^3)$ and Step 4 requires $O(\sigma^2)$ using programming tricks. In Step 3, we can use the algorithm for finding the longest paths in DAG (directed acyclic graph), it requires $O(\sigma^2)$ because there is at most σ nodes. Step 5 calls Step 2, Step 3 and Step 4 at most $O(n)$ times. It means that Step 2, Step 3, Step 4 and Step 5 requires $O(\sigma^2 kn + \sigma^3 n)$. Step 6 requires $O(1)$ for output the result. So, the time complexity of this algorithm is $O(\sigma^2 kn + \sigma^3 n)$. □

Theorem 4. Algorithm BNMAS is $|\Sigma|$-approximate.

Proof. It is similar to Algorithm ELR, Algorithm BNMAS prunes the problem as smaller one in each round and finds the common subsequence recursively. After pruning the prefix sequences, Step 4 ensures that the better symbol than long run one can be found. The reason is that we consider the combination of degree $d(i)$ and the cover relationship between $v(i, j)$, where $i, j \in \Sigma$. So the length of common subsequence obtained from Algorithm BNMAS is better than the long run algorithm, so Algorithm BNMAS is also $|\Sigma|$-approximate. □

4. Experimental Results

In this section, we will show some experimental results of our algorithms. We test algorithm in binary and DNA sequences for the lengths 100, 300, 500, and 1000. For binary sequences, we compare the results of our algorithm with the expansion algorithm for four sequences. For DNA sequences, we test our algorithm for 2, 4, 10, and 20 sequences. The sequences are generated with randomly uniform distribution. The ratios are the average of 100 sets for each testing condition case.

In Table 1, we simulate Algorithm BNMAS in one-way and duplex direction. The one-way direction means that our algorithm only finds the common subsequence from tail to head, and duplex direction returns the longer one of the common subsequences of both sides from head to tail and from tail to head. The duplex direction requires twice time of the one-way one. The results of our algorithm in one-way and duplex direction are both better than the expansion algorithm for four random sequences on binary alphabets. The time complexity of the expansion algorithm is larger than ours. We conclude that our algorithm is more practical than the expansion algorithm.

It is well known that finding k-LCS is hard. For the experimental result of multiple sequences, we design a method to generate multiple sequences with their LCS as follows. Generate two random

sequences s_1 and s_2 with uniform distribution and find the LCS t of s_1 and s_2. Inserting random symbols into LCS t until to the expected length can generate other sequences in the sequence set.

In Table 2, we show the performance ratio of Algorithm BNMAS with duplex direction on DNA sequences with different number and length. In this table, one can see that the ratios of our algorithm are about from 0.885 to 0.823 in average when sequence lengths are from 100 to 1000. It means that out algorithm can get a good common subsequence whose length is about 0.8 × |LCS| when the sequence length is less than 1000. Besides, our algorithm is stable for two sequences. Because our algorithm is fast, one can use our algorithm to select possible sequences that have high identity to the query sequences in database searching.

Table 1. The performance ratios of the expansion algorithm and Algorithm BNMAS for four random sequences on binary alphabets. Ratio = |CS|/|LCS|.

Seq Length	Expansion	Our Algorithm	
		One-way	Duplex
100	0.746	0.803	0.820
300	0.697	0.726	0.742
500	0.681	0.711	0.720
1000	0.674	0.690	0.696

Table 2. The performance ratio of Algorithm BNMAS on DNA sequences. Ratio = |CS|/|LCS|. The numbers of sequences are 2, 4, 10, and 20, respectively.

Len \ #(seq)	2	4	10	20
100	0.886	0.774	0.763	0.814
300	0.848	0.665	0.602	0.579
500	0.835	0.636	0.550	0.529
1000	0.823	0.602	0.507	0.480

Figure 1 is the rank-identity graph of two DNA sequences with different identities. Here, we fix a sequence s and generate other 200 sequences $\{s_1, s_2, \ldots, s_{200}\}$ by mutating s randomly to obtain 200 sequence pairs with various identities. We draw the rank-identity graph for each pair (s, s_i), where $1 \le i \le 200$. Figure 1 approves that one can use our algorithm as a selector to select possible candidates in database searching. For example, we can set 0.9 × length as a threshold to select candidate sequences. Then one can use the precise LCS algorithms to find the best one in the candidate sequences. The approach can fast find the near sequences with high probability.

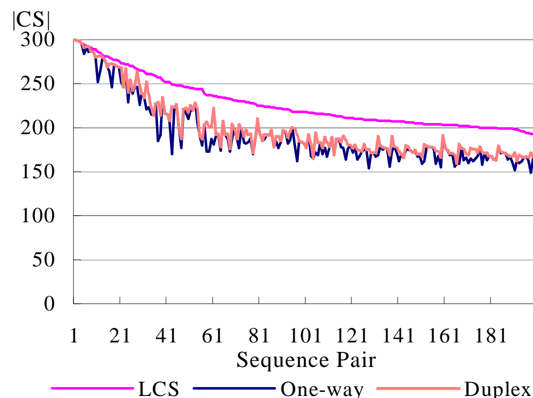

Figure 1. The rank-identity graph of two DNA sequences with different identities. The top line is of the LCS. The middle and bottom lines are the lengths of common subsequence found by our algorithm in one-way and duplex directions respectively. The length of input sequences is 300.

5. Conclusion

In this paper, we propose two approximate algorithms for finding the longest common subsequence of multiple sequences. The time complexities of our algorithms are $O(\sigma k n)$ and $O(\sigma^2 k n + \sigma^3 n)$ respectively, where σ is the size of symbol set, k and n are the number and length of input sequences. The first algorithm improves the long run algorithm but its time complexity is σ-multiple more than the long run algorithm. Note that symbol set Σ is finite and fixed, and $\sigma = |\Sigma|$ is a constant in biological sequences.

In the experimental results, our algorithm performs well and stable for two sequences on DNA sequences. Our algorithm is fast, and it can be used to select possible candidate sequences that have high identity to the query sequences in database searching. Compared with the expansion algorithm, our algorithm is more practical. In the future, we will analyze the real data and study the cutting approach to improve and refine our algorithm, especially for input sequences with larger size, in either number or length.

References

[1] L. Bergroth, H. Hakonen and T. Raita, “New approximation algorithms for longest common subsequences”, *In Proceedings of String Processing and Information Retrieval: A South American Symposium, SPIRE 1998*, pp. 32-40, 1998.

[2] L. Bergroth, H. Hakonen and T. Raita, “A survey of longest common subsequence

algorithms", *In Proceedings of Seventh International Symposium on String Processing and Information Retrieval*, *SPIRE 2000*, pp. 39-48, 2000.
[3] P. Bonizzoni, G. D. Vedova and G. Mauri, "Experimenting an approximation algorithm for the LCS", *Discrete Applied Mathematics*, vol. 110, pp. 13-24, 2001.
[4] M. Elloumi, "Comparison of strings belonging to the same family", *Information Sciences*, vol. 111, pp. 49-63, 1998.
[5] C. B. Fraser, "Subsequences and supersequences of strings", *University of Glasgow*, *Computing Science Department Research Report*, *TR-1995-16*, 1995.
[6] K. Hakata and H. Imai, "The longest common subsequence problem for small alphabet size between many strings", *In Proceedings of the Third International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science 650*, Springer Verlag, pp. 469-478, 1992.
[7] D. S. Hirschberg, "Algorithms for the longest common subsequence problem", *Journal of ACM*, vol. 24, pp. 664-675, 1977.
[8] R. W. Irving and C. B. Fraser, "Two algorithms for the longest common subsequence of three (or more) strings", *In Proceedings of CPM'92, the Fourth Annual Symposium on Combinatorial Pattern Matching*, Arizona *In Lecture Notes in computer Science 644*, Springer Verlag, pp. 214-229, 1992.
[9] T. Jiang and M. Li, "On the approximation of shortest common supersequences and longest common subsequences", *SIAM Journal on Computing*, vol. 24, pp. 1122-1139, 1995.
[10] T. Johtela, J. Smed, H. Hakonen and T. Raita, "An efficient heuristic for the LCS problem", *Third South American Workshop on String Processing*, *WSP'96*, pp. 126-140, August 1996.
[11] D. Maier, "The complexity of some problems on subsequences and supersequences", *Journal of ACM*, vol. 25, pp. 322-336, 1978.
[12] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances", *Journal of Computer and System Sciences*, vol. 20, pp. 18-31, 1980.
[13] C. Rick, "Simple and fast linear space computation of longest common subsequences", *Information Processing Letters*, vol. 75, pp. 275-281, 2000.
[14] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *Journal of Molecular Biology*, vol. 147, pp. 195-197, 1981.
[15] Y. T. Tsai and J. T. Hsu, "An approximation algorithm for multiple longest common subsequence problems", *In Proceeding of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, *SCI2002*, pp. 456-460, 2002.
[16] C. B. Yang and R. C. T. Lee, "Systolic algorithms for the longest common subsequence problem", *Journal of the Chinese Institute of Engineers*, vol. 10, no. 6, pp. 691-699, 1987.