

Efficient Polynomial-Time Algorithms for the Constrained LCS Problem with Strings Exclusion *

Hsing-Yen Ann

National Center for High-Performance Computing, Tainan, Taiwan

Chang-Biau Yang[†] and Chiou-Ting Tseng

Department of Computer Science and Engineering

National Sun Yat-sen University, Kaohsiung 80424, Taiwan

November 26, 2012

Key words: design of algorithms, longest common subsequence, constrained LCS, NP-hard, finite automata

Abstract

In this paper, we revisit a recent variant of the longest common subsequence (LCS) problem, the string-excluding constrained LCS (STR-EC-LCS) problem, which was first addressed by Chen and Chao [8]. Given two sequences X and Y of lengths m and n , respectively, and a constraint string P of length r , we are to find a common subsequence Z of X and Y which excludes P as a substring and the length of Z is maximized. In fact, this problem cannot be correctly solved by the previously proposed algorithm. Thus, we give a correct algorithm with $O(mnr)$ time to solve it. Then, we revisit the STR-EC-LCS problem with multiple constraints $\{P_1, P_2, \dots, P_k\}$. We propose a polynomial-time algorithm which runs in $O(mnR)$ time, where $R = \sum_{i=1}^k |P_i|$, and thus it overthrows the previous claim of NP-hardness.

1 Introduction

Computing the similarity of two sequences is one of the most important fundamental techniques in computer science. Let $X = x_1x_2 \cdots x_m$ be a sequence(string) of length m over some fixed alphabet Σ , where $x_i \in \Sigma$. Let $X_{i..j} = x_ix_{i+1} \cdots x_j$ denote a *substring* of X , where $1 \leq i \leq j \leq m$. A substring $X_{i..j}$ is called a *prefix* or a *suffix* of X if $i = 1$ or $j = m$, respectively. A *subsequence* of X is obtained by deleting zero or more symbols from X arbitrarily. A sequence is called a *common subsequence*

*This research work was partially supported by the National Science Council of Taiwan under contract NSC 99-2221-E-110-048.

[†]Corresponding author: cbyang@cse.nsysu.edu.tw

of two sequences X and Y if it is a subsequence of both X and Y . Given two sequences X and Y , the *longest common subsequence* (LCS) problem is to find a subsequence of X and Y whose length is the longest among all common subsequences of the two given sequences. The LCS problem is a well-known measurement for computing the similarity of two strings. It can be widely applied in diverse areas, such as file comparison, pattern matching and computational biology. The most referred algorithm, proposed by Wagner and Fischer [31], solves the LCS problem in quadratic time with the *dynamic programming* (DP) technique. Other advanced algorithms were proposed in the past decades [2, 3, 5, 18, 19, 21, 22, 32]. When the number of input sequences is not fixed, finding the LCS of multiple sequences has been proved to be NP-hard [25], and therefore some approximate and heuristic algorithms were proposed [7, 27].

Applying the constraints to the LCS problem is meaningful for some biological applications [28]. Therefore, the *constrained LCS* (CLCS) problem, a recent variant of the LCS problem which was first addressed by Tsai [29], has received much attention. Given two input sequences X and Y of lengths m and n , respectively, and a constrained sequence P of length r , the CLCS problem is to find the common subsequences Z of X and Y such that P is a subsequence of Z and the length of Z is the maximum. In the following, without loss of generality, we assume that $r \leq n \leq m$. The most referred algorithms were proposed independently [6, 9] which solve the CLCS problem based on the DP technique in $O(mnr)$ time and space. Some improved algorithms have also been proposed [13, 20]. Iliopoulos *et al.* [23] discussed the LCS and CLCS problems on the indeterminate strings. Deorowicz [11] gave a bit-parallel algorithm for solving the CLCS problem. Ann *et al.* [4] considered the sequences which are in *run-length encoding* (RLE) format, a linear-time compressing technique, and greatly reduced the number of elements required to be evaluated in the DP lattice. Moreover, Peng *et al.* [26] extended the problem to the one with weighted constraints, a more generalized problem. Referring to the variant of multiple constraints, the *multiple CLCS* problem was also proved to be NP-hard [17].

Recently, Gotthilf *et al.* [16] proposed a variant of the CLCS problem, the *restricted LCS* problem, which excludes the given constraint as a subsequence of the answer. Furthermore, they proved that the restricted LCS problem also becomes NP-hard when the number of constraints is not fixed. Independently, Chen and Chao [8] addressed the more generalized forms of the CLCS problem, the *generalized constrained longest common subsequence* (GC-LCS) problem. Given two input sequences X and Y of lengths m and n , respectively, and a constraint string P of length r , the GC-LCS problem is a set of four problems which are to find the longest common subsequence of X and Y including/excluding P as a subsequence/substring, respectively. Table 1 shows the four

Table 1: The four GC-LCS problems defined by Chen and Chao [8] and the time complexities of their methods.

	Including P	Excluding P
As a subsequence	SEQ-IC-LCS	SEQ-EC-LCS $O(mnr)$
As a substring	STR-IC-LCS $O(mnr)$	STR-EC-LCS $O(mnr)$

problems. Evidently, the original CLCS problem addressed by Tsai [29] is specified as SEQ-IC-LCS, and the restricted LCS problem addressed by Gotthilf *et al.* [16] is specified as SEQ-EC-LCS.

For the problems in Table 1 except SEQ-IC-LCS, Chen and Chao proposed the corresponding algorithms which are all in $O(mnr)$ time. However, as we will show soon, their algorithm for STR-EC-LCS cannot work correctly. Besides, Chen and Chao [8] stated that "one can further show that the STR-IC-LCS, SEQ-EC-LCS, and STR-EC-LCS problems with multiple constrained patterns are also NP-hard". We will propose an algorithm of polynomial time for solving STR-EC-LCS with multiple constraints, thus it implies that the above statement is not correct for STR-EC-LCS. In 2010, Farhana *et al.* [15] proposed an $O(r(m+n) + (m+n)\log(m+n))$ time algorithm for all four variants in Table 1 by using the finite automata with an important assumption that Σ is constant. In 2012, Deorowicz [12] proposed a quadratic algorithm to the STR-IC-LCS problem and also claimed that the time complexity of Farhana *et al.* is wrong. Recently, Tseng *et al.* [30] proposed efficient algorithms for solving the STR-IC-LCS problem, with single constraint and multiple constraints.

In this paper, we focus on the STR-EC-LCS problem, with single constraint and with multiple constraints. In Section 2, we review Chen and Chao's algorithm for the STR-EC-LCS problem with single constraint and give a counterexample which breaks their algorithm. In Section 3, a correct straightforward backtracking algorithm is given, though its time complexity is exponential. Then, we propose an efficient algorithm with time complexity $O(mnr)$ in Section 4. In Section 5, we consider the variant of the STR-EC-LCS problem that multiple constraint strings are given and all of them have to be excluded. We propose an efficient polynomial-time algorithm to solve the multiple STR-EC-LCS problem and thus it shows that this problem is not NP-hard. And finally, our conclusion is given in Section 6.

2 Counterexamples to Chen and Chao's Algorithm

Definition 1. (STR-EC-LCS problem [8]). *Given two input sequences X and Y of lengths m and n , respectively, and a constraint string P of length r , the STR-EC-LCS (string excluding) problem*

$$L_{Chen}(i, j, k) = \begin{cases} L_{Chen}(i-1, j-1, k) & \text{if } k = 1 \text{ and } x_i = y_j = p_k, \\ \max \left\{ \begin{array}{l} 1 + L_{Chen}(i-1, j-1, k-1), \\ 1 + L_{Chen}(i-1, j-1, k) \end{array} \right\} & \text{if } k \geq 2 \text{ and } x_i = y_j = p_k, \\ 1 + L_{Chen}(i-1, j-1, k) & \text{if } x_i = y_j \text{ and } (k = 0, \text{ or } k > 0 \text{ and } x_i \neq p_k), \\ \max \left\{ \begin{array}{l} L_{Chen}(i-1, j, k), \\ L_{Chen}(i, j-1, k) \end{array} \right\} & \text{if } x_i \neq y_j. \end{cases}$$

$$L_{Chen}(i, 0, k) = L_{Chen}(0, j, k) = 0 \quad \text{for any } 0 \leq i \leq m, 0 \leq j \leq n, 0 \leq k \leq r.$$

Figure 1: Chen and Chao's recurrence formula [8] for the STR-EC-LCS problem.

is to find the longest common subsequence Z of X and Y excluding P as a substring.

For the STR-EC-LCS problem, based on the following theorem, Chen and Chao [8] proposed a dynamic programming algorithm with $O(mnr)$ time. Their recurrence formula is shown in Figure 1, where $L_{Chen}(i, j, k)$ denotes the length of the LCS of $X_{1..i}$ and $Y_{1..j}$ excluding $P_{1..k}$ as a substring.

Theorem 1. [8] Let $X = x_1x_2 \cdots x_m$, $Y = y_1y_2 \cdots y_n$ and $P = p_1p_2 \cdots p_r$. Let $S_{i,j,k}$ denote the set of all LCSs of $X_{1..i}$ and $Y_{1..j}$ excluding $P_{1..k}$ as a substring. If $z_{1..l} \in S_{i,j,k}$, the following conditions hold:

- (1) If $x_i = y_j = p_k$ and $k = 1$, then $z_l \neq x_i$ and $z_{1..l} \in S_{i-1,j-1,k}$.
- (2) If $x_i = y_j = p_k$ and $k \geq 2$, then $z_l = x_i = y_j = p_k$ and $z_{l-1} = p_{k-1}$ implies $z_{1..l-1} \in S_{i-1,j-1,k-1}$.
- (3) If $x_i = y_j = p_k$ and $k \geq 2$, then $z_l = x_i = y_j = p_k$ and $z_{l-1} \neq p_{k-1}$ implies $z_{1..l-1} \in S_{i-1,j-1,k}$.
- (4) If $x_i = y_j = p_k$ and $k \geq 2$, then $z_l \neq x_i$ implies $z_{1..l} \in S_{i-1,j-1,k}$.
- (5) If $x_i = y_j$ and $x_i \neq p_k$, then $z_l = x_i = y_j$ and $z_{1..l-1} \in S_{i-1,j-1,k}$.
- (6) If $x_i \neq y_j$, then $z_l \neq x_i$ implies $z_{1..l} \in S_{i-1,j,k}$.
- (7) If $x_i \neq y_j$, then $z_l \neq y_j$ implies $z_{1..l} \in S_{i,j-1,k}$.

Chen and Chao did not prove the correctness of the theorem, they only stated that the proof is similar to the previous theorem in the same article. We find that the algorithm in Figure 1 is not correct from two points of view. First, each answer in $S_{i,j,k-1}, k \geq 2$, is always a common subsequence of $X_{1..i}$ and $Y_{1..j}$ excluding $P_{1..k}$ as a substring. It implies that $L_{Chen}(i-1, j-1, k-1) \leq L_{Chen}(i-1, j-1, k)$ is always true. With this fact, the upper max operation in Figure 1 is no longer meaningful. Here is a very simple counterexample, $X = Y = P = ab$. $L_{Chen}(2, 2, 2) = \max\{1 + L_{Chen}(1, 1, 1), 1 + L_{Chen}(1, 1, 2)\}$. $L_{Chen}(1, 1, 1) = L_{Chen}(0, 0, 1) = 0$. $L_{Chen}(1, 1, 2) = 1 + L_{Chen}(0, 0, 2) = 1$. So $L_{Chen}(2, 2, 2) = 2$, but the correct answer should be 1. One may think that the formula was typed

with a minor mistake. So we try to modify the upper max operation into $\max\{1 + L_{Chen}(i - 1, j - 1, k - 1), L_{Chen}(i - 1, j - 1, k)\}$. However, it is still incorrect by another counterexample $X = axbc$, $Y = abyc$ and $P = ac$. The correct answer is $Z = abc$, the original formula can get the correct answer while the modified formula will get an incorrect answer ab or bc as its answer. Second, condition 3 of Theorem 1 is a wrong statement. A similar counterexample is given, $X = Y = P = abc$ and $i = j = k = 3$. We have $S_{3,3,3} = \{ab, ac, bc\}$, $S_{2,2,3} = \{ab\}$. We take $z_{1..2} = ac$ from $S_{3,3,3}$ and $l = 2$, but $z_{1..1} = a \notin S_{2,2,3} = \{ab\}$. Finally, we conclude that Chen and Chao's algorithm fails to solve the STR-EC-LCS problem.

3 A Straightforward Backtracking Algorithm

In this section, we give a simple straightforward backtracking algorithm which correctly solves the STR-EC-LCS problem. Although it requires exponential time, we will show that it can be reduced to polynomial time in the next section.

Algorithm 1 shows a backtracking algorithm for solving the STR-EC-LCS problem. We denote $\mathcal{L}_1(i, j, Tail)$ as the longest length of the common subsequence of $X_{1..i}$ and $Y_{1..j}$ such that the concatenation of the common subsequence and string $Tail$ excludes P as a substring. For each recursive step, string $Tail$ denotes the tail of the possible answer subsequence obtained by the previous step. The answer of STR-EC-LCS is obtained by invoking $\mathcal{L}_1(m, n, \phi)$ recursively, where ' ϕ ' denotes the *empty string*. Consider the example, $X = axbc$, $Y = abyc$ and $P = ac$. A part of the recursive process of invoking $\mathcal{L}_1(4, 4, \phi)$ is illustrated in Figure 2. The bold edges constitute the path corresponding to the answer, abc . Note that $\mathcal{L}_1(0, 0, 'ac') = -\infty$ since the constraint is violated, which means that node $\mathcal{L}_1(0, 0, 'ac')$ is never considered as a part of the answer.

Algorithm 1 $\mathcal{L}_1(i, j, Tail)$ {▷ Given X, Y and P }

```

1: if  $P$  is a prefix of  $Tail$  then {▷ boundary condition: violating the constraint}
2:   return  $-\infty$ 
3: else if  $i = 0$  or  $j = 0$  then {▷ normal boundary condition}
4:   return 0
5: else if  $x_i = y_j$  then {▷ matched}
6:    $New\_Tail \leftarrow x_i \oplus Tail$  {▷ new tail is formed by concatenation of  $x_i$  and  $Tail$ }
7:   return  $\max\{\mathcal{L}_1(i - 1, j - 1, New\_Tail) + 1, \mathcal{L}_1(i - 1, j, Tail), \mathcal{L}_1(i, j - 1, Tail)\}$ 
8: else {▷  $x_i \neq y_j$ , mismatched}
9:   return  $\max\{\mathcal{L}_1(i - 1, j, Tail), \mathcal{L}_1(i, j - 1, Tail)\}$ 
10: end if

```

It is not difficult to examine the correctness of Algorithm 1. The boundary conditions are handled in Lines 1-4, where Lines 3-4 deal with the normal terminations. Lines 1-2 deal with the

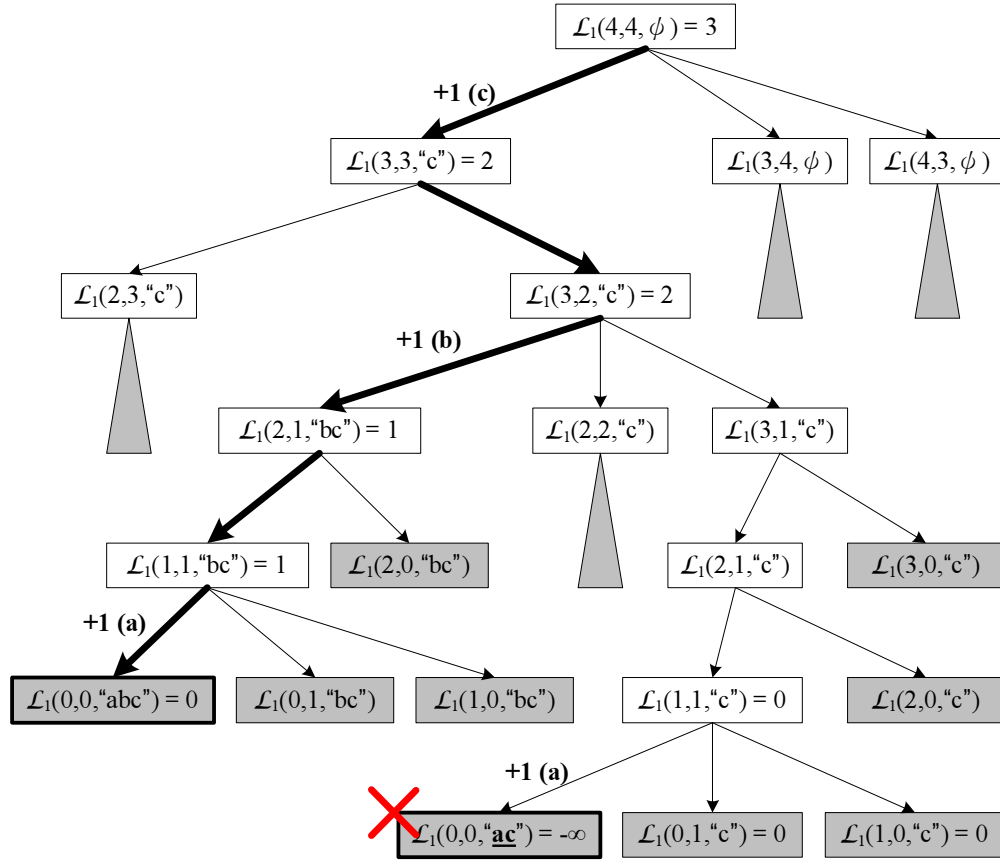


Figure 2: A part of the process of invoking $\mathcal{L}_1(4, 4, \phi)$, where $X = axbc$, $Y = abyc$ and $P = ac$. The result shows that $\mathcal{L}_1(4, 4, \phi) = 3$.

case that the constraint P has been contained in $Tail$ as a substring, or more precisely, a prefix. The constraint is violated in this case, thus $-\infty$ is returned which means that we will never consider this state as a part of the answer. Lines 5-7 consider the case that x_i and y_j are matched. In this case, we call \mathcal{L}_1 recursively to determine whether the matched symbol x_i should be chosen as a part of the answer or not. If we choose x_i as a part of the answer, x_i will be inserted in front of $Tail$ to form a new string New_Tail , where \oplus denotes the string concatenation. Lines 8-9 consider the mismatched case that either x_i or y_j is dropped.

Each state of Algorithm 1 consists of the 3-tuple $\mathcal{L}_1(i, j, Tail)$, where $0 \leq i \leq m$ and $0 \leq j \leq n$. Since the string $Tail$ is always a common subsequence of X and Y , it is evident that there are $O(mn2^n)$ states in the worst case.

Theorem 2. *Algorithm 1 can correctly solve the STR-EC-LCS problem in exponential time.*

4 Our Algorithm for the STR-EC-LCS Problem

Although Algorithm 1 requires exponential time, in this section, we will show that the required time can be reduced to be polynomial. After investigating the relationship between the string $Tail$ and the constraint P in Algorithm 1, we find that a property holds when the recursive function \mathcal{L}_1 solves the STR-EC-LCS problem. The property is given in Lemma 1.

Definition 2. (Longest prefix-suffix [10]). *Given two strings S_1 and S_2 , we denote $\mathcal{LPS}(S_1, S_2)$ as the longest prefix of S_1 that matches a suffix of S_2 .*

Lemma 1. *For a given string S excluding P as a substring, $\mathcal{L}_1(i, j, S) = \mathcal{L}_1(i, j, \mathcal{LPS}(S, P))$ with regard to any fixed pair of X and Y .*

Proof. We assume $\mathcal{L}_1(i, j, S) \neq \mathcal{L}_1(i, j, \mathcal{LPS}(S, P))$ and prove it by contradiction. For easy presentation, we denote $S = VV'$ where $V = \mathcal{LPS}(S, P)$. Besides, we denote the answer strings of $\mathcal{L}_1(i, j, S)$ and $\mathcal{L}_1(i, j, V)$ as U_1 and U_2 , respectively. In other words, $|U_1| = \mathcal{L}_1(i, j, S)$ and $|U_2| = \mathcal{L}_1(i, j, V)$. If $\mathcal{L}_1(i, j, S) \neq \mathcal{L}_1(i, j, V)$, then two possible cases should be considered as follows.

1. $\mathcal{L}_1(i, j, S) > \mathcal{L}_1(i, j, V)$. By definition, U_1S excludes P as a substring, since U_1 is the answer string of $\mathcal{L}_1(i, j, S)$. U_1V excludes P as a substring because U_1V is a prefix of U_1S . However, it implies that U_1 would be a better answer for $\mathcal{L}_1(i, j, V)$ than U_2 , which is a contradiction.
2. $\mathcal{L}_1(i, j, S) < \mathcal{L}_1(i, j, V)$. By definition, U_2V excludes P as a substring. If $U_2S = U_2VV'$ could contain P as a substring, the occurrence of P in U_2VV' should start inside U_2 and end inside V' since $S = VV'$ excludes P . Then, there would exist a suffix of P that is also a prefix of VV'

and its length is longer than $|V|$. However, $V = \mathcal{LPS}(S, P)$, which is a contradiction. Thus, U_2S excludes P as a substring. But, it implies that U_2 would be a better answer for $\mathcal{L}_1(i, j, S)$ than U_1 , which is a contradiction. \square

For example, if $P = abc$, it is clear that $\mathcal{L}_1(i, j, 'bca') = \mathcal{L}_1(i, j, 'bcddee') = \mathcal{L}_1(i, j, 'bc')$ and $\mathcal{L}_1(i, j, 'aaaaa') = \mathcal{L}_1(i, j, \phi)$. According to Lemma 1, before each call of function \mathcal{L}_1 in Algorithm 1, we can shorten the length of the third parameter *Tail* by replacing it with $\mathcal{LPS}(\textit{Tail}, P)$. Since $\mathcal{LPS}(\textit{Tail}, P)$ is always a suffix of P , it becomes feasible to record each state of the recursive process in Algorithm 1 with a 3-tuple (i, j, l) , where l denotes the length of $\mathcal{LPS}(\textit{Tail}, P)$. With this trick, we define $\mathcal{L}_2(i, j, l)$ as the longest length of the common subsequence of $X_{1..i}$ and $Y_{1..j}$ such that the answer has to exclude $P_{1..r-l}$ as a substring. Based on this new definition, we present an improved version of Algorithm 1, as shown in Algorithm 2. The length of STR-EC-LCS is obtained by calling $\mathcal{L}_2(m, n, 0)$ recursively, that is, $\mathcal{L}_2(m, n, 0) = \mathcal{L}_1(m, n, \phi)$.

Algorithm 2 $\mathcal{L}_2(i, j, l)$ {▷ Given X, Y and P }

```

1: if  $l = |P|$  then {▷ boundary condition: violating the constraint}
2:   return  $-\infty$ 
3: else if  $i = 0$  or  $j = 0$  then {▷ normal boundary condition}
4:   return 0
5: else if  $x_i = y_j$  then {▷ matched}
6:    $l' \leftarrow |\mathcal{LPS}(x_i \oplus P_{(r-l+1)..r}, P)|$ 
7:   return  $\max\{\mathcal{L}_2(i-1, j-1, l') + 1, \mathcal{L}_2(i-1, j, l), \mathcal{L}_2(i, j-1, l)\}$ 
8: else {▷  $x_i \neq y_j$ , mismatched}
9:   return  $\max\{\mathcal{L}_2(i-1, j, l), \mathcal{L}_2(i, j-1, l)\}$ 
10: end if

```

The most notable difference between Algorithm 1 and Algorithm 2 locates in Lines 5-7. In Line 6, $P_{(r-l+1)..r}$ represents the longest prefix of *Tail* which matches a suffix of P , where $|P_{(r-l+1)..r}| = l$. After a new match x_i is inserted in front of $P_{(r-l+1)..r}$, we compute the new length l' of the longest prefix of $x_i \oplus P_{(r-l+1)..r}$ which matches a suffix of P . Obviously, there are at most $O(mnr)$ states in the recursive process when $\mathcal{L}_2(m, n, 0)$ is called, which are much less than $O(mn2^n)$ states by calling $\mathcal{L}_1(m, n, \phi)$. Given two strings S_1 and S_2 , $\mathcal{LPS}(S_1, S_2)$ can be computed in $O(|S_1||S_2|)$ time by a straightforward algorithm. By utilizing such a straightforward algorithm, Algorithm 2 achieves $O(mnr) \times O(r^2) = O(mnr^3)$ running time.

To improve the efficiency of Algorithm 2, we have to find a more efficient way to compute the value of $|\mathcal{LPS}(x_i \oplus P_{(r-l+1)..r}, P)|$ in Line 6. By observing the paths in the recursive process of calling \mathcal{L}_2 , the behaviors of the third parameter l can be described as follows. When x_i is equal to $p_{(r-l)}$, we set the new prefix-suffix length l' with $l + 1$, which means that the length of matched

	1	2	3	4	5	6	7	8	9	10
S	a	b	a	b	b	a	b	a	b	c
f	0	0	1	2	0	1	2	3	4	0

(a)

$S_{1..9}$ $ababbabab$
 $S_{1..4}$ $ababbabab$
 $S_{1..2}$ $ababbabab$

(b)

Figure 3: An example for illustrating the prefix function. (a) The pattern $S = ababbababc$ and its corresponding prefix function. (b) The sliding process of $S_{1..9}$ when the matching failure occurs.

suffix of P is increased by one. On the other hand, when x_i is not equal to $p_{(r-l)}$, the length of matched suffix of P has to be shortened to the largest l' such that $P_{(r-l+1)..(r-l+l'-1)} = P_{(r-l'+2)..r}$ and $x_i = p_{(r-l'+1)}$. One can see that this process is similar to sliding a pattern when *Knuth-Morris-Pratt* (KMP) algorithm [10, 24] is used to solved the *string matching* problem. KMP algorithm searches for a pattern in a text efficiently by precomputing the *prefix function* of the given pattern [10] and then sliding the pattern efficiently when the matching failure occurs.

Definition 3. (Prefix function [10]). *Given a string S , the prefix function $f(i)$ denotes the length of the longest prefix of $S_{1..i-1}$ that matches a suffix of $S_{1..i}$.*

Figure 3(a) shows the table of the pattern $S = ababbababc$ and its corresponding prefix function. For example, $f(9) = 4$, $f(4) = 2$ and $f(2) = 0$ illustrate the sliding process of KMP algorithm [10, 24] when the matching failure occurs after $S_{1..9}$ was matched, as shown in Figure 3(b). Note that the constraint P is backward matched when \mathcal{L}_1 and \mathcal{L}_2 are called recursively. Therefore, we will precompute the prefix function of the reversed string \bar{P} of P . We should also notice that the amortized analysis of the linear-time matching, or more precisely, constant-time sliding, of KMP algorithm would be violated since there are more than one path in the recursive process of calling \mathcal{L}_2 . To ensure the pattern sliding to be executed efficiently, we define the *next function*, which achieves constant query time by precomputing a two-dimensional table.

Definition 4. (Next function). *Given a string S and a symbol $\sigma \in \Sigma$, the next function $\pi(i, \sigma)$ denotes the length of the longest prefix of $S_{1..i+1}$ that matches a suffix of $S_{1..i} \oplus \sigma$.*

Figure 4(a) shows the precomputed table for the next function of $S = ababbababc$. Figure 4(b) shows the sliding result after appending each symbol in alphabet Σ to $S_{1..9}$. For example, after appending symbol 'c' to $S_{1..9}$, the matched prefix of pattern S extends to $S_{1..10}$. However, after

	0	1	2	3	4	5	6	7	8	9	10
		<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>a</i>	1	1	3	1	3	6	1	8	1	3	-
<i>b</i>	0	2	0	4	5	0	7	0	9	5	-
<i>c</i>	0	0	0	0	0	0	0	0	0	10	-

(a)

S		$ababbababc$
$S_{1..9}$		$ababbabab$
$S_{1..9} \oplus 'a'$	$ababbababa$	
$S_{1..9} \oplus 'b'$	$ababbababb$	
$S_{1..9} \oplus 'c'$		$ababbababc$

(b)

Figure 4: An example for illustrating the next function. (a) The precomputed table for the next function π of the pattern $S = ababbababc$. (b) The sliding results corresponding to $\pi(9, 'a')$, $\pi(9, 'b')$ and $\pi(9, 'c')$.

appending symbol ' a ' or ' b ' to $S_{1..9}$, the length of the matched prefix is shortened to $\pi(9, 'a') = 3$ or $\pi(9, 'b') = 5$, respectively.

Algorithm 3 shows how to compute the table of next function π when a pattern S and its precomputed prefix function f are given. Clearly, each element can be computed in constant time and the whole table can be built in $O(|S||\Sigma|)$ time. Now go back to our aim, to improve the efficiency of Algorithm 2. One can easily see that the value of $|\mathcal{LPS}(x_i \oplus P_{(r-l+1)..r}, P)|$ in Line 6 of Algorithm 2 can be answered in constant time after the next function π of the reversed string \bar{P} is precomputed. That is, $|\mathcal{LPS}(x_i \oplus P_{(r-l+1)..r}, P)| = \pi(l, x_i)$. We summarize the result in the following theorem.

Algorithm 3 Next-Function $\{\triangleright$ Given $S\}$

```

1: for  $\sigma \in \Sigma$  and  $\sigma \neq s_1$  do
2:    $\pi(0, \sigma) \leftarrow 0$ 
3: end for
4:  $\pi(0, s_1) \leftarrow 1$ 
5: for  $i = 1$  to  $|S| - 1$  do
6:   for  $\sigma \in \Sigma$  do
7:     if  $\sigma = s_{i+1}$  then
8:        $\pi(i, \sigma) \leftarrow i + 1$ 
9:     else
10:       $\pi(i, \sigma) \leftarrow \pi(f(i), \sigma)$ 
11:    end if
12:  end for
13: end for

```

Theorem 3. Algorithm 2 solves the STR-EC-LCS problem in $O(mnr)$ time.

Proof. The correctness of Algorithm 2 has been discussed above. Now we consider its time complexity. The computation of the next function π of \bar{P} takes $O(r|\Sigma|)$ time, and it is clear that $|\Sigma| = O(m + n + r)$. Remind that we have assumed $r \leq n \leq m$ in the beginning of this paper. Therefore, the overall time complexity is simplified from $O(mnr + r|\Sigma|)$ to $O(mnr)$. \square

5 Our Algorithm for the Multiple STR-EC-LCS Problem

In this section, we will show that the *multiple STR-EC-LCS* problem, in which an arbitrary number of constraints should be excluded, can be solved in polynomial time. The LCS problem of multiple sequences [25], the CLCS problem with multiple constraints (*multiple SEQ-IC-LCS*) [17], and the *multiple SEQ-EC-LCS* (restricted LCS) problem [16] have been proved to be NP-hard. In addition, Chen and Chao [8] said that the multiple STR-IC-LCS, SEQ-EC-LCS and STR-EC-LCS problems can be proved to be NP-hard, but they did not give the proof. They also presented exponential-time algorithms for solving the latter two problems by simply extending their algorithms for one constraint. We will extend Algorithm 2 to solve the multiple STR-EC-LCS problem in polynomial time, and thus it also shows that the multiple STR-EC-LCS problem is not NP-hard. The formal definition of the multiple STR-EC-LCS problem is given as follows.

Definition 5. (Multiple STR-EC-LCS problem [8]). *Given two sequences X and Y of lengths m and n , respectively, and a set of k constraints $P = \{P_1, P_2, \dots, P_k\}$, the multiple STR-EC-LCS problem is to find the longest common subsequence Z of X and Y excluding each constraint $P_i \in P$ as a substring.*

In the previous section, we employ the KMP string matching algorithm as a building block of Algorithm 2. Now we review *Aho-Corasick algorithm* [1], an elegant *dictionary matching* algorithm, which is based on the idea similar to KMP algorithm. Here, a *dictionary* represents a set of patterns. Therefore, the dictionary matching problem is a more general form of the string matching problem. Aho-Corasick algorithm is implemented as a finite state machine, called *Aho-Corasick automaton*, which consists of three functions, a *goto* function, an *output* function and a *failure* function. For a constant alphabet size, in the worst case, the construction time of Aho-Corasick automaton is linear to the total length of the given patterns and each of the state transitions requires constant time [1]. When the alphabet size varies, the construction time is still expected linear if the dynamic perfect hashing technique is applied [14].

Aho and Corasick gave an example as shown in Figure 5. In Figure 5(a), the *goto* function is represented as the solid edges of a *trie* and the *output* function indicates when the matches occur

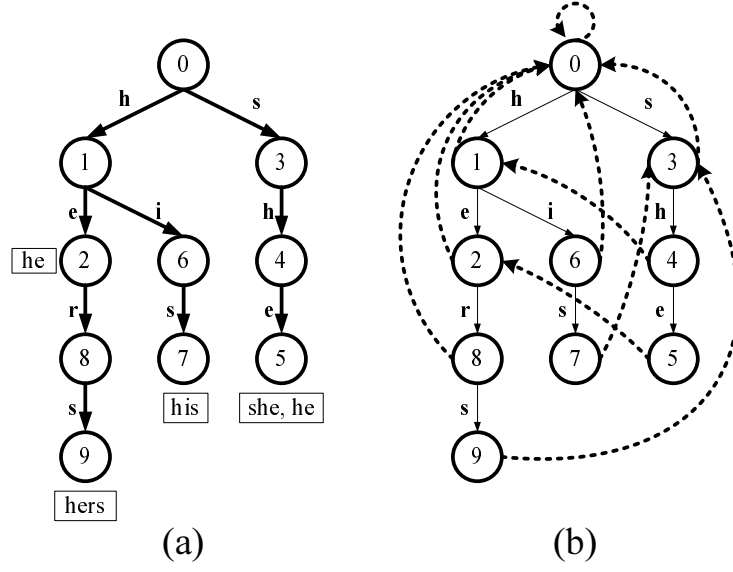


Figure 5: Aho and Corasick's example [1]. (a) The *goto* function and *output* function. (b) The *failure* function.

and which words are output. For example, the outputs of nodes 2 and 5 are $\{\text{he}\}$ and $\{\text{she, he}\}$, respectively. The *failure* function indicates which state to go if there is no symbol to be further matched. It is represented by the dashed edges in Figure 5(b). The *failure list* of a given node is the ordered list of the nodes which locate on the path to the root via dashed edges. For example, both nodes 7 and 9 have the same failure list $\{3 \rightarrow 0\}$, and the failure list of node 8 is $\{0\}$.

The main difference between KMP algorithm and Aho-Corasick algorithm is the extension of the *goto* function, which forms a list in KMP algorithm and forms a trie in Aho-Corasick algorithm. This difference makes the movements between the states more complex in Aho-Corasick algorithm. In the previous section, we use the next function to identify the length of the matched prefix of the pattern after a given symbol is appended. For Aho-Corasick algorithm, the next function will identify the new state rather than the new length. We define the *Aho-Corasick-Next* function as follows.

Definition 6. (Aho-Corasick-Next function). *Given a goto function, a failure function, a symbol $\sigma \in \Sigma$ and a current state q , Aho-Corasick-Next function $\delta(q, \sigma)$ denotes the destination of the first node in q 's failure list which has a goto function of symbol σ . If there exists no such node in the failure list, the function returns the root, i.e. node 0.*

Figure 6 shows the Aho-Corasick-Next function corresponding to the example in Figure 5. We take node 4 as an example. When we receive a symbol 'e', the new state is node 5. When a symbol 'i' is received, it fails to node 1 and the new state becomes node 6. When symbols 'h' and 's' are

	0	1	2	3	4	5	6	7	8	9
e	0	2	0	0	5	0	0	0	0	0
h	1	1	1	4	1	1	1	4	1	4
i	0	6	0	0	6	0	0	0	0	0
r	0	0	8	0	0	8	0	0	0	0
s	3	3	3	3	3	3	7	3	9	3

Figure 6: The Aho-Corasick-Next function δ corresponding to the example in Figure 5.

received, they fail to node 0 and the new states become node 1 and node 3, respectively. For other symbols, the new state is the root. It is easy to see that each element of Aho-Corasick-Next function can be computed in constant time by using an algorithm similar to Algorithm 3.

To solve the multiple STR-EC-LCS problem, we first precompute the Aho-Corasick-Next function δ with patterns $\{\overline{P_1}, \overline{P_2}, \dots, \overline{P_k}\}$, the reversed strings of the constraints. Then this problem can be solved by Algorithm 4, which is slightly modified from Algorithm 2. To get the length of the multiple STR-EC-LCS problem, we invoke $\mathcal{L}_3(m, n, 0)$ recursively. We summarize the result in Theorem 4.

Algorithm 4 $\mathcal{L}_3(i, j, q)$ {▷ Given X, Y and δ }

```

1: if  $output[q] \neq \phi$  then {▷ boundary condition: violating one or more constraints}
2:   return  $-\infty$ 
3: else if  $i = 0$  or  $j = 0$  then {▷ normal boundary condition}
4:   return 0
5: else if  $x_i = y_j$  then {▷ matched}
6:    $q' \leftarrow \delta(q, x_i)$ 
7:   return  $\max\{\mathcal{L}_3(i-1, j-1, q') + 1, \mathcal{L}_3(i-1, j, q), \mathcal{L}_3(i, j-1, q)\}$ 
8: else {▷  $x_i \neq y_j$ , mismatched}
9:   return  $\max\{\mathcal{L}_3(i-1, j, q), \mathcal{L}_3(i, j-1, q)\}$ 
10: end if

```

Theorem 4. *Algorithm 4 solves the multiple STR-EC-LCS problem in $O(mnR)$ time, where $R = \sum_{i=1}^k |P_i|$.*

Proof. We analyze the time complexity of Algorithm 4 as follows. The precomputation of Aho-Corasick automaton and the Aho-Corasick-Next function δ of the reversed constraints $\{\overline{P_1}, \overline{P_2}, \dots, \overline{P_k}\}$ takes $O(R|\Sigma|)$ time by using a dynamic programming algorithm similar to Algorithm 3. There are at most $O(R)$ states in the Aho-Corasick automaton and thus the time required for the dynamic programming part is $O(mnR)$. The overall time complexity is $O(mnR) + O(R|\Sigma|) = O(mnR)$. □

To show the degree of similarity between two strings or files, the answer to the length of the

STR-EC-LCS is enough. However, for some practical utilization, one may ask to show the answer string of the problem. After the value of each cell in the \mathcal{L}_3 lattice has been obtained, we can invoke Algorithm 5 with calling $LCS_3(m, n, 0)$ to get the answer string. In fact, Algorithm 5 is similar to the traditional LCS tracing back technique with the slight modification on calling $\delta(q, x_i)$ during a match. The time complexity of Algorithm 5 is obviously $O(m + n)$, since there are at most $m + n$ recursive calls, each taking constant time.

Algorithm 5 $LCS_3(i, j, q)$ {▷ Given X, Y, δ and \mathcal{L}_3 }

```

1: if  $\mathcal{L}_3(i, j, q) = 0$  then {▷ boundary condition: end of tracing}
2:   return  $\phi$ 
3: else if  $x_i = y_j$  then
4:    $q' \leftarrow \delta(q, x_i)$ 
5:   if  $\mathcal{L}_3(i, j, q) = \mathcal{L}_3(i - 1, j - 1, q') + 1$  then {▷ using the match}
6:     return  $LCS_3(i - 1, j - 1, q') \oplus x_i$ 
7:   end if
8: end if
9: {▷ all remainders}
10: if  $\mathcal{L}_3(i, j, q) = \mathcal{L}_3(i - 1, j, q)$  then
11:   return  $LCS_3(i - 1, j, q)$ 
12: else
13:   return  $LCS_3(i, j - 1, q)$ 
14: end if

```

6 Concluding Remarks

In this paper, we study the STR-EC-LCS problems of single constraint and multiple constraints. For the STR-EC-LCS problem with single constraint, we present an algorithm which corrects Chen and Chao's algorithm with the same time complexity. For the STR-EC-LCS problem with multiple constraints, we show that even the number of constraints increases, the problem can still be solved efficiently in polynomial time, which implies that the problem is not NP-hard. Our algorithm for the multiple STR-EC-LCS problem requires only $O(mnR)$ time, where R is the total length of the constraints. To get the answer string, we also present a simple tracing algorithm, which is similar to the traditional tracing back technique.

Another similar variant was also addressed by Chen and Chao [8], the *multiple STR-IC-LCS* problem, in which the resulting sequence has to contain all constraints as its substrings. In fact, it is not difficult to see that this problem is complementary to the multiple STR-EC-LCS problem, thus we can also employ Aho-Corasick automaton to solve the multiple STR-IC-LCS problem. Two key differences should be noticed when Aho-Corasick automaton is applied. First, in each recursive step of the multiple STR-EC-LCS algorithm, if any constraint is violated, we return $-\infty$ to prevent

the current step to be considered. On the other hand, in each recursive step of the multiple STR-IC-LCS algorithm, we shall return $-\infty$ when the normal boundary conditions without keeping all constraints occur. Second, obtained from the first, each state in Aho-Corasick automaton for solving the multiple STR-IC-LCS problem will be split into 2^k new states by attaching k flags to denote the combinations which constraints have been kept. However, the number of states is still exponential and the overall required time is $O(2^k \cdot mnR)$.

For the STR-IC-LCS problem with single constraint, recently Tseng *et al.* [30], presented an algorithm with $O(mn)$ time to solve it. Furthermore, for the multiple STR-IC-LCS problem, they restricted that the constraints are given in some order. With this restriction, they proposed an algorithm with $O(mnR)$ time for solving it. If no restriction is applied on the multiple constraints, there is still no efficient algorithm.

References

- [1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, Vol. 18, No. 6, pp. 333–340, 1975.
- [2] H. Y. Ann, C. B. Yang, Y. H. Peng, and B. C. Liaw, "Efficient algorithms for the block edit problems," *Information and Computation*, Vol. 208, No. 3, pp. 221–229, 2010.
- [3] H. Y. Ann, C. B. Yang, C. T. Tseng, and C. Y. Hor, "A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings," *Information Processing Letters*, Vol. 108, pp. 360–364, 2008.
- [4] H. Y. Ann, C. B. Yang, C. T. Tseng, and C. Y. Hor, "Fast algorithms for computing the constrained LCS of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, 2012.
- [5] A. Apostolico and C. Guerra, "The longest common subsequences problem revisited," *Algorithmica*, Vol. 2, No. 1-4, pp. 315–336, 1987.
- [6] A. N. Arslan and Ö. Egecioğlu, "Algorithms for the constrained longest common subsequence problems," *International Journal of Foundations Computer Science*, Vol. 16, No. 6, pp. 1099–1109, 2005.
- [7] C. Blum, M. J. Blesa, and M. López-Ibáñez, "Beam search for the longest common subsequence problem," *Computers and Operations Research*, Vol. 36, No. 12, pp. 3178–3186, 2009.

- [8] Y. C. Chen and K. M. Chao, “On the generalized constrained longest common subsequence problems,” *Journal of Combinatorial Optimization*, Vol. 21, No. 3, pp. 383–392, 2011.
- [9] F. Y. L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, and S. K. Kim, “A simple algorithm for the constrained sequence problems,” *Information Processing Letters*, Vol. 90, No. 4, pp. 175–179, 2004.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.
- [11] S. Deorowicz, “Bit-parallel algorithm for the constrained longest common subsequence problem,” *Fundamenta Informaticae*, Vol. 99, No. 4, pp. 409–433, Dec. 2010.
- [12] S. Deorowicz, “Quadratic-time algorithm for a string constrained LCS problem,” *Information Processing Letters*, Vol. 112, No. 11, pp. 423–426, June 2012.
- [13] S. Deorowicz and J. Obstoj, “Constrained longest common subsequence computing algorithms in practice,” *Computing and Informatics*, Vol. 29, No. 3, pp. 427–445, 2010.
- [14] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan, “Dynamic perfect hashing: Upper and lower bounds,” *SIAM Journal on Computing*, Vol. 123, No. 4, pp. 738–761, 1994.
- [15] E. Farhana, J. Ferdous, T. Moosa, and M. S. Rahman, “Finite automata based algorithms for the generalized constrained longest common subsequence problems,” *Proceedings of the 17th international conference on String processing and information retrieval, SPIRE’10*, Los Cabos, Mexico, pp. 243–249, 2010.
- [16] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein, “Restricted LCS,” *Proceedings of the 17th international conference on String processing and information retrieval, SPIRE’10*, Los Cabos, Mexico, pp. 250–257, 2010.
- [17] Z. Gotthilf, D. Hermelin, and M. Lewenstein, “Constrained LCS: hardness and approximation,” *Proceedings of the 19th annual symposium on Combinatorial Pattern Matching, CPM ’08*, Pisa, Italy, pp. 255–262, 2008.
- [18] D. S. Hirschberg, “Algorithms for the longest common subsequence problem,” *Journal of the ACM*, Vol. 24, No. 4, pp. 664–675, 1977.

- [19] J. W. Hunt and T. G. Szymanski, “A fast algorithm for computing longest common subsequences,” *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.
- [20] C. S. Iliopoulos and M. S. Rahman, “New efficient algorithms for the LCS and constrained LCS problems,” *Information Processing Letters*, Vol. 106, No. 1, pp. 13–18, 2008.
- [21] C. Iliopoulos, M. S. Rahman, M. Voráček, and L. Vagner, “Finite automata based algorithms on subsequences and supersequences of degenerate strings,” *Journal of Discrete Algorithms*, Vol. 8, No. 2, pp. 117–130, June 2010.
- [22] C. S. Iliopoulos and M. S. Rahman, “A new efficient algorithm for computing the longest common subsequence,” *Theoretical Computer Science*, Vol. 45, No. 2, pp. 355–371, June 2009.
- [23] C. S. Iliopoulos, M. S. Rahman, and W. Rytter, “Algorithms for two versions of LCS problem for indeterminate strings,” *Journal of Combinatorial Mathematics and Combinatorial Computing*, Vol. 71, pp. 155–172, Nov. 2009.
- [24] D. E. Knuth, J. H. Morris Jr., and V. Pratt, “Fast pattern matching in strings,” *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323–350, 1977.
- [25] D. Maier, “The complexity of some problems on subsequences and supersequences,” *Journal of the ACM*, Vol. 25, pp. 322–336, 1978.
- [26] Y. H. Peng, C. B. Yang, K. S. Huang, and K. T. Tseng, “An algorithm and applications to sequence alignment with weighted constraints,” *International Journal of Foundations of Computer Science*, Vol. 21, No. 1, pp. 51–59, 2010.
- [27] S. J. Shyu and C. Y. Tsai, “Finding the longest common subsequence for multiple biological sequences by ant colony optimization,” *Computers and Operations Research*, Vol. 36, No. 1, pp. 73–91, 2009.
- [28] C. Y. Tang, C. L. Lu, M. D. T. Chang, Y. T. Tsai, Y. J. Sun, K. M. Chao, J. M. Chang, Y. H. Chiou, C. M. Wu, H. T. Chang, and W. I. Chou, “Constrained multiple sequence alignment tool development and its application to RNase family alignment,” *Journal of Bioinformatics and Computational Biology*, Vol. 1, pp. 267–287, 2003.
- [29] Y. T. Tsai, “The constrained longest common subsequence problem,” *Information Processing Letters*, Vol. 88, No. 4, pp. 173–176, 2003.

- [30] C. T. Tseng, C. B. Yang, and H. Y. Ann, “Efficient algorithms for the longest common subsequence problem with sequential substring constraints,” *to appear in Journal of Complexity*.
- [31] R. Wagner and M. Fischer, “The string-to-string correction problem,” *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.
- [32] C. B. Yang and R. C. T. Lee, “Systolic algorithm for the longest common subsequence problem,” *Journal of the Chinese Institute of Engineers*, Vol. 10, No. 6, pp. 691–699, 1987.