Efficient Polynomial-Time Algorithms for Variants of the Multiple Constrained LCS Problem

Hsing-Yen Ann
National Center for High-Performance Computing
Tainan 74147, Taiwan

Chang-Biau Yang* and Chiou-Ting Tseng
Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung 80424, Taiwan
*Email: cbyang@cse.nsysu.edu.tw

***Abstract*—In this paper, we revisit a recent variant of the longest common subsequence problem, the string-excluding constrained LCS (STR-EC-LCS) problem, which was first addressed by Chen and Chao [Journal of Combinatorial Optimization, 21(3), 2011]. Given two sequences X and Y of lengths n and m, 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. This problem cannot be correctly solved by the previously proposed algorithm. Thus, we give a correct algorithm with $O(nmr)$ time to solve it.**

Keywords— design of algorithms; longest common subsequence; constrained LCS; NP-hardness; biological application

I. Introduction

Computing the similarity of two sequences is one of the most important fundamentals in computer field. Given a sequence X of length n over some fixed alphabet Σ, a *subsequence* of X is formed by deleting zero or more symbols from X arbitrarily. We use $X = x_1x_2\cdots x_n$ to denote the list of symbols in X. A *substring* $X_{i..j} = x_ix_{i+1}\cdots x_j$ of X consists of successive symbols in X, where $1 \le i \le j \le n$. A substring $X_{i..j}$ is called a prefix or a suffix of X if $i = 1$ or $j = n$, respectively. A sequence is called a *common subsequence* 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 out a subsequence of X and Y where its length is 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 was proposed by Wagner and Fischer [21] which solves the LCS problem in quadratic time with the *dynamic programming* (DP) technique. Other advanced algorithms have also been proposed for the past decades [1, 2, 11, 12, 22]. When the number of input sequences is not fixed, finding the LCS of the multiple sequences has been proved as NP-hard [15], and therefore some approximate and heuristic algorithms have been proposed [5, 17].

Applying the constraints to the LCS problem is meaningful for some biological applications [18]. Therefore, the *constrained LCS* (CLCS) problem, a recent variant of the LCS problem which was first addressed by Tsai [19], has gained much attention. Given two input sequences X and Y of lengths n and m, 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 maximized. In the following, without loss of generality, we may assume that $r \le m \le n$. The most referred algorithms were presented independently [4, 7] which solve the CLCS problem based on the DP technique in $O(nmr)$ time and space. Some improved algorithms have also been proposed. Iliopoulos and Rahman [13] consider the matched pairs and employed a special data structure, *van Emde Boas tree* [20], to solve this problem. Ann *et al.* [3] considered the sequences which are in *run-length encoding* (RLE) format, a linear-time compressing technique, and greatly reduce the number of elements required to be evaluated in the DP lattice. Moreover, Peng *et al.* [16] 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 proven to be NP-hard [10].

Recently, Gotthilf *et al.* [9] 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 [6] 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 n and m, 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 (or excluding) P as a subsequence (or substring), respectively. Table I shows the four problems. Evidently, the original CLCS problem addressed by Tsai [19] is specified as SEQ-IC-LCS, and the restricted LCS problem addressed by Gotthilf *et al.* [9] is specified as SEQ-EC-LCS. For the problems in Table I except SEQ-IC-LCS, Chen and Chao proposed the corresponding algorithms which are all in $O(nmr)$ time. However, as we will show soon, their algorithm for STR-EC-LCS cannot work correctly.

In this paper, we focus on the STR-EC-LCS problem and its variants. In Section II, we review Chen and Chao's algorithm and give a counter example which breaks their algorithm. In Section III, a correct straightforward backtracking algorithm is given, though it runs in exponential time. Then, we propose an efficient algorithm in Section IV. And finally, the conclusion goes in Section V.

TABLE I
THE FOUR GC-LCS PROBLEMS DEFINED BY CHEN AND CHAO [6] AND THE TIME COMPLEXITIES OF THEIR PROPOSED METHODS.

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

II. A COUNTER EXAMPLE TO CHEN AND CHAO'S ALGORITHM

Definition 1. (STR-EC-LCS problem [6]). *Given two input sequences X and Y of lengths n and m, respectively, and a constraint string P of length r, the STR-EC-LCS (string excluding) problem is to find the longest common subsequence Z of X and Y excluding P as a substring.*

For the STR-EC-LCS problem, Chen and Chao [6] proposed an dynamic programming algorithm with $O(nmr)$ time to solve it. 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.

Now we provide a simple counter example, $X = axbc$, $Y = abyc$ and $P = ac$, which breaks Chen and Chao's algorithm for the STR-EC-LCS problem. Evidently, the LCS of X and Y, abc, does not contain ac as a substring. That is to say, abc excludes the constraint P. However, Chen and Chao's algorithm would return the answer $L_{Chen}(4, 4, 2) = 2$, where the corresponding result sequence is ab or bc. Part of the recursive process of calling $L_{Chen}(4, 4, 2)$ is illustrated in Figure 2. One can see that their answer is not correct and the length is shorter than $|abc| = 3$. It concludes that Chen and Chao's algorithm fails to solve the STR-EC-LCS problem.

III. A STRAIGHTFORWARD BACKTRACKING ALGORITHM

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

Algorithm 1 shows a backtracking algorithm for solving the STR-EC-LCS problem. We denote $\mathcal{L}_1(i, j, \textit{Tail})$ 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 sequence obtained by the previous step. The answer of STR-EC-LCS is obtained by invoking $\mathcal{L}_1(n, m, \phi)$ recursively, where 'ϕ' denotes the *empty string*. We consider the example, $X = axbc$, $Y = abyc$ and $P = ac$, which breaks Chen and Chao's algorithm for the STR-EC-LCS problem in the previous section. Part of the recursive process of invoking $\mathcal{L}_1(4, 4, \phi)$ is illustrated in Figure 3. The bold edges indicate the path corresponding to the answer, abc. We also note that $\mathcal{L}_1(0, 0, \text{'}ac\text{'}) = -\infty$ since the constraint is violated, which means that node $\mathcal{L}_1(0, 0, \text{'}ac\text{'})$ is never considered as a part of the answer.

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 case that the constraint P has been contained in *Tail* as a substring, or more precisely, a prefix. The constraint has been violated in this case, thus $-\infty$ is returned which means that we would 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 case of mismatch in which either x_i or y_j will be dropped.

It is evident that even if we use the dynamic programming technique to reduce the total number of states in the recursive process, there are still $O(nm|\Sigma|^m)$ states in the worst case.

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

IV. 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 and thus an efficient algorithm with polynomial time is proposed. After investigating the relationship between the string *Tail* and the constraint P in Algorithm 1, we find that a property holds when solving STR-EC-LCS with the recursive function $\mathcal{L}_1$, as shown in Lemma 1.

Definition 2. (Longest prefix-suffix). *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, $\mathcal{L}_1(i, j, S) = \mathcal{L}_1(i, j, \mathcal{LPS}(S, P))$.*

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 ease of presentation, we denote $S = VV'$ where $V = \mathcal{LPS}(S, P)$. Besides, we denote the corresponding 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 would be considered as follows.

1) $\mathcal{L}_1(i, j, S) > \mathcal{L}_1(i, j, V)$. It means that U_1S excludes P as a substring and U_1V contains P as a substring, otherwise U_1 can be concatenated with V and gains a better length $|U_1| > \mathcal{L}_1(i, j, V)$. However, U_1V is a prefix of U_1S. If U_1V contains P as a substring, U_1S would also contain P as a substring, which is a contradiction.
2) $\mathcal{L}_1(i, j, S) < \mathcal{L}_1(i, j, V)$. It means that $U_2S = U_2VV'$ contains P as a substring and U_2V excludes P as a substring. By definition, $S = VV'$ excludes P, the occurrence of P in U_2VV' should start inside U_2 and end inside V'. Thus, there would exist a suffix of P that is also a prefix of VV' and its length is longer than $|V|$.

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

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

Fig. 1. Chen and Chao's recurrence formula [6] for the STR-EC-LCS problem.

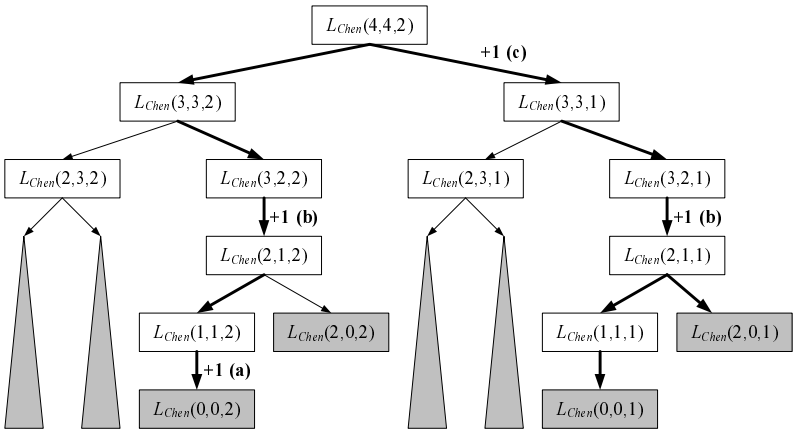

Fig. 2. Part of the process of Chen and Chao's recurrence formula, where $X = axbc$, $Y = abyc$ and $P = ac$. The result sequences are ab and bc.

Algorithm 1 $\mathcal{L}_1(i, j, \mathit{Tail})$

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** $\max\{\mathcal{L}_1(i-1, j-1, \mathit{New\_Tail}) + 1, \mathcal{L}_1(i-1, j, \mathit{Tail}), \mathcal{L}_1(i, j-1, \mathit{Tail})\}$
8: **else** {▷ $x_i \neq y_j$, mismatched}
9: **return** $\max\{\mathcal{L}_1(i-1, j, \mathit{Tail}), \mathcal{L}_1(i, j-1, \mathit{Tail})\}$
10: **end if**

However, $V = \mathcal{LPS}(S, P)$, which is a contradiction. □

For example, if $P = abc$, it is clear that $\mathcal{L}_1(i, j, \text{'}bca\text{'}) = \mathcal{L}_1(i, j, \text{'}bcdddee\text{'}) = \mathcal{L}_1(i, j, \text{'}bc\text{'})$ and $\mathcal{L}_1(i, j, \text{'}aaaaa\text{'}) = \mathcal{L}_1(i, j, \phi)$. According to Lemma 1, before each call of function $\mathcal{L}_1$ in Algorithm 1, we may shorten the length of the third parameter *Tail* by replacing it with $\mathcal{LPS}(\mathit{Tail}, P)$. Since $\mathcal{LPS}(\mathit{Tail}, P)$ 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 $\mathcal{LPS}(\mathit{Tail}, P)$. With this trick, we present an improved version of Algorithm 1 as shown in Algorithm 2. The answer of STR-EC-LCS is got by calling $\mathcal{L}_2(n, m, 0)$ recursively, that is, $\mathcal{L}_2(n, m, 0) = \mathcal{L}_1(n, m, \phi)$.

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. After x_i is inserted in front of $P_{(r-l+1)..r}$, we will 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 $O(nmr)$ states in the recursive process when calling $\mathcal{L}_2(n, m, 0)$, which are much less than $O(nm|\Sigma|^m)$ states when calling $\mathcal{L}_1(n, m, \phi)$. 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 $O(nmr) \times O(r^2) = O(nmr^3)$ running time.

To improve the efficiency of Algorithm 2, we have to find a more efficient way to compute the value $|\mathcal{LPS}(x_i \oplus$

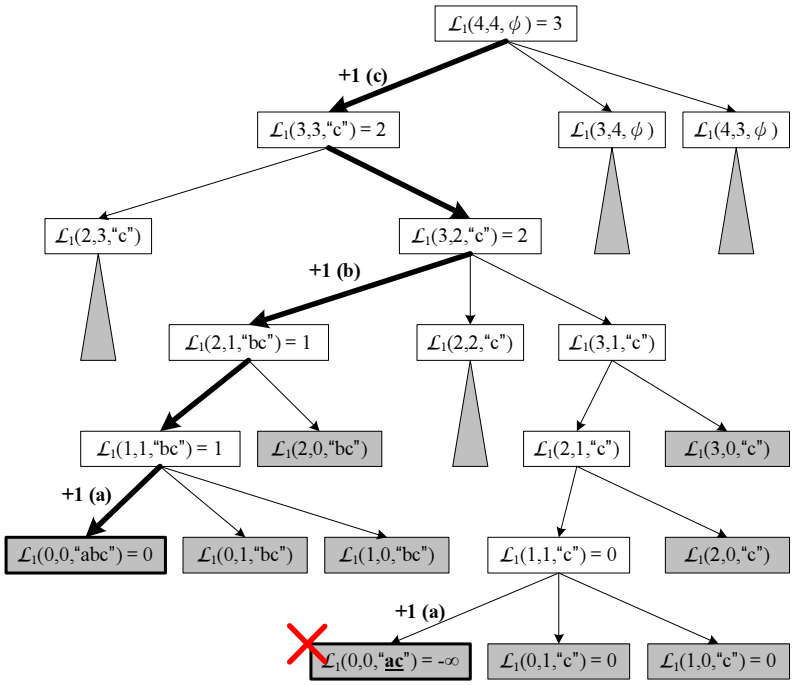

Fig. 3. Part of the process of invoking $\mathcal{L}_1(4,4,\phi)$, where $X = axbc$, $Y = abyc$ and $P = ac$. The result shows that $\mathcal{L}_1(4,4,\phi) = 3$.

Algorithm 2 $\mathcal{L}_2(i,j,l)$

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**

$P_{(r-l+1)..r}, P)|$ 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 suffix of P increases 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 slide a pattern when executing *Knuth-Morris-Pratt* (*KMP*) *algorithm* [8, 14], a famous *string matching* algorithm. KMP algorithm searches for a pattern in a text efficiently by precomputing the *prefix function* of the given pattern [8] and then sliding the pattern efficiently when the matching failure occurs.

Definition 3. (Prefix function [8]). *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 4(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 [8, 14] when the matching failure occurs after $S_{1..9}$ was matched, as shown in Figure 4(b). We should notice 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 $\overline{P}$ 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 sliding of the pattern to be executed efficiently, we define the *next function* as follows, 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 $S_{1..i} \oplus \sigma$.*

	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)

$S_{1..9}$ $ababbabab|c$
$S_{1..4}$ $abab\sout{babab}c$
$S_{1..2}$ $ab\sout{abbabab}c$

(b)

Fig. 4. 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.

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

(a)

S $|ababbababc$
$S_{1..9}$ $|ababbabab$
$S_{1..9} \oplus$ 'a' $\sout{ababbab}|aba$
$S_{1..9} \oplus$ 'b' $\sout{ababb}|ababb$
$S_{1..9} \oplus$ 'c' $|ababbababc$

(b)

Fig. 5. 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 $\pi(9, \text{'}a\text{'})$, $\pi(9, \text{'}b\text{'})$ and $\pi(9, \text{'}c\text{'})$.

Figure 5(a) shows the precomputed table for the next function of $S = ababbababc$. Figure 5(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 appending symbol 'a' or 'b' to $S_{1..9}$, the length of the matched prefix is shortened to $\pi(9, \text{'}a\text{'}) = 3$ or $\pi(9, \text{'}b\text{'}) = 5$, 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 computed 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 $|\mathcal{LPS}(x_i \oplus P_{(r-l+1)..r}, P)|$ in Line 6 of Algorithm 2 can be answered in constant time after precomputing the next function π of the reversed string $\overline{P}$. 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.

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

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

Algorithm 3 Compute-Next-Function

```
1: for σ ∈ Σ and σ ≠ s_1 do
2:     π(0, σ) ⇐ 0
3: end for
4: π(0, s_1) ⇐ 1
5: for i = 1 to |S| − 1 do
6:     for σ ∈ Σ do
7:         if σ = s_{i+1} then
8:             π(i, σ) ⇐ i + 1
9:         else
10:            π(i, σ) ⇐ π(f(i), σ)
11:        end if
12:    end for
13: end for
```

V. Concluding Remarks

In this paper, we study the STR-EC-LCS problem with single constraint. We give an algorithm which corrects Chen and Chao's algorithm with the same time complexity. For the STR-EC-LCS problem with multiple constraints, Chen and Chao [6] claimed that the problem is NP-hard. However, the claim may not be correct, since we are developing an efficient algorithm, which requires only polynomial time. The new algorithm is somewhat complicated, so we omit its description here.

Another similar variant was also addressed by Chen and Chao [6], 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. We will also try to solve this problem in the future.

References

[1] 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.

[2] 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.

[3] 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," *Proc. of the 2009 International Conference on Bioinformatics and Computational Biology (BIOCOMP'09), Las Vegas, USA*, pp. 646–649, 2009.

[4] A. N. Arslan and Ö. Eğecioğlu, "Algorithms for the constrained longest common subsequence problems," *In-*

ternational Journal of Foundations Computer Science, Vol. 16, No. 6, pp. 1099–1109, 2005.

[5] 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.

[6] 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.

[7] 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.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, second ed., 2001.

[9] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein, "Restricted LCS," *String Processing and Information Retrieval (SPIRE), Lecture Notes in Computer Science*, Vol. 6393, pp. 250–257, 2010.

[10] Z. Gotthilf, D. Hermelin, and M. Lewenstein, "Constrained LCS: hardness and approximation," *Combinatorial Pattern Matching, 19th Annual Symposium (CPM2008), Pisa, Italy*, pp. 255–262, 2008.

[11] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, Vol. 24, No. 4, pp. 664–675, 1977.

[12] 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.

[13] 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.

[14] 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.

[15] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, Vol. 25, pp. 322–336, 1978.

[16] 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.

[17] 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.

[18] 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.

[19] Y. T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, Vol. 88, No. 4, pp. 173–176, 2003.

[20] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Information Processing Letters*, Vol. 6, No. 3, pp. 80–82, 1977.

[21] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.

[22] 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.