Efficient Algorithms for the Longest Common Subsequence Problem with Sequential Substring Constraints

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

***Abstract*—In this paper, we generalize the inclusion constrained longest common subsequence (CLCS) problem to the hybrid CLCS problem which is the combination of the sequence inclusion CLCS and the string inclusion CLCS, called the *sequential substring constrained longest common subsequence* (SSCLCS) problem. In the SSCLCS problem, we are given two strings A and B of lengths m and n, respectively, formed by alphabet Σ and a constraint sequence C formed by ordered strings $(C^1, C^2, C^3, \cdots, C^l)$ with total length r. We are to find the longest common subsequence D of A and B containing $C^1, C^2, C^3, \cdots, C^l$ as substrings and the order of C's are retained. This problem have two variants that the strings in C may or may not overlap. We proposed algorithms with $O(mnl + (m+n)(|\Sigma|+r))$ and $O(mnr + (m+n)|\Sigma|)$ time for the two variants of the problem. For the special case with one or two constraints, our algorithms runs in $O(mn+(m+n)(|\Sigma|+r))$ and $O(mnr + (m+n)|\Sigma|)$ time, which are an order faster than the algorithm proposed by Chen and Chao [1].**

***Keywords*-constrained longest common subsequence; hybrid; sequential substring;**

I. Introduction

Given two strings $A = a_1a_2a_3\cdots a_m$ and $B = b_1b_2b_3\cdots b_n$, the *longest common subsequence* (LCS) problem is to find the longest common part of A and B by deleting zero or more characters from A and B. It was first proposed in 1974 by Wagner and Fischer [2]. Much ink has been spent on this topic in the past decades [3–5], and lots of variants to the LCS problem have also been proposed, such as the *mosaic LCS* problem [6], the *merged LCS* problem [7, 8], the *cyclic string correction problem* [9] and the *block edit* problem [10].

Given two strings A and B and a constraint sequence C with length m, n and r respectively, the *constrained longest common subsequence* (CLCS) problem is to find the LCS of A and B containing C as a subsequence. In 2003, Tsai [11] first proposed an algorithm with complexity $O(m^2n^2r)$. In the same year, Peng [12] also proposed an improved algorithm with $O(mnr)$ time and space complexity. Later on, many other papers [13–15] also proposed improved algorithms for the CLCS problem. Recently, Gotthilf *et al.* [16], Chen and Chao [1] proposed the related variant which excludes the given constraint as a subsequence. Chen and Chao [1] also provided solutions for another two variants which are string inclusion and string exclusion CLCS problems, although the algorithm for string exclusion CLCS is wrong as stated by Ann *et al.* [17]. In 2010, Chen [18] proposed an algorithm for the hybrid CLCS problem which is the combination of sequence inclusion CLCS and sequence exclusion CLCS. In this problem, two strings A, B and two constraint sequences P, Q are given, we are asked to find the CLCS of A and B containing P as a subsequence and excluding Q as a subsequence. Adi *et al.* [19] and Boni *et al.* [20] paid attention to the CLCS problem which the occurrence of each symbol is limited.

In this paper, we generalize the inclusion CLCS problem to the hybrid CLCS problem which involves the sequence inclusion CLCS and the string inclusion CLCS, called *sequential substring constrained longest common subsequence* (SSCLCS) problem. The problem is defined as follows.

Definition 1. *(SSCLCS) Given two strings A and B of lengths m and n, respectively, and a constraint sequence C formed by ordered strings $(C^1, C^2, C^3, \cdots, C^l)$ of total length r, where C^i is called the ith partition of the constraint and each $C^i = c_1^i c_2^i \cdots c_{l_i}^i$, the SSCLCS problem is to find the LCS D of A and B such that D contains substrings $C^1, C^2, C^3, \cdots, C^l$ and the partition order is retained.*

The sequence inclusion CLCS is a special case of SSCLCS when each partition is a single character and the string inclusion CLCS is also a special case of SSCLCS when there is only one partition. There are two different definitions that the partition order is retained. First, the partitions cannot overlap in the resulting SSCLCS. Second, the partitions may overlap in the resulting SSCLCS but the positions are monotonically increasing, that is, the starting and ending positions of the partitions in the resulting $SSCLCS$ are both increasing. For example, consider $A = atcatatgag$, $B = atcatctagg$ and $C = (acat, tag)$. $acatagg$ is an SSCLCS of the second variant, but it is not the first one. Here, we only consider the monotonically increasing case. If two neighboring partitions have the containing relation in the SSCLCS, it means one of the partition is a substring of the other. In this case, there is no use of the shorter string, and we can spend $O(r^2)$ time to preprocess the input constraints to filter the contained ones out.

The rest of this paper is organized as follows. In Section II, we give an improved algorithm for the string inclusion CLCS problem with one partition. The required time is improved from $O(mnr)$ [1] to $O(mn + (m+n)(|\Sigma|+r))$. In Section III, we propose an algorithm for the SSCLCS problem with multiple partitions which do not overlap in the resulting answer. In Section IV, we present an algorithm for the multi-partition case that the partitions may overlap in the resulting SSCLCS. Our algorithms require $O(mnl+(m+n)(|\Sigma|+r))$ and $O(mnr + (m+n)|\Sigma|)$ time for the two variants of the problem, respectively. Finally, in Section V, we will give some conclusions and future work.

II. An Improved Algorithm for the String Inclusion CLCS Problem

In this section, exactly one partition is considered, so we omit the superscript when we refer to the constraint C. That is, $C = c_1c_2c_3 \cdots c_r$. Chen and Chao [1] proposed an algorithm with $O(mnr)$ time for solving the string inclusion CLCS problem by calculating a 3D lattice directly with the dynamic programming technique applying to A, B and C. As noted above, the string inclusion CLCS problem is a special case of the SSCLCS problem with only one partition. Because many cells in their lattice are not used, we can compact the 3D lattice into a 2D lattice. Since the characters of the constraint C need to be consecutive in SSCLCS, after the first character of C is matched, the next character in SSCLCS must be the second character of C. With this fact, we can find the possible match of the constraint by continuously finding the next occurrence of the next character in the constraint in A and B. For example, consider $A = atcatatgag$, $B = atcatctagg$ and $C = tag$. We have $a_2 = b_5 = c_1$, so we can find the best SSCLCS containing C starting at (2, 5) by jumping through (4,8), (8, 9). On the other side, we can wait until the last character of C is matched, and then we find the nearest occurrence of the previous character reversely. Since we perform the dynamic programming approach, we should not refer to cells that have not yet calculated. Thus, we will perform the matching process in the backward (reverse) way.

We use $LCS(S_1, S_2)$ to denote the LCS between S_1 and S_2 and $|LCS(S_1, S_2)|$ to denote its length. $A_{i..j}$ is also used to represent the substring of a string A starting at position i and ending at position j. It is easy to obtain the following fact.

Proposition 1. *Suppose that $a_i = b_j = c_r$. If $A_{\hat{i}..i}$ and $B_{\hat{j}..j}$ contains C as their subsequences, then $LCS(A_{1..\hat{i}-1}, B_{1..\hat{j}-1}) \oplus C \oplus LCS(A_{i+1..m}, B_{j+1..n})$ forms a feasible solution of the SSCLCS problem, where $\oplus$ denotes the string concatenation operation.*

Furthermore, if there is another i', $\hat{i} \leq i'$, and $A_{i'..i}$ also contains C as its subsequence, then the solution derived from $A_{i'..i}$ is no worse than the above solution obtained in Proposition 1. Thus, we can conclude the following theorem.

Theorem 1. *Let $T = \{(i', j', i, j) | a_i = b_j = c_r$, i' and j' are the largest indices such that $A_{i'..i}$ and $B_{j'..j}$ contains C as their subsequences.$\}$ The SSCLCS solution can be obtained by finding the maximum of $LCS(A_{1..i'-1}, B_{1..j'-1}) \oplus C \oplus LCS(A_{i+1..m}, B_{j+1..n})$, where $(i', j', i, j) \in T$.*

To find the previous occurrence of a certain character, we reverse the *NextMatch* table proposed by Landau *et al.* [21] into the *PrevMatch* table which records the previous occurrence position of each symbol in every position. An example of the *PrevMatch* table for $A = atcatatgag$ is shown in Table I, where -1 means that the character never appears. The *PrevMatch* table can be constructed in $O(|S||\Sigma|)$ time and space, where S denotes the input string and Σ denotes the alphabet set of S.

We call the index $i'(j')$ in Theorem 1 as the corresponding starting position to ending position $i(j)$. For each ending position, the corresponding starting position can be calculated by using the *PrevMatch* table. We name the starting position table for A and B as ζ_A and ζ_B, respectively. For the positions where the starting position does not exist, we fill -1 in the ζ table. For example, suppose $A = atcatatgag$ and $C = acat$. Then, we have $\zeta_A = [-1, -1, -1, -1, 1, -1, 1, -1, -1, -1]$. For the same A, suppose $C = tag$, we have $\zeta_A = [-1, -1, -1, -1, -1, -1, -1, 5, -1, 7]$. The time required for constructing ζ_A and ζ_B is $O((m + n)r)$.

TABLE I
The *PrevMatch* table for $A = atcatatgag$.

	1	2	3	4	5	6	7	8	9	10
	a	t	c	a	t	a	t	g	a	g
a	-1	1	1	1	4	4	6	6	6	9
c	-1	-1	-1	3	3	3	3	3	3	3
g	-1	-1	-1	-1	-1	-1	-1	-1	8	8
t	-1	-1	2	2	2	5	5	7	7	7

We find the string inclusion CLCS with a two-layer dynamic programming lattice. Let $M[i, j, k]$ denote the length of SSCLCS between $A_{1..i}$ and $B_{1..j}$ with k constraints satisfied. When $k = 0$, it is layer 0 that represents the lattice of the ordinary LCS, in which no constraint is considered. And, when $k = 1$, it is layer 1 that represents the lattice of CLCS length, containing the given constraint string. Layer 0 can be constructed by the ordinary LCS dynamic programming formula, with an additional boundary condition that $M[i, j, 0] = -\infty$ if $i < 0$ or $j < 0$. The dynamic programming formula of layer 1 is described in Equation 1. The string inclusion CLCS can be found by tracing back from $M[m, n, 1]$ following the *PrevMatch* table and the ordinary LCS trace back link in the dynamic programming lattice.

For example, the 2 layers for $A = atcatatgag$, $B = atcatctagg$ and $C = acat$ are illustrated in Table II.

Theorem 2. *The string inclusion CLCS problem can be solved by Equation 1.*

Proof: The correctness of layer 0 follows from the ordinary dynamic programming for LCS. For layer 1, initially there is no CLCS containing the constraint, so we set the length to $-\infty$. The value on layer 1 becomes nonnegative only after we refer to layer 0 and the ζ tables does not return -1, that is, when we find an occurrence of the constraint string. And since there cannot be any other character in the region matching the constraint string, adding the length of the constraint is safe. After we find the constraint string, the rest of the part can be found with the ordinary dynamic programming formula. □

The time complexity of our algorithm is $O(mn + (m + n)(|\Sigma| + r))$, which improves a lot from Chen and Chao's method [1] with $O(mnr)$ time. Our space complexity is $O(mn + (m + n)|\Sigma|)$.

III. Algorithms for Non-overlapping Partitions

In Section II, we presented an algorithm for the case where there is a single partition in the constraint sequence. In this section, we are going to extend it to two or more partitions which do not overlap in the SSCLCS answer.

$$M[i,j,1] = \max \begin{cases} -\infty & \text{if } i \leq 0 \text{ or } j \leq 0; \\ M[i-1,j-1,1]+1 & \text{if } a_i = b_j; \\ M[\zeta_A[i]-1, \zeta_B[j]-1, 0]+r & \text{if } a_i = b_j = c_r; \\ \max \begin{cases} M[i-1,j,1] \\ M[i,j-1,1] \end{cases} & \text{otherwise.} \end{cases} \quad (1)$$

TABLE II
THE 2-LAYER DYNAMIC PROGRAMMING LATTICE FOR THE STRING INCLUSION CLCS PROBLEM WITH $A = atcatatgag$, $B = atcatctagg$ AND $C = acat$.

Layer 0												
j / i		0	1	2	3	4	5	6	7	8	9	10
		-	a	t	c	a	t	c	t	a	g	g
0	-	0	0	0	0	0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1	1	1	1	1
2	t	0	1	2	2	2	2	2	2	2	2	2
3	c	0	1	2	3	3	3	3	3	3	3	3
4	a	0	1	2	3	4	4	4	4	4	4	4
5	t	0	1	2	3	4	5	5	5	5	5	5
6	a	0	1	2	3	4	5	5	5	6	6	6
7	t	0	1	2	3	4	5	5	6	6	6	6
8	g	0	1	2	3	4	5	5	6	6	7	7
9	a	0	1	2	3	4	5	5	6	7	7	7
10	g	0	1	2	3	4	5	5	6	7	8	8
Layer 1												
		-	a	t	c	a	t	c	t	a	g	g
0	-	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	a	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	t	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	c	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
4	a	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
5	t	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	4	4	4	4
6	a	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	4	5	5	5
7	t	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	5	5	5	5
8	g	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	5	5	6	6
9	a	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	5	6	6	6
10	g	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	4	4	5	6	7	7

For ease of understanding, we will first discuss the case that exactly two partitions are involved in the constraint sequence. We extend the idea used in the previous section to solve this problem. Layer 0 stores the ordinary LCS length, in which no constraint is considered. Layers 1 and 2 correspond to the matching of the first partition and both partitions, respectively. We also construct the *PrevMatch* tables of A and B first. Since the ζ table depends on the constraint, the ζ tables for the two partitions are different. We denote them as ζ^1 and ζ^2. Layers 0 and 1 are constructed as the previous section. For layer 2, because these two partitions cannot overlap, we can apply the similar DP as layer 1 to it. Note that the value in layer 1 will become positive only after the end of the first matching to C^1. If the corresponding starting position of C^2 is in the middle of the first matching to C^1 in the layer 1, the SSCLCS length will still be $-\infty$. For example, if we add a second partition tag to the example in Table II, the values of layer 2 are all $-\infty$ except that the values of (10, 9) and (10, 10) are 7. When $M[8,9,2]$ refers to $M[4,6,1]$, there is no LCS between $atca$ and $atcatc$ containing $acat$, so the SSCLCS should still be $-\infty$.

Now, we propose the algorithm for an arbitrary number of partitions. Let ζ_A^k and ζ_B^k denote the ζ tables for C^k on A and B, respectively. The dynamic programming formula is given in Equation 2.

Theorem 3. *Equation 2 solves the SSCLCS problem with k partitions that the partitions cannot overlap.*

Proof: The correctness of each $M[i,j,k]$ is shown as follows. For $k = 0$ the dynamic programming formula is similar to the ordinary dynamic programming formula for computing LCS because there is no constraint in layer 0. The only difference is having pseudo-cells with $i, j \leq 0$ with value $-\infty$ to deal with the case when C^1 is not found in layer 1. For $k \geq 1$, it is separated into four cases. First, before the partition of this layer is contained in the SSCLCS, its length should be $-\infty$, so we set the initial value of the boundary condition to $-\infty$. Second, when $a_i \neq b_j$, the LCS length cannot be increased, so we adopt the ordinary dynamic programming formula. Third, when $a_i = b_j$, it can be added into the SSCLCS of $A_{1,\cdots,i-1}$ and $B_{1,\cdots,j-1}$. In this case, if the partition of this layer is not contained in SSCLCS($A_{1,\cdots,i-1}$, $B_{1,\cdots,j-1}$), then $M[i-1,j-1,k]$ will be $-\infty$, so the obtained $M[i,j,k]$ will still be $-\infty$. Otherwise, the constraint cannot stop us from adding it in. Fourth, when $a_i = b_j = C^k_{l_k}$, it possibly satisfies the partition of this layer. We try to find the shortest suffix of $A_{1,\cdots,i-1}$ and $B_{1,\cdots,j-1}$ containing $C^k_{1\cdots l_k-1}$ to maximize the SSCLCS length, with the help of the ζ table.

$$M[i,j,k] = \max \begin{cases} -\infty & \text{if } k=0 \text{ and } (\ i<0 \text{ or } j<0); \\ 0 & \text{if } k=0 \text{ and } i=0 \text{ and } j\geq 0; \\ 0 & \text{if } k=0 \text{ and } i\geq 0 \text{ and } j=0; \\ -\infty & \text{if } k\geq 1 \text{ and } (\ i\leq 0 \text{ or } j\leq 0); \\ M[i-1,j-1,k]+1 & \text{if } a_i=b_j; \\ M[\zeta_A^k[i]-1,\zeta_B^k[j]-1,k-1]+l_k & \text{if } k\geq 1 \text{ and } a_i=b_j=c_{l_k}^k; \\ \max\begin{cases} M[i-1,j,k] \\ M[i,j-1,k] \end{cases} & \text{otherwise.} \end{cases} \tag{2}$$

But if C^k is not a subsequence of $A_{1,\cdots,i-1}$ or $B_{1,\cdots,j-1}$ the ζ table will return -1, so we set the virtual boundary condition with $i<0$ or $j<0$ to return $-\infty$. □

The time and space complexity of the preprocessing are $O((m+n)(|\Sigma|+r))$. Both time and space complexities of Equation 2 are $O(mnl)$. So the total time and space complexities are $O(mnl+(m+n)(|\Sigma|+r))$.

IV. Algorithms for Overlapping Partitions

In this section, we discuss the SSCLCS variant that the partitions may overlap, but the starting and ending positions are both increasing. We first discuss the case of two partitions in Subsection IV-A and then we extend the algorithm to an arbitrary number of partitions in Subsection IV-B

TABLE III
THE ζ TABLE FOR $A = atcatatgag$, $C = acat$ AND tag.

acat										
w \ i	1	2	3	4	5	6	7	8	9	10
	a	t	c	a	t	a	t	g	a	g
1(t)	-1	2	-1	-1	5	-1	7	-1	-1	-1
2(a)	-1	1	-1	-1	4	-1	6	-1	-1	-1
3(c)	-1	-1	-1	-1	3	-1	3	-1	-1	7
4(a)	-1	-1	-1	-1	1	-1	1	-1	-1	-1
5(-)	-2	-2	-2	-2	0	-2	0	-2	-2	-2
tag										
	a	t	c	a	t	a	t	g	a	g
1(g)	-1	-1	-1	-1	-1	-1	-1	8	-1	10
2(a)	-1	-1	-1	-1	-1	-1	-1	6	-1	9
3(t)	-1	-1	-1	-1	-1	-1	-1	5	-1	7
4(-)	-2	-2	-2	-2	-2	-2	-2	4	-2	6

A. An Algorithm for two Partitions

We can divide the problem into two cases: the two partitions do not overlap and the two partitions overlap in the SSCLCS answer. The first case was solved in the previous section. For the second case, the situations will be different for each overlapping length. Because the number of valid overlapping lengths is no more than $min(l_1,l_2)$, it is beneficial to find all valid overlapping lengths in advance. We can apply the brute force method with time complexity of $O(l_1l_2)$ since this is not the dominating part of the time complexity.

The DP formula for layers 0 and 1 is the same as Equation 2. For layer 2, when we match to $c_{l_2}^2$, we cannot directly refer to $\zeta_A^2[i]-1$ or $\zeta_B^2[j]-1$. Instead, we have to consider every valid overlapping length and add the suffix length of C^2 after the overlap to the length of the SSCLCS ending with C^1. To achieve this, we need to extend the ζ table from one dimension to two dimensions. Each $\zeta_A^2[i,\varpi](\zeta_B^2[j,\varpi])$ records the corresponding starting position where the match to $C^2_{l_2-\varpi+1,\cdots,l_2}$ ends at $a_i(b_j)$. So the original ζ_A^2 table is equal to $\zeta_A^2[i,l_2]$. We set virtual $\zeta_A^2[i,l_2+1]=\zeta_A^2[i,l_2]-1$ and $\zeta_B^2[j,l_2+1]=\zeta_B^2[j,l_2]-1$ for the non-overlapping case. For example, the ζ tables for $A=atcatagtag$, $C=acat$ and tag are shown in Table III.

Let W_2 be the set of all valid overlapping lengths between C^2 and C^1. For a valid overlapping length $w\in W_2$, the SSCLCS length is equal to $M[\zeta_A[i,l_2-w+1],\zeta_B[j,l_2-w+1],1]+l_2-w$. Thus, the DP formula for layer 2 is given as follows.

Consider our previous example, $A=atcatatgag$, $B=atcatctagg$ and $C=(acat,tag)$, whose valid overlapping lengths are 0 and 1. The earliest matching to C^2 occurs at $M[8,9,2]$, referring to Table II, $M[8,9,2]=\max(M[5,7,1]+2,M[4,6,1]+3)=6$.

The required time and space is analyzed as follows. Let $|W_2|$ be the total number of valid overlaps between the two partitions where $|W_2|\leq\min(l_1,l_2)$. For the preprocessing, we spend $O((m+n)|\Sigma|)$ time and space to construct the *PrevMatch* tables of A and B. $O((m+n)l_1))$ time and $O(m+n)$ space are required to construct the ζ^1 tables. We take $O((m+n)l_2))$ time and space to construct the new ζ^2 tables. It takes $O(l_1l_2)$ time and $O(|W_2|)$ space to calculate the valid overlapping lengths. For the DP lattice, layers 0 and 1 are constructed in $O(mn)$ time and space. Layer 2 is constructed in $O(mn|W_2|)$ time and space because there are at most $|W_2|$ cases in each cell. So the total time and space complexity is $O(mn|W_2|+(m+n)(|\Sigma|+r)+l_1l_2)=O(mnr+(m+n)|\Sigma|)$, where $r=l_1+l_2$.

Chen and Chao [1] proposed an algorithm for the case that two constraints of lengths ρ_1 and ρ_2, respectively, are given and their order are arbitrary in the CLCS. The algorithm requires $O(mn\rho_1\rho_2)$ time and $O(mn(\rho_1+\rho_2))$ space. To solve this problem, we need can perform our algorithm in this subsection twice by setting the two partitions differently. So our algorithm is an order faster than the algorithm proposed by Chen and Chao.

B. An Algorithm for an Arbitrary Number of Partitions

In this section, we extend the algorithm for two partitions into an arbitrary number of partitions. $M[i,j,k]$ still denotes the SSCLCS length between $a_{1,\cdots,i}$ and $b_{1,\cdots,j}$ containing $C^1,\cdots,C^k$ as substrings. In the preprocessing phase, we will first construct the *PrevMatch* tables for A and B. Second, we

$$M[i,j,2] = \max \begin{cases} -\infty & \text{if } i \le 0 \text{ or } j \le 0; \\ M[i-1,j-1,2]+1 & \text{if } a_i = b_j; \\ M[\zeta_A^2[i,l_2-w+1],\zeta_B^2[j,l_2-w+1],1]+l_2-w, & \\ \quad \text{where } w \text{ is a valid overlapping length in } W_2 & \text{if } a_i = b_j = c_{l_2}^2; \\ M[i-1,j,2] & \\ M[i,j-1,2] & \end{cases} \tag{3}$$

will use the *PrevMatch* tables to construct the corresponding ζ tables for all partitions with all suffix lengths. Third, between every two consevutive partitions C^k and C^{k-1}, we will find out all valid overlapping lengths,which forms a set W_k.

The DP formula of layers 0 and 1 is the same as Equation 2. The DP formula for the remaining layers, $k \ge 2$, is given in Equation 4.

Theorem 4. *The combination of Equations 2 and 4 solve the k-partition SSCLCS problem where the partitions may overlap.*

Proof: The SSCLCS answer corresponds to characters in positions $p_{a,1}, p_{a,2}, \cdots, p_{a,l}$ from A and $p_{b,1}, p_{b,2}, \cdots, p_{b,l}$ from B. And C_1^k matches p_{a,ν_k} and p_{b,υ_k} for all $1 \le k \le l$. If two adjacent layers does not overlap, then the correctness follows from Theorem 3. If C^k overlaps with C^{k+1} in SSCLCS with length ϱ_k, it follows that $C^k_{l_k-\varrho_k+1,\cdots,l_k} = C^{k+1}_{1,\cdots,\varrho_k}$. We always find the nearest match in both A and B, so when we match for $C^{k+1}_{1,\cdots,\varrho_k}$ from $M[p_{a,\nu_{k+1}+\varrho_k-1}, p_{b,\upsilon_{k+1}+\varrho_k-1}, k+1]$ and $M[p_{a,\nu_{k+1}+\varrho_k-1}, p_{b,\upsilon_{k+1}+\varrho_k-1}, k]$, we will trace both back to $M[p_{a,\nu_{k+1}}, p_{b,\upsilon_{k+1}}, k]$. Thus, $M[p_{a,\nu_{k+1}+l_k}, p_{b,\upsilon_{k+1}+l_k}, k+1]$ $= M[p_{a,\nu_k}-1, p_{b,\upsilon_k}-1, k] + l_k + l_{k+1} - \varrho_k$ which matches with our assumption. □

The complexity of our algorithm is analyzed as follows. In the preprocessing phase, we need $O((m+n)|\Sigma|)$ time and space to construct the *PrevMatch* table of A and B. For each partition C^k, we require $O((m+n)l_k)$ time and space to construct the ζ table, so the total time for the ζ tables is $O((m+n)r)$. For every two consecutive partitions C^{k-1} and C^k, we take $O(l_{k-1}l_k)$ time to find the valid overlapping lengths and use $O(|W_k|) = O(l_k)$ space to store it, where $|W_k| \le \min(l_{k-1}, l_k)$. So the total time and space for finding all valid overlapping lengths are $O(r^2)$ and $O(r)$, respectively. For the DP lattice, $O(|W_k|) = O(l_k)$ time and $O(1)$ space are required in each cell. So the total time and space for the DP lattice are $O(mnr)$ and $O(mnl)$, respectively. Thus, the overall time and space are $O(mnr + (m+n)|\Sigma|)$ and $O(mnl + (m+n)(|\Sigma|+r))$, respectively.

V. Conclusion and Future Work

In this paper, we present a new variant of the CLCS problem, called sequential-substring CLCS, which the constraint consists of a set partitions whose positions in the SSCLCS answer are monotonically increasing. We propose algorithms for two different variants to this problem that the partitions cannot overlap or may overlap. For the former variant, we propose an algorithm with complexity $O(mnl + (m+n)(|\Sigma|+r))$. And for the second variant, we proposed an $O(mnr+(m+n)|\Sigma|)$ time and $O(mnl+(m+n)(|\Sigma|+r))$ space algorithm. We also proposed algorithms for the string inclusion CLCS problem with one or two constraint strings and our algorithms achieve an order improvement to the previous known algorithms by Chen and Chao [1]

There are two possible future work to our SSCLCS problem. The first one is to restrict the range between the partitions which might be able to be used in the motif finding. The second one is to exclude the constraint, that is, the CLCS does not contain the sequential subsequences.

References

[1] 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, 2009.

[2] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association for Computing Machinery*, vol. 21, no. 1, pp. 168–173, 1974.

[3] C. B. Yang and R. C. T. Lee, "Systolic algorithm for the longest common subsequence problem," *Journal of the Chinese Institue of Engineers*, vol. 10, no. 6, pp. 691–699, 1987.

[4] A. Apostolico and C. Guerra, "The longest common subsequences problem revisited," *Algorithmica*, vol. 18, pp. 1–11, 1987.

[5] C. Rick, "Simple and fast linear space computation of longest common subsequences," *Information Processing Letters*, vol. 75, pp. 275–281, 2000.

[6] K. S. Huang, C. B. Yang, K. T. Tseng, Y. H. Peng, and H. Y. Ann, "Dynamic programming algorithms for the mosaic longest common subsequence problem," *Information Processing Letters*, vol. 102, pp. 99–103, 2007.

[7] K. S. Huang, C. B. Yang, K. T. Tseng, H. Y. Ann, and Y. H. Peng, "Efficient algorithms for finding interleaving relationship between sequences," *Information Processing Letters*, vol. 105, no. 5, pp. 188–193, 2008.

[8] Y. H. Peng, C. B. Yang, K. S. Huang, C. T. Tseng, and C.-Y. Hor, "Efficient sparse dynamic programming for the merged lcs problem with block constraints," *International Journal of Innovative Computing, Information and Control*, vol. 6, no. 4, pp. 1935–1947.

[9] F. Nicolas and E. Rivals, "Longest common subsequence problem for unoriented and cyclic strings," *Theoretical Computer Science*, vol. 370, pp. 1–18, Feb. 2007.

[10] 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, Mar. 2010.

$$M[i,j,k] = \max \begin{cases} -\infty & \text{if } i \leq 0 \text{ or } j \leq 0; \\ M[i-1,j-1,k]+1 & \text{if } a_i = b_j; \\ M[\zeta_A[i,l_k-w+1],\zeta_B[j,l_k-w+1],k-1]+l_2-w, & \\ \quad \text{where } w \text{ is a valid overlapping length in } W_k & \text{if } a_i = b_j = c^k_{l_k}; \\ M[i-1,j,k] & \\ M[i,j-1,k] & \end{cases} \tag{4}$$

[11] Y. T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, vol. 88, pp. 173–176, 2003.

[12] C.-L. Peng, "An approach for solving the constrained longest common subsequence problem," *Master Thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan*, July 2003.

[13] A. N. Arslan and O. Egecioglu, "Algorithms for the constrained longest common subsequence problems," *International Journal of Foundations Computer Science*, vol. 16, no. 5, pp. 1099–1109, 2005.

[14] 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.

[15] 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.

[16] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein, "Restricted LCS," in *Proceedings of the 17th international conference on String processing and information retrieval*, ser. SPIRE'10, 2010, pp. 250–257.

[17] H. Y. Ann, C. B. Yang, and C. T. Tseng, "Efficient polynomial-time algorithms for variants of the multiple constrained LCS problem," in *Proceedings of the 2011 International Conference on Computing and Security (ICCS'11), Ulaanbaatar, Mongolia*, July 2011.

[18] Y. C. Chen, "Algorithms for the hybrid constrained longest common subsequence problem," in *Proc. of the 27th Workshop on Combinatorial Mathematics and Computation Theory*, Taichung, Taiwan, 2010, pp. 32–37.

[19] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi, "Repetition-free longest common subsequence," *Electronic Notes in Discrete Mathematics*, vol. 30, pp. 243–248, Feb. 2008.

[20] P. Bonizzoni, G. Della Vedova, R. Dondi, and Y. Pirola, "Variants of constrained longest common subsequence," *Information Processing Letters*, vol. 110, pp. 877–881, Sept. 2010.

[21] G. M. Landau, E. Myers, and M. Ziv-Ukelson, "Two algorithms for LCS consecutive suffix alignment," *Journal of Computer and System Sciences*, vol. 73, no. 7, pp. 1095–1117, 2007.