Journal of Complexity 29 (2013) 44–52

Contents lists available at SciVerse ScienceDirect

Journal of Complexity

journal homepage: www.elsevier.com/locate/jco

Efficient algorithms for the longest common subsequence problem with sequential substring constraints[☆]

Chiou-Ting Tseng[a], Chang-Biau Yang[a,*], Hsing-Yen Ann[b]

[a] *Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung 80424, Taiwan*

[b] *National Center for High-Performance Computing, Tainan 74147, Taiwan*

ARTICLE INFO

Article history:
Received 10 June 2011
Accepted 18 July 2012
Available online 11 August 2012

Keywords:
Algorithm design
Bioinformatics
Longest common subsequence
Constrained LCS
Sequential substring

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 $(C^1, C^2, C^3, \ldots, C^l)$ with total length r. The problem is that of finding the longest common subsequence D of A and B containing $C^1, C^2, C^3, \ldots, C^l$ as substrings and with the order of the C's retained. This problem has two variants, depending on whether the strings in C cannot overlap or may overlap. We propose algorithms with $O(mnl + (m+n)(|\Sigma| + r))$ and $O(mnr + (m+n)|\Sigma|)$ time for the two variants. For the special case with one or two constraints, our algorithm runs in $O(mn + (m+n)(|\Sigma| + r))$ or $O(mnr + (m+n)|\Sigma|)$ time, respectively—an order faster than the algorithm proposed by Chen and Chao.

1. 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 that of finding 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 [23]. Much ink has been expended

☆ This research work was partially supported by the National Science Council of Taiwan under contract NSC100-2221-E-110-050.

* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

on this topic in the past few decades [24,3,21], and lots of variants of the LCS problem have also been proposed, such as the *mosaic LCS* problem [15], the *merged LCS* problem [14,20], the *cyclic string correction problem* [18] and the *block edit* problem [2].

Given two strings A and B and a constraint sequence C with lengths m, n and r respectively, the *constrained longest common subsequence* (CLCS) problem is that of finding the LCS of A and B containing C as a subsequence. In 2003, Tsai [22] first proposed an algorithm with complexity $O(m^2n^2r)$. In the same year, Peng [19] also proposed an improved algorithm with $O(mnr)$ time and space complexity. Later on, many other papers [4,10,16] also proposed improved algorithms for the CLCS problem. Recently, Gotthilf et al. [13], Chen and Chao [9] proposed the related variant which excludes the given constraint as a subsequence. Chen and Chao [9] also provided solutions for two other variants which are string inclusion and string exclusion CLCS problems, although the algorithm for string exclusion CLCS is wrong. In 2010, Chen [8] proposed an algorithm for the hybrid CLCS problem which is the combination of sequence inclusion CLCS and sequence exclusion CLCS. Given two strings A and B, and two constraint sequences P and Q, this problem is that of finding the CLCS of A and B containing P as a subsequence and excluding Q as a subsequence.

On the other hand, some researchers put a constraint on the number of symbol occurrences in the alphabet in the LCS. In the *exemplar LCS* (ELCS) problem [6], Σ is divided into the mandatory set Σ_m and the optional set Σ_o. The ELCS problem is that of restricting the number of symbol occurrences in Σ_m and Σ_o in the final LCS obtained. The *repetition free LCS* (RFLCS) problem [1,12,11,5] and *doubly constrained LCS* (DC-LCS) problem [7] are those of finding the LCS such that each symbol appears at most once and twice, respectively. They are both special cases of the ELCS problem, when $\Sigma_o = \Sigma$.

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 the *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 $(C^1, C^2, C^3, \dots, C^l)$ of total length r, where C^i is called the ith partition of the constraint and each $C^i = c^i_1 c^i_2 \cdots c^i_{l_i}$, the SSCLCS problem is that of finding the LCS D of A and B such that D contains substrings $C^1, C^2, C^3, \dots, C^l$ and the partition order is retained.

The sequence inclusion CLCS is a special case of SSCLCS where each partition is a single character and the string inclusion CLCS is also a special case of SSCLCS where there is only one partition. There are two different definitions of the partition order being 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 for 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 2, we give an improved algorithm for the string inclusion CLCS problem with one partition. The required time is improved from $O(mnr)$ [9] to $O(mn+(m+n)(|\Sigma|+r))$. In Section 3, we propose an algorithm for the SSCLCS problem with multiple partitions which do not overlap in the resulting answer. In Section 4, we present an algorithm for the multi-partition case where 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 5, we will give some conclusions.

2. 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 [9] proposed an algorithm with $O(mnr)$ time for solving the string inclusion CLCS problem by calculating a 3D lattice directly with the dynamic

Table 1
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

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 of 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 hand, 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 been calculated. Thus, we will perform the matching process in the backward (reverse) way.

We use $\text{LCS}(S_1, S_2)$ to denote the LCS between S_1 and S_2 and $|\text{LCS}(S_1, S_2)|$ to denote its length. $A_{i..j}$ is also used to denote 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 $B_{\hat{j}..j}$ contain C as their subsequences, then $\text{LCS}(A_{1..\hat{i}-1}, B_{1..\hat{j}-1}) \oplus C \oplus \text{LCS}(A_{i+1..m}, B_{j+1..n})$ forms a feasible solution of the SSCLCS problem, where $\oplus$ denotes the string concatenation operation.*

Furthermore, if there is another i', $\hat{i} \le i'$, and $A_{i'..i}$ also contains C as its subsequence, then the solution derived from $A_{i'..i}$ is not 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 $B_{j'..j}$ contain C as their subsequences$\}$. The SSCLCS solution can be obtained by finding the maximum of $\text{LCS}(A_{1..i'-1}, B_{1..j'-1}) \oplus C \oplus \text{LCS}(A_{i+1..m}, B_{j+1..n})$, 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. [17] into the *PrevMatch* table which records the previous occurrence position of each symbol in every position. An example of a *PrevMatch* table for $A = atcatatgag$ is shown in Table 1, 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 the starting position corresponding to ending position i (j). For each ending position, the corresponding starting position can be calculated with at most r lookups in the *PrevMatch* table. We name the starting position tables for A and B as *StartPos* tables ζ_A and ζ_B, respectively. For the position whose corresponding starting position does not exist, we fill −1 in the *StartPos* 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 the two *StartPos* tables is $O((m + n)r)$, since each position requires at most r lookups in the *PrevMatch* table and each lookup takes only constant time.

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 partitions (strings) satisfied. When $k = 0$, it is layer 0 that represents the lattice of the ordinary LCS, in which no constraint is considered. And,

Table 2
The two-layer dynamic programming lattice for the string inclusion CLCS problem with $A = atcatatgag$, $B = atcatctagg$ and $C = acat$.

i	j											
		0	1	2	3	4	5	6	7	8	9	10
		–	a	t	c	a	t	c	t	a	g	g
Layer 0												
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												
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

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 using the ordinary LCS dynamic programming formula, with the additional boundary condition that $M[i, j, 0] = -\infty$ if $i < 0$ or $j < 0$. The dynamic programming formula for layer 1 is described in Eq. (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.

$$M[i, j, 1] = \max \begin{cases} -\infty & \text{if } i \le 0 \text{ or } j \le 0; \\ M[i-1, j-1, 1] + 1 & \text{if } a_i = b_j; \\ M[\zeta_A[i] - 1, \zeta_B[j] - 1, 0] + r & \text{if } a_i = b_j = c_r; \\ M[i-1, j, 1] & \multirow{2}{*}{\text{otherwise.}} \\ M[i, j-1, 1] & \end{cases} \tag{1}$$

As examples, the two layers for $A = atcatatgag$, $B = atcatctagg$ and $C = acat$ are illustrated in Table 2.

Theorem 2. *The string inclusion CLCS problem can be solved by using Eq. (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 of one cell on layer 1 becomes positive only after it refers to layer 0 and the *StartPos* tables do not return -1, that is, when we find an occurrence of the constraint string. And since there cannot be any other character inside the region matching the constraint string, adding the length of the constraint is evidently correct. After we find the constraint string, the rest of the part can be found with the ordinary dynamic programming formula. $\square$

In summary, we first construct the *PrevMatch* tables for A and B in $O((m+n)|\Sigma|)$ time and space. Second, we use the *PrevMatch* tables to construct the *StartPos* tables ζ_A and ζ_B in $O((m+n)r)$ time and space. With ζ_A and ζ_B, each cell in the M table can be obtained in constant time. So the time complexity of our algorithm is $O(mn+(m+n)(|\Sigma|+r))$, which improves a lot on that of Chen and Chao's method [9] with $O(mnr)$ time. Our space complexity is $O(mn+(m+n)|\Sigma|)$.

3. Algorithms for non-overlapping partitions

In Section 2, 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.

For ease of understanding, we will first discuss the case where 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 *StartPos* table depends on the constraint, the *StartPos* 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 a DP similar to that for 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 layer 1, the SSCLCS length will still be $-\infty$. For example, if we add a second partition *tag* to the example in Table 2, 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 *StartPos* tables for C^k on A and B, respectively. The dynamic programming formula is given in Eq. (2):

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

Theorem 3. *Eq. (2) solves the SSCLCS problem with l 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 almost the same as the ordinary dynamic programming formula for computing the traditional LCS because there is no constraint in layer 0. The only difference is that some extra pseudo-cells have value $-\infty$ when $i<0$ or $j<0$. For $k\ge 1$, 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 $a_i\ne b_j$, 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 $A_{1..i-1}$ and $B_{1..j-1}$. In this case, if the partition of this layer is not contained in SSCLCS($A_{1..i-1}, B_{1..j-1}$), then $M[i-1,j-1,k]$ is $-\infty$, so the $M[i,j,k]$ obtained will still be $-\infty$. Otherwise, the constraint cannot stop us from adding it. Fourth, when $a_i=b_j=C_{l_k}^k$, it possibly satisfies the partition of this layer. In this case, we can jump directly to the corresponding starting position, with the help of the *StartPos* table, and add length l_k to the answer. But if C^k is not a subsequence of $A_{1..i}$ or $B_{1..j}$, the *StartPos* table will return -1, so we set the virtual boundary condition with $i<0$ or $j<0$ to return $-\infty$. □

Table 3
The *StartPos* tables for $A = atcatatgag$, $C = (acat, tag)$.

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

The time and space complexities of the preprocessing for constructing the *StartPos* tables are both $O((m+n)(|\Sigma|+r))$. The time and space complexities of Eq. (2) are both $O(mnl)$, since the required time for each cell is constant. So the total time and space complexities are both $O(mnl + (m+n)(|\Sigma| + r))$.

4. Algorithms for overlapping partitions

In this section, we discuss the SSCLCS variant where the partitions may overlap, but the starting and ending positions are both monotonically increasing. We first discuss the case of two partitions in Section 4.1 and then we extend the algorithm to an arbitrary number of partitions in Section 4.2

4.1. An algorithm for two partitions

When two partitions are allowed to be overlapped, the situations become different for each overlapping length (including zero length). Because the number of valid overlapping lengths is not more than $\min(l_1, l_2)$, it is beneficial to find, in advance, the lengths of all valid overlapping substrings for which a certain suffix of C^1 overlaps a certain prefix of C^2. We can apply the brute force method with time complexity $O(l_1l_2)$ since this is not the dominating part of the time complexity.

The DP formula for layer 0 is the same as Eq. (2). For layers 1 and 2, when we match a_i and b_j to $c^2_{l_2}$, we cannot directly refer to $\zeta^2_A[i] - 1$ or $\zeta^2_B[j] - 1$. Instead, we have to consider every valid overlapping length and add the remaining suffix length of C^2, obtained by removing the overlapping length, to the length of the SSCLCS ending with C^1. To achieve this, we have to extend the *StartPos* table from one dimension to two dimensions. If there is an overlapping length w, $1 \le w \le \min(l_1, l_2) - 1$, we may only consider the match to the suffix $C^2_{w+1..l_2}$. However, there cannot be any character matching between C^2_w and C^2_{w+1}, so we have to consider the match to $C^2_{w..l_2}$. We use $\zeta^2_A[i, w](\zeta^2_B[j, w])$ to record the corresponding starting position where the match to $C^2_{w..l_2}$ ends at $a_i(b_j)$. So the original ζ^2_A table is equal to $\zeta^2_A[i, 1]$. When there is no overlap, we set virtual $\zeta^2_A[i, 0] = \zeta^2_A[i, 1] - 1$ and $\zeta^2_B[j, 0] = \zeta^2_B[j, 1] - 1$ for this case. As examples, the two-dimensional *StartPos* tables for $A = atcatagtag$ and $C = (acat, tag)$ are shown in Table 3.

Let $M_1[i, j, 1]$ be the lattice same as layer 1 of Eq. (2). $M_2[i, j, 1]$ is the lattice for recording the SSCLCS length with matching C^1 exactly at the end and it is obtained by removing "$M[i-1, j-1, k] + 1$ if $a_i = b_j$" from Eq. (2). Let W_2 be the set of all valid lengths of overlap between C^1 and C^2. For a valid overlapping length $w \in W_2$, the SSCLCS length is equal to $M_2[\zeta^2_A[i, w], \zeta^2_B[j, w], 1] + l_2 - w$. We jump to $\zeta^2_A[i, w]$ but not $\zeta^2_A[i, w+1] - 1$ because C^2_w and C^2_{w+1} must be consecutive in the SSCLCS. The situation is similar for $\zeta_B^2[j, w]$. Thus, the DP formula for layer 2 is given as follows:

$$M_1[i,j,2] = \max \begin{cases} -\infty & \text{if } i \le 0 \text{ or } j \le 0; \\ M_1[i-1,j-1,2]+1 & \text{if } a_i = b_j; \\ M_2[\zeta_A^2[i,w], \zeta_B^2[j,w], 1] + l_2 - w, & \\ \quad \text{where } w \text{ is a valid overlapping length in } W_2 & \text{if } a_i = b_j = c_{l_2}^2; \\ M_1[\zeta_A^2[i,0], \zeta_B^2[j,0], 1] + l_2, & \text{if } a_i = b_j = c_{l_2}^2; \\ M_1[i-1,j,2] & \text{otherwise.} \\ M_1[i,j-1,2] & \end{cases} \tag{3}$$

Consider our previous example, $A = atcatatgag$, $B = atcatctagg$ and $C = (acat, tag)$, whose valid overlapping length is 1. The earliest matching to C^2 occurs at $M_1[8,9,2]$; $M_1[8,9,2] = \max(M_1[4,6,1]+3, M_2[5,7,1]+2) = 6$.

The required time and space are analyzed as follows. Let $|W_2|$ be the total number of valid overlaps between the two partitions where $|W_2| \le \min(l_1, l_2)$. For the preprocessing, we spend $O((m+n)|\Sigma|)$ time and space in constructing 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 $O(l_1 l_2)$ 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 $O(mn|W_2| + (m+n)(|\Sigma| + r) + l_1 l_2) = O(mnr + (m+n)|\Sigma|)$, where $r = l_1 + l_2$.

Chen and Chao [9] proposed an algorithm for the case where two constraints of lengths ρ_1 and ρ_2 are given and the order is arbitrary in the CLCS. Their algorithm requires $O(mn\rho_1\rho_2)$ time and $O(mn(\rho_1+\rho_2))$ space. To solve this problem, we 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.

4.2. An algorithm for an arbitrary number of partitions

In this section, we extend the algorithm for two partitions to one for an arbitrary number of partitions. $M[i,j,k]$ still denotes the SSCLCS length between $A_{1,\dots,i}$ and $B_{1,\dots,j}$ containing $C^1, \dots, C^k$ as substrings. In the preprocessing phase, we first construct the *PrevMatch* tables for A and B. Second, we use the *PrevMatch* tables to construct the *StartPos* tables for all partitions with all overlapping lengths. Third, for every two consecutive partitions C^{k-1} and C^k, we find all valid lengths of overlap, to form the set W_k.

Let $M_1[i,j,k]$ record the SSCLCS length between $A_{1..i}$ and $B_{1..j}$ with k partitions satisfied and $M_2[i,j,k]$ store the SSCLCS length with matching C^k exactly at the end. The DP formula is given as follows:

$$M_1[i,j,k] = \max \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0; \\ 0 & \text{if } k = 0 \text{ and } i = 0 \text{ and } j \ge 0; \\ 0 & \text{if } k = 0 \text{ and } i \ge 0 \text{ and } j = 0; \\ -\infty & \text{if } k \ge 1 \text{ and } (i \le 0 \text{ or } j \le 0); \\ M_1[i-1,j-1,k]+1 & \text{if } a_i = b_j; \\ M_2[i,j,k] & \text{if } k \ge 1 \text{ and } a_i = b_j = c_{l_k}^k; \\ M_1[i-1,j,k] & \text{otherwise.} \\ M_1[i,j-1,k] & \end{cases} \tag{4}$$

$$M_2[i,j,k] = \max \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0; \\ 0 & \text{if } k = 0 \text{ and } i = 0 \text{ and } j \geq 0; \\ 0 & \text{if } k = 0 \text{ and } i \geq 0 \text{ and } j = 0; \\ -\infty & \text{if } k \geq 1 \text{ and } (i \leq 0 \text{ or } j \leq 0); \\ M_2[\zeta_A^k[i,w], \zeta_B^k[j,w], k-1] + l_k - w, & \\ \quad \text{where } w \text{ is a valid overlapping} & \\ \quad \text{length in } W_k & \text{if } a_i = b_j = c_{l_k}^k; \\ M_1[\zeta_A^k[i,0], \zeta_B^k[j,0], k-1] + l_k, & \text{if } a_i = b_j = c_{l_k}^k; \\ M_2[i-1,j,k] & \\ M_2[i,j-1,k] & \text{otherwise.} \end{cases} \tag{5}$$

Theorem 4. *The combination of Eqs. (4) and (5) solves the l-partition SSCLCS problem where the partitions may overlap.*

Proof. The SSCLCS answer D corresponds to characters in positions $p_{a,1}, p_{a,2}, \ldots, p_{a,|D|}$ from A and $p_{b,1}, p_{b,2}, \ldots, p_{b,|D|}$ from B. Suppose c_1^k matches p_{a,ν_k} and p_{b,υ_k} for all $1 \leq k \leq l$. If two adjacent layers do not overlap, then the correctness follows from Theorem 3. If C^{k-1} overlaps with C^k in SSCLCS with length w_k, it follows that $C^{k-1}_{l_{k-1}-w_k+1..l_{k-1}} = C^k_{1..w_k}$. We always find the nearest match in both A and B, so when we match for $C^k_{1..w_k}$ from $M[p_{a,\nu_k+w_k-1}, p_{b,\upsilon_k+w_k-1}, k]$ and $M[p_{a,\nu_k+w_k-1}, p_{b,\upsilon_k+w_k-1}, k-1]$, we will trace both back to $M[p_{a,\nu_k}, p_{b,\upsilon_k}, k-1]$. Thus, $M[p_{a,\nu_k+l_k-1}, p_{b,\upsilon_k+l_k-1}, k] = M[p_{a,\nu_{k-1}}, p_{b,\upsilon_{k-1}}, k-1] + l_{k-1} + l_k - w_k - 1$ which matches with our assumption. □

The complexity of the above 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 *StarPos* table, so the total time required for all *StarPos* 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 them, where $|W_k| \leq \min(l_{k-1}, l_k)$. So the total time and space required 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.

5. Conclusion

In this paper, we present a new variant of the CLCS problem, called the sequential substring CLCS, in which the constraint consists of a set partitions whose positions in the SSCLCS answer are monotonically increasing. We propose algorithms for two different variants depending on whether the partitions cannot overlap or may overlap. For the former variant, the time and space complexities are both $O(mnl + (m+n)(|\Sigma| + r))$. And for the second variant, our algorithm requires $O(mnr + (m+n)|\Sigma|)$ time and $O(mnl + (m+n)(|\Sigma| + r))$ space. For the special cases where only one or two constraint strings are given, our algorithms achieve an order improvement in time complexity with respect to the previous known algorithms proposed by Chen and Chao [9].

There are two possible future avenues to explore for our SSCLCS problem. The first one is that of restricting the range for the distance between the partitions, which might be useful in motif finding. The second one is that of excluding the constraint, that is, considering cases where the CLCS does not contain sequential substrings.

References

[1] S.S. Adi, M.D. Braga, C.G. Fernandes, C.E. Ferreira, F.V. Martinez, M.-F. Sagot, M.A. Stefanes, C. Tjandraatmadja, Y. Wakabayashi, Repetition-free longest common subsequence, Discrete Applied Mathematics 158 (12) (2010) 1315–1324.

[2] H.Y. Ann, C.B. Yang, Y.H. Peng, B.C. Liaw, Efficient algorithms for the block edit problems, Information and Computation 208 (3) (2010) 221–229.
[3] A. Apostolico, C. Guerra, The longest common subsequences problem revisited, Algorithmica 18 (1987) 1–11.
[4] A.N. Arslan, O. Egecioglu, Algorithms for the constrained longest common subsequence problems, International Journal of Foundations of Computer Science 16 (5) (2005) 1099–1109.
[5] G. Blin, P. Bonizzoni, R. Dondi, F. Sikora, On the parameterized complexity of the repetition free longest common subsequence problem, Information Processing Letters 112 (7) (2012) 272–276.
[6] P. Bonizzoni, G.D. Vedova, R. Dondi, G. Fertin, R. Rizzi, S. Vialette, Exemplar longest common subsequence, IEEE/ACM Transactions on Computational Biology and Bioinformatics 4 (4) (2007) 535–543.
[7] P. Bonizzoni, G. Della Vedova, R. Dondi, Y. Pirola, Variants of constrained longest common subsequence, Information Processing Letters 110 (2010) 877–881.
[8] 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.
[9] Y.C. Chen, K.M. Chao, On the generalized constrained longest common subsequence problems, Journal of Combinatorial Optimization 21 (3) (2011) 383–392.
[10] F.Y.L. Chin, A.D. Santis, A.L. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Information Processing Letters 90 (4) (2004) 175–179.
[11] C.G. Fernandes, C.E. Ferreira, C. Tjandraatmadja, Yoshiko Wakabayashi, A polyhedral investigation of the LCS problem and a repetition-free variant, in: Proceedings of the 8th Latin American Conference on Theoretical Informatics, LATIN'08, 2008. pp. 329–338.
[12] C.E. Ferreira, C. Tjandraatmadja, A branch-and-cut approach to the repetition-free longest common subsequence problem, Electronic Notes in Discrete Mathematics 36 (2010) 527–534.
[13] Z. Gotthilf, D. Hermelin, G.M. Landau, M. Lewenstein, Restricted LCS, in: Proceedings of the 17th International Conference on String Processing and Information Retrieval, SPIRE'10, 2010. pp. 250–257.
[14] K.S. Huang, C.B. Yang, K.T. Tseng, H.Y. Ann, Y.H. Peng, Efficient algorithms for finding interleaving relationship between sequences, Information Processing Letters 105 (5) (2008) 188–193.
[15] K.S. Huang, C.B. Yang, K.T. Tseng, Y.H. Peng, H.Y. Ann, Dynamic programming algorithms for the mosaic longest common subsequence problem, Information Processing Letters 102 (2007) 99–103.
[16] C.S. Iliopoulos, M.S. Rahman, New efficient algorithms for the LCS and constrained LCS problems, Information Processing Letters 106 (1) (2008) 13–18.
[17] G.M. Landau, E. Myers, M. Ziv-Ukelson, Two algorithms for LCS consecutive suffix alignment, Journal of Computer and System Sciences 73 (7) (2007) 1095–1117.
[18] F. Nicolas, E. Rivals, Longest common subsequence problem for unoriented and cyclic strings, Theoretical Computer Science 370 (2007) 1–18.
[19] 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, 2003.
[20] Y.H. Peng, C.B. Yang, K.S. Huang, C.T. Tseng, C.-Y. Hor, Efficient sparse dynamic programming for the merged LCS problem with block constraints, International Journal of Innovative Computing, Information and Control 6 (4) (2010) 1935–1947.
[21] C. Rick, Simple and fast linear space computation of longest common subsequences, Information Processing Letters 75 (2000) 275–281.
[22] Y.T. Tsai, The constrained longest common subsequence problem, Information Processing Letters 88 (2003) 173–176.
[23] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, Journal of the Association for Computing Machinery 21 (1) (1974) 168–173.
[24] C.B. Yang, R.C.T. Lee, Systolic algorithm for the longest common subsequence problem, Journal of the Chinese Institute of Engineers 10 (6) (1987) 691–699.