Space-Efficient STR-IC-LCS Computation

Yuuki Yonemoto[1] Yuto Nakashima[2]
Shunsuke Inenaga[2,3] Hideo Bannai[4]
[1] Department of Information Science and Technology,
Kyushu University, Fukuoka, Japan
yonemoto.yuuki.240@s.kyushu-u.ac.jp
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
nakashima.yuto.003@m.kyushu-u.ac.jp
inenaga@inf.kyushu-u.ac.jp
[3] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan
[4] M&D Data Science Center,
Tokyo Medical and Dental University, Tokyo, Japan

Abstract

One of the most fundamental method for comparing two given strings A and B is the *longest common subsequence* (LCS), where the task is to find (the length) of the longest common subsequence. In this paper, we address the *STR-IC-LCS* problem which is one of the constrained LCS problems proposed by Chen and Chao [J. Comb. Optim, 2011]. A string Z is said to be an STR-IC-LCS of three given strings A, B, and P, if Z is one of the longest common subsequences of A and B that contains P as a substring. We present a space efficient solution for the STR-IC-LCS problem. Our algorithm computes the length of an STR-IC-LCS in $O(n^2)$ time and $O((\ell+1)(n-\ell+1))$ space where ℓ is the length of a longest common subsequence of A and B of length n. When $\ell = O(1)$ or $n-\ell = O(1)$, then our algorithm uses only linear $O(n)$ space.

1 Introduction

Comparison of two given strings (sequences) has been a central task in Theoretical Computer Science, since it has many applications including alignments of biological sequences, spelling corrections, and similarity searches.

One of the most fundamental method for comparing two given strings A and B is the *longest common subsequence LCS*, where the task is to find (the length of) a common subsequence L that can be obtained by removing zero or more characters from both A and B, and no such common subsequence longer than L exists. A classical dynamic programming (DP) algorithm is able to compute an LCS of A and B in quadratic $O(n^2)$ time with $O(n^2)$ working space, where n is the length of the input strings [12]. In the word RAM model with ω machine word size, the so-called "Four-Russian" method allows one to compute the length of an LCS of two given strings in $O(n^2/k + n)$ time, for any $k \leq \omega$, in the case of constant-size alphabets [9]. Under a common assumption that $\omega = \log_2 n$, this method leads to weakly sub-quadratic $O(n^2/\log^2 n)$ time solution for constant alphabets. In the case of general alphabets, the state-of-the-art algorithm computes the length of an LCS in $O(n^2 \log^2 k/k^2 + n)$ time [2], which is weakly sub-quadratic $O(n^2(\log\log n)^2/\log^2 n)$ time for $k \leq \omega = \log_2 n$. It is widely believed that such "log-shaving" improvements would be the best possible one can hope, since an $O(n^{2-\epsilon})$-time LCS computation for any constant $\epsilon > 0$ refutes the famous strong exponential time hypothesis (SETH) [1].

Recall however that this conditional lower-bound under the SETH does not enforce us to use (strongly) quadratic *space* in LCS computation. Indeed, a simple modification to the DP method

permits us to compute the length of an LCS in $O(n^2)$ time with $O(n)$ working space. There also exists an algorithm that computes an LCS string in $O(n^2)$ time with only $O(n)$ working space [6]. The aforementioned log-shaving methods [9, 2] use only $O(2^k + n)$ space, which is $O(n)$ for $k \leq \omega = \log_2 n$.

In this paper, we follow a line of research called the *Constrained* LCS problems, in which a pattern P that represents a-priori knowledge of a user is given as a third input, and the task is to compute the longest common subsequence of A and B that meets the condition w.r.t. P [11, 4, 3, 5, 8, 7]. The variant we consider here is the *STR-IC-LCS* problem of computing a longest string Z which satisfies that (1) Z *includes* P as a *substring* and (2) Z is a common subsequence of A and B. We present a space-efficient algorithm for the STR-IC-LCS problem in $O(n^2)$ time with $O((\ell+1)(n-\ell+1))$ working space, where $\ell = \mathsf{lcs}(A, B)$ denotes the length of an LCS of A and B. Our solution improves on the state-of-the-art STR-IC-LCS algorithm of Deorowicz [5] that uses $\Theta(n^2)$ time and $\Theta(n^2)$ working space, since $O((\ell+1)(n-\ell+1)) \subseteq O(n^2)$ always holds. Our method requires only sub-quadratic $o(n^2)$ space whenever $\ell = o(n)$. In particular, when $\ell = O(1)$ or $n - \ell = O(1)$, which can happen when we compare very different strings or very similar strings, respectively, then our algorithm uses only linear $O(n)$ space.

Our method is built on a non-trivial extension of the LCS computation algorithm by Nakatsu et al. [10] that runs in $O(n(n-\ell+1))$ time with $O((\ell+1)(n-\ell+1))$ working space. We remark that the $O(n^{2-\epsilon})$-time conditional lower-bound for LCS also applies to our case since STR-IC-LCS with the pattern P being the empty string is equal to LCS, and thus, our solution is almost time optimal (except for log-shaving, which is left for future work).

Related Work.

There exists four variants of the *Constrained* LCS problems, STR-IC-LCS/SEQ-IC-LCS/STR-EC-LCS/SEQ-EC-LCS, each of which is to compute a longest string Z such that (1) Z includes/excludes the constraint pattern P as a substring/subsequence and (2) Z is a common subsequence of the two target strings A and B [11, 4, 3, 5, 8, 7]. Yamada et al. [13] proposed an $O(n\sigma + (\ell'+1)(n-\ell'+1)r)$-time and space algorithm for the STR-EC-LCS problem, which is also based on the method by Nakatsu et al. [10], where σ is the alphabet size, ℓ' is the length of an STR-EC-LCS and r is the length of P. However, the design of our solution to STR-IC-LCS is quite different from that of Yamada et al.'s solution to STR-EC-LCS.

2 Preliminaries

2.1 Strings

Let Σ be an *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0. For a string $S = uvw$, u, v and w are called a *prefix*, *substring*, and *suffix* of S, respectively.

The i-th character of a string S is denoted by $S[i]$, where $1 \leq i \leq |S|$. For a string S and two integers $1 \leq i \leq j \leq |S|$, let $S[i..j]$ denote the substring of S that begins at position i and ends at position j, namely, $S[i..j] = S[i] \cdots S[j]$. For convenience, let $S[i..j] = \varepsilon$ when $i > j$. S^R denotes the reversed string of S, i.e., $S^R = S[|S|] \cdots S[1]$. A non-empty string Z is called a *subsequence* of another string S if there exist increasing positions $1 \leq i_1 < \cdots < i_{|Z|} \leq |S|$ in S such that $Z = S[i_1] \cdots S[i_{|Z|}]$. The empty string ε is a subsequence of any string. A string that is a subsequence of two strings A and B is called a *common subsequence* of A and B.

2.2 STR-IC-LCS

Let A, B, and P be strings. A string Z is said to be *an STR-IC-LCS* of two target strings A and B *including* the pattern P if Z is a longest string such that (1) P is a substring of Z and (2) Z is a common subsequence of A and B.

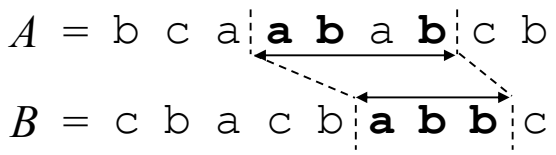

Figure 1: Let $A = \mathtt{bcdababcb}$, $B = \mathtt{cbacbaaba}$, and $P = \mathtt{abb}$. The length of an STR-IC-LCS of these strings is 6. One of such strings can be obtained by minimal intervals [4..7] over A and [6..8] over B because $\mathsf{lcs}(\mathtt{bca}, \mathtt{cbacb}) = 2$, $|P| = 3$, and $\mathsf{lcs}(\mathtt{cb}, \mathtt{c}) = 1$.

For ease of exposition, we assume that $n = |A| = |B|$, but our algorithm to follow can deal with the general case where $|A| \neq |B|$. We can also assume that $|P| \leq n$, since otherwise there clearly is no solution. In this paper, we present a space-efficient algorithm that computes an STR-IC-LCS in $O(n^2)$ time and $O((\ell+1)(n-\ell+1))$ space, where $\ell = \mathsf{lcs}(A, B)$ is the longest common subsequence length of A and B. In case where there is no solution, we use a convention that $Z = \bot$ and its length $|\bot|$ is -1. We remark that $\ell \geq |Z|$ always holds.

3 Space-efficient solution for STR-IC-LCS problem

In this section, we propose a space-efficient solution for the STR-IC-LCS problem.

Problem 1 (STR-IC-LCS problem)**.** *For any given strings A, B of length n and P, compute an STR-IC-LCS of A, B, and P.*

Theorem 2. *The STR-IC-LCS problem can be solved in $O(n^2)$ time and $O((\ell+1)(n-\ell+1))$ space where ℓ is the length of LCS of A and B.*

In Section 3.1, we explain an overview of our algorithm. In Section 3.2, we show a central technique for our space-efficient solution and Section 3.3 concludes with the detailed algorithm.

3.1 Overview of our solution

Our algorithm uses an algorithm for the STR-IC-LCS problem which was proposed by Deorowicz [5]. Firstly, we explain an outline of the algorithm. Let I_A be the set of minimal intervals over A which have P as a subsequence. Remark that I_A is linear-size since each interval cannot contain any other intervals. There exists a pair of minimal intervals $[\mathsf{b}_A, \mathsf{e}_A]$ over A and $[\mathsf{b}_B, \mathsf{e}_B]$ over B such that the length of an STR-IC-LCS is equal to the sum of the three values $\mathsf{lcs}(A[1..\mathsf{b}_A - 1], B[1..\mathsf{b}_B - 1])$, $|P|$, and $\mathsf{lcs}(A[\mathsf{e}_A + 1..n], B[\mathsf{e}_B + 1..n])$ (see also Fig. 1 for an example). First, the algorithm computes I_A and I_B and computes the sum of three values for any pair of intervals. If we have an LCS table d of size $n \times n$ such that $d(i, j)$ stores $\mathsf{lcs}(A[1..i], B[1..j])$ for any integers $i, j \in [1..n]$, we can check any LCS value between prefixes of A and B in constant time. It is known that this table can be computed in $O(n^2)$ time by using a simple dynamic programming. Since the LCS tables for prefixes and suffixes requires $O(n^2)$ space, the algorithm also requires $O(n^2)$ space.

Our algorithm uses a space-efficient LCS table by Nakatsu et al. [10] instead of the table d for computing LCSs of prefixes (suffixes) of A and B. The algorithm by Nakatsu et al. also computes a table by dynamic programming, but the table does not gives $\mathsf{lcs}(A[1..i], B[1..j])$ for several pairs (i, j). In the next part, we show a way to resolve this problem.

3.2 Space-efficient prefix LCS

First, we explain a dynamic programming solution by Nakatsu et al. for computing an LCS of given strings A and B. We give a slightly modified description in order to describe our algorithm.

For any integers $i, s \in [1..n]$, let $f_A(s,i)$ be the length of the shortest prefix $B[1..f_A(s,i)]$ of B such that the length of the longest common subsequence of $A[1..i]$ and $B[1..f_A(s,i)]$ is s. For convenience, $f_A(s,i) = \infty$ if no such prefix exists. The values $f_A(s,i)$ will be computed using dynamic programming as follows:

$$f_A(s,i) = \min\{f_A(s,i-1), j_{s,i}\},$$

where $j_{s,i}$ is the index of the leftmost occurrence of $A[i]$ in $B[f_A(s-1,i-1)+1..n]$. Let s' be the largest value such that $f_A(s',i) < \infty$ for some i, i.e, the s'-th row is the lowest row which has an integer value in the table f_A. We can see that the length of the longest common subsequence of A and B is s' (i.e., $\ell = \mathsf{lcs}(A,B) = s'$). See Fig. 2 for an instance of f_A. Due to the algorithm, we do not need to compute all the values in the table f_A for obtaining the length of an LCS. Let F_A be the sub-table of f_A such that $F_A(s,i)$ stores a value $f_A(s,i)$ if $f_A(s,i)$ is computed in the algorithm of Nakatsu et al. Intuitively, F_A stores the first $n-l+1$ diagonals of length at most l. Let $\langle i \rangle$ be the set of pairs in the i-th diagonal line ($1 \leq i \leq n$) of the table f_A:

$$\langle i \rangle = \{(s, i+s-1) \mid 1 \leq s \leq n-i+1\}.$$

Formally, $F_A(s,i) = \mathsf{undefined}$ if either

1. $s > i$,
2. $(s,i) \in \langle j \rangle$ $(j > n-\ell+1)$, or
3. $F_A(s-1,i-1) = \infty$ or $\mathsf{undefined}$.

Any other $F_A(s,i)$ stores the value $f_A(s,i)$. Since the lowest row number of each diagonal line $\langle j \rangle$ $(j > n-\ell+1)$ is less than ℓ, we do not need to compute values which is described by the second item. Actually, we do not need to compute the values in $\langle n-\ell+1 \rangle$ for computing the LCS since the maximum row number in the last diagonal line is also ℓ. However, we need the values on the last line in our algorithm. Hence the table F_A uses $O((\ell+1)(n-\ell+1))$ space (subtable which need to compute is parallelogram-shaped of height ℓ and base $n-\ell$). See Fig. 3 for an instance of F_A.

Now we describe a main part of our algorithm. Recall that a basic idea is to compute $\mathsf{lcs}(A[1..i], B[1..j])$ from F_A. If we have all the values on the table f_A, we can check the length $\mathsf{lcs}(A[1..i], B[1..j])$ as follows.

Observation 3. *The length of an LCS of $A[1..i]$ and $B[1..j]$ for any $i, j \in [1..n]$ is the largest s such that $f_A(s,i) \leq j$. If no such s exists, $A[1..i]$ and $B[1..j]$ have no common subsequence of length s.*

However, F_A does not store several integer values w.r.t. the second condition of $\mathsf{undefined}$ for some i and j. See also Fig. 3 for an example of the fact. In this example, we can see that $\mathsf{lcs}(A[1..7], B[1..4]) = f_A(3,7) = 3$ from the table f_A, but $F_A(3,7) = \mathsf{undefined}$ in F_A. In order to resolve this problem, we also define F_B (and f_B). Formally, for any integers $j, s \in [1..n]$, let $f_B(s,j)$ be the length of the shortest prefix $A[1..f_B(s,j)]$ of A such that the length of the longest common subsequence of $B[1..j]$ and $A[1..f_B(s,j)]$ is s. Our algorithm accesses the length of an LCS of $A[1..i]$ and $B[1..j]$ for any given i and j by using two tables F_A and F_B. The following lemma shows a key property for the solution.

Lemma 4. *Let s be the length of an LCS of $A[1..i]$ and $B[1..j]$. If $F_A(s,i) = \mathsf{undefined}$ then $F_B(s,j) \neq \mathsf{undefined}$.*

This lemma implies that the length of an LCS of $A[1..i]$ and $B[1..j]$ can be obtained if we have the two sparse tables (see also Fig. 4). Before we prove this lemma, we show the following property. Let U_{F_A} be the set of pairs (s,i) of integers such that $F_B(s,j) \neq \mathsf{undefined}$ where $F_A(s,i) = j$.

Lemma 5. *For any $1 \leq s \leq \mathsf{lcs}(A,B)$, there exists i such that $(s,i) \in U_{F_A}$.*

A = b c a a b a b c b

f_A

s \ i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	1	1	1	1	1	1	1	1
2		4	3	3	2	2	2	2	2
3			6	6	5	3	3	3	3
4				∞	7	6	5	4	4
5					∞	∞	7	7	5
6			∞			∞	∞	9	8
7							∞	∞	∞
8								∞	∞
9									∞

B = c b a c b a b b c

f_B

s \ i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	1	1	1	1	1	1	1	1
2		5	3	2	2	2	2	2	2
3			6	6	5	3	3	3	3
4				8	7	6	5	5	5
5					9	9	7	7	7
6			∞			∞	∞	9	8
7							∞	∞	∞
8								∞	∞
9									∞

Figure 2: The LCS-table f_A which is defined by Nakatsu et al. of $A = \texttt{bcdababcb}$. This figure also illustrates the table f_B of $B = \texttt{cbacbaaba}$.

A = b c a a b a b c b

F_A

s \ i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	1	1	1					
2		4	3	3	2				
3			6	6	5	3			
4				∞	7	6	5		
5						∞	7	7	
6								9	8
7									∞
8									
9									

B = c b a c b a b b c

F_B

s \ i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	1	1	1					
2		5	3	2	2				
3			6	6	5	3			
4				8	7	6	5		
5					9	9	7	7	
6						∞	∞	9	8
7									∞
8									
9									

Figure 3: A sparse table F_A of f_A for $A = \texttt{bcdababcb}$ and $B = \texttt{cbacbaaba}$ does not give $\mathsf{lcs}(A[1..i], B[1..j])$ for some (i, j).

Proof. Let $\ell = \mathsf{lcs}(A, B)$ and $A[i_1] \cdots A[i_\ell]$ be an LCS of A and B which can be obtained by backtracking over F_A. Suppose that $F_B(s, F_A(s, i_s)) = \mathsf{undefined}$ for some $s \in [1..\ell]$. Since $F_A(1, i_1) < \ldots < F_A(\ell, i_\ell)$, $F_B(s', F_A(s', i_{s'})) = \mathsf{undefined}$ for any $s' \in [s..\ell]$. However, $F_B(\ell, F_A(\ell, i_\ell))$ is not $\mathsf{undefined}$. Therefore, $F_B(s, F_A(s, i_s)) \neq \mathsf{undefined}$ for any $s \in [1..\ell]$. This implies that the statement holds. □

Now we are ready to prove Lemma 4 as follows.

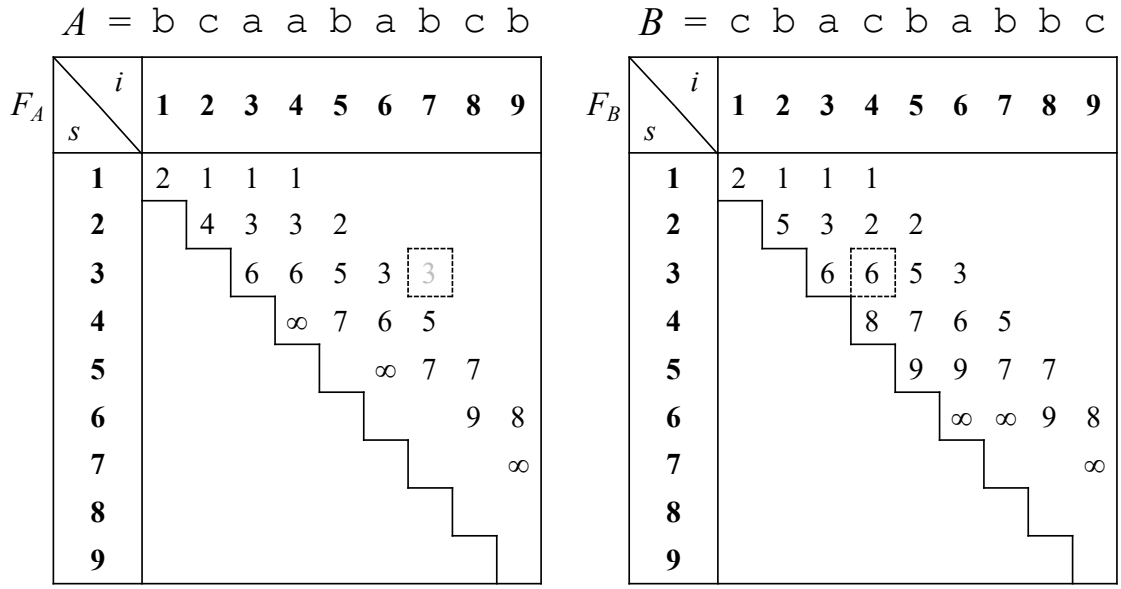

Figure 4: Due to Observation 3, $f_A(3,7)$ gives the fact that $\mathsf{lcs}(A[1..7], B[1..4]) = 3$. However, $F_A(3,7) = \mathsf{undefined}$. Then we can obtain the fact that $\mathsf{lcs}(A[1..7], B[1..4]) = 3$ by using F_B. Namely, $F_B(3,4)$ gives the LCS value.

of Lemma 4. Let $\ell = \mathsf{lcs}(A,B)$ and $X = A[i_1] \cdots A[i_\ell]$ be an LCS of A and B which can be obtained by F_A. $j_1, \ldots, j_\ell$ denotes the sequence of positions over B where $F_A(k, i_k) = j_k$ for any $k \in [1..\ell]$. Assume that $F_A(s,i) = \mathsf{undefined}$. Let m be the largest integer such that $i_{s+m} \leq i$ holds. If no such m exists, namely $i < i_1$, the statement holds since $F_A(s,i) \neq \mathsf{undefined}$. Due to Lemma 3, $F_A(s+m, i) > j$. Thus $j < j_{s+m}$ holds. On the other hand, we consider the table F_B (and f_B). Let $i' = f_B(s,j)$ and $i'' = f_B(s+m, j)$. Due to Lemma 3, $i' \leq i < i''$ holds. From Lemma 5, $F_B(s+m, j_{s+m}) \neq \mathsf{undefined}$. This implies that $F_B(s+m, j)$ $(= i'')$ is not $\mathsf{undefined}$. By the definition of X, $j_{s+m} - j \geq m - 1$. Notice that (s,j) is in $(j-s+1)$-th diagonal line. These facts imply that $F_B(s,j) \neq \mathsf{undefined}$. See also Fig. 5 for an illustration. □

3.3 Algorithm

First, our algorithm computes sets of minimal intervals I_A and I_B (similar to the algorithm by Deorowicz [5]). Second, compute the tables F_A and F_B for computing LCSs of prefixes, and the tables F_{A^R} and F_{B^R} for computing LCSs of suffixes (similar to the algorithm by Nakatsu et al. [10]). Third, for any pairs of intervals in I_A and I_B, compute the length of an LCS of corresponding prefixes/suffixes and obtain a candidate of the length of an STR-IC-LCS. As stated above, the first and the second steps are similar to the previous work. Here, we describe a way to compute the length of an LCS of prefixes on F_A and F_B in the third step. We can also compute the length of an LCS of suffixes on F_{A^R} and F_{B^R} by using a similar way.

We assume that I_A and I_B are sorted in increasing order of the beginning positions. Let $[\mathsf{b}_A(x)..\mathsf{e}_A(x)]$ and $[\mathsf{b}_B(y)..\mathsf{e}_B(y)]$ be a x-th interval in I_A and a y-th interval in I_B, respectively. We process $O(n^2)$-queries in increasing order of the beginning position of the intervals in I_A. For each interval $[\mathsf{b}_A(x)..\mathsf{e}_A(x)]$ in I_A, we want to obtain the length of an LCS of $A[1..\mathsf{b}_A(x)-1]$ and $B[1..\mathsf{b}_B(1)-1]$. For convenience, let $i_x = \mathsf{b}_A(x) - 1$ and $j_y = \mathsf{b}_B(y) - 1$. In the rest of this section, we use a pair (x,y) of integers to denote a prefix-LCS query (computing $\mathsf{lcs}(A[1..i_x], B[1..i_y])$). We will find the LCS by using Observation 3. Here, we describe how to compute prefix-LCS queries $(i_x, j_1), \ldots, (i_x, j_{|I_B|})$ in this order for a fixed i_x.

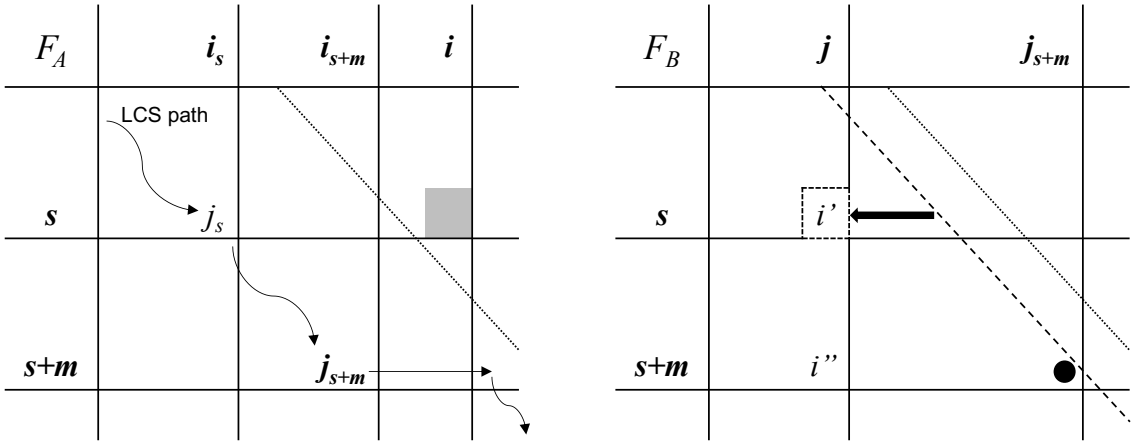

Figure 5: This figure shows an illustration for the proof of Lemma 4. The length s of an LCS of $A[1..i]$ and $B[1..j]$ cannot be obtained over F_A because $F_A(i, s) = \mathsf{undefined}$ (the highlighted cell). However, the length can be obtained by $F_B(s, j)$ over F_B. The existence of $F_B(s+m, j_{s+m})$ from an LCS path guarantees the fact that $F_B(s, j) \neq \mathsf{undefined}$.

Lemma 6. *All required prefix-LCS values for an interval* $[\mathsf{b}_A(x)..\mathsf{e}_A(x)]$ *in* I_A *and all intervals in* I_B *can be computed in* $O(n)$ *time.*

Proof. There exist two cases for each i_x. Formally, (1) $F_A[i_x, 1] \neq \mathsf{undefined}$ or (2) $F_A[i_x, 1] = \mathsf{undefined}$.

In the first case, we scan the i_x-th column of F_A from the top to the bottom in order to find the maximum value which is less than or equal to j_1. If such a value exists in the column, then the row number s_1 is the length of an LCS. After that, we are given the next prefix-LCS query (i_x, j_2). It is easy to see that $s_0 = \mathsf{lcs}(A[1..i_x], B[1..j_1]) \leq \mathsf{lcs}(A[1..i_x], B[1..j_2])$ since $j_1 < j_2$. This implies that the next LCS value is equal to s_0 or that is placed in a lower row in the column. This means that we can start to scan the column from the s_0-th row. Thus we can answer all prefix-LCSs for a fixed i_x in $O(n)$ time (that is linear in the size of I_B).

In the second case, we start to scan the column from the top $F_A[i_x, i_x - n - \ell + 1]$ (the first $i_x - n - \ell$ rows are $\mathsf{undefined}$). If $F_A[i_x, i_x - n - \ell + 1] \leq j_1$, then the length of an LCS for the first query (i_x, j_1) can be found in the table (similar to the first case) and any other queries $(i_x, j_2), \ldots, (i_x, j_{|I_B|})$ can be also answered in the similar way. Otherwise (if $F_A[i_x, i_x - n - \ell + 1] > j_1$), the length which we want may be in the "undefined" domain. Then we use the other table F_B. We scan the j_1-th column in F_B from the top to the bottom in order to find the maximum value which is less than or equal to i_x. By Lemma 4, such a value must exist in the column (if $\mathsf{lcs}(A[1..i_x], B[1..j_1]) > 0$ holds) and the row number s' is the length of an LCS. After that, we are given the next query (i_x, j_2). If $F_A[i_x, i_x - n - \ell + 1] \leq j_2$, then the length can be found in the table (similar to the first case). Otherwise (if $F_A[i_x, i_x - n - \ell + 1] > j_2$), the length must be also in the "undefined" domain. Since such a value must exist in the j_2-th column in F_B by Lemma 4, we scan the column in F_B. It is easy to see that $s' = \mathsf{lcs}(A[1..i_x], B[1..j_1]) \leq \mathsf{lcs}(A[1..i_x], B[1..j_2])$. This implies that the length of an LCS that we want to find is in lower row. Thus it is enough to scan the j_2-th column from the s'-th row to the bottom. Then we can answer the second query (i_x, j_2). Hence we can compute all LCSs for a fixed i_x in $O(n + \ell)$ time (that is linear in the size of I_B or the number of rows in the table F_B).

Therefore we can compute all prefix-LCSs for each interval in I_A in $O(n)$ time (since $n \geq \ell$). □

On the other hand, we can compute all required suffix-LCS values with computing prefix-LCS values. We want a suffix-LCS value of $A[\mathsf{e}_A(x)+1..n]$ and $B[\mathsf{e}_B(y)+1..n]$ $(1 \leq y \leq |I_B|)$ when we compute the length of an LCS of $A[1..\mathsf{b}_A(x) - 1]$ and $B[1..\mathsf{b}_B(y) - 1]$. Recall that we process all intervals of I_B in increasing order of the beginning positions when computing prefix-LCS values

Algorithm 1 Algorithm for computing the length of STR-IC-LCS

Input: A, B, P $(|A| = n, |B| = n, |P| = r)$
Output: l, C

compute I_A and I_B
compute F_A, F_B, F_{A^R}, and F_{B^R}
$\ell \leftarrow \mathsf{lcs}(A, B)$;
$l \leftarrow 0$;
for $i = 1$ to $|I_A|$ **do**
  $k_1^A \leftarrow 1$; $k_1^B \leftarrow 1$; $k_2^A \leftarrow \ell$; $k_2^B \leftarrow \ell$;
  for $j = 1$ to $|I_B|$ **do**
    $k_1 \leftarrow 0$; $k_2 \leftarrow 0$;
    compute $\mathsf{lcs}(A[1..\mathsf{b}_A(i) - 1], B[1..\mathsf{b}_B(j) - 1])$ // as k_1 by Algorithm 2
    compute $\mathsf{lcs}(A[\mathsf{e}_A(i) + 1..n], B[\mathsf{e}_B(j) + 1..n])$ // as k_2 by Algorithm 3
    if $k_1 + k_2 + r > l$ **then**
      $l \leftarrow k_1 + k_2 + r$
    end if
  end for
end for
return l

with a fixed interval of I_A. This means that we need to process all intervals of I_B in "decreasing order" when computing suffix-LCS values with a fixed interval of I_A. We can do that by using an almost similar way on F_{A^R} and F_{B^R}. The most significant difference is that we scan the $|A[\mathsf{e}_A(x) + 1..n]|$-th column of F_{A^R} from the ℓ-th row to the first row.

Overall, we can obtain the length of an STR-IC-LCS in $O(n^2)$ time in total. Also this algorithm requires space for storing all minimal intervals and tables, namely, requiring $O(n + (\ell+1)(n-\ell+1)) = O((\ell+1)(n-\ell+1))$ space in the worst case. Finally, we can obtain Theorem 2.

In addition, we can also compute an STR-IC-LCS (as a string), if we store a pair of minimal intervals which produce the length of an STR-IC-LCS. Namely, we can find a cell which gives the prefix-LCS value over F_A or F_B. Then we can obtain a prefix-LCS string by a simple backtracking (a suffix-LCS can be also obtained by backtracking on F_{A^R} or F_{B^R}). On the other hand, we can also use an algorithm that computes an LCS string in $O(n^2)$ time and $O(n)$ space by Hirschberg [6]. We conclude with supplemantal pseudocodes of our algorithm (see Algorithms 1,2, and 3).

4 Conclusions and future work

We have presented a space-efficient algorithm that finds an STR-IC-LCS of two given strings A and B of length n in $O(n^2)$ time with $O((\ell+1)(n-\ell+1))$ working space, where ℓ is the length of an LCS of A and B. Our method improves on the space requirements of the algorithm by Deorowicz [5] that uses $\Theta(n^2)$ space, irrespective of the value of ℓ.

Our future work for STR-IC-LCS includes improvement of the $O(n^2)$-time bound to, say, $O(n(n-\ell+1))$. We note that the algorithm by Nakatsu et al. [10] for finding (standard) LCS runs in $O(n(n-\ell+1))$ time. There also exists an $O(n\sigma + (\ell'+1)(n-\ell'+1)r)$-time solution for the STR-EC-LCS problem that runs fast when the length ℓ' of the solution is small [13], where $r = |P|$.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP21K17705 (YN), JP22H03551 (SI), JP20H04141 (HB), and by JST PRESTO Grant Number JPMJPR1922 (SI).

Algorithm 2 Computing $\mathsf{lcs}(A[1..\mathsf{b}_A(i)-1], B[1..\mathsf{b}_B(j)-1])$

```
1: for k ← k_1^A to ℓ do
2:   if F_A[b_A(i)−1, k] ≤ b_B(j)−1 then
3:     if F_A[b_A(i)−1, k+1] > b_B(j)−1 then
4:       k_1 ← k
5:       k_1^A ← k
6:       break
7:     end if
8:   else if F_A[b_A(i)−1, k] > b_B(j)−1 then
9:     if F_A[b_A(i)−1, k−1] = undefined then
10:      k_1^A ← k
11:      for k' = k_1^B to ℓ do
12:        if F_B[b_B(j)−1, k'] > b_A(i)−1 then
13:          k_1 ← 0
14:          k_1^B ← k'
15:          break
16:        else if F_B[b_B(j)−1, k'+1] > b_A(i)−1 then
17:          k_1 ← k'
18:          k_1^B ← k'
19:          break
20:        end if
21:      end for
22:    else
23:      k_1 ← 0
24:      k_1^A ← k
25:      break
26:    end if
27:  end if
28: end for
```

Algorithm 3 Computing $\mathsf{lcs}(A[\mathsf{e}_A(i)+1..n], B[\mathsf{e}_B(j)+1..n])$

```
1: for k = k_2^A to 1 do
2:   if F_{A^R}[n − e_A(i), k] ≤ n − e_B(j) then
3:     k_2 ← k
4:     k_2^A ← k
5:     break
6:   else if F_{A^R}[n − e_A(i), k] > n − e_B(j) then
7:     if F_{A^R}[n − e_A(i), k−1] = undefined then
8:       k_2^A ← k
9:       for k' = k_2^B to 1 do
10:        if F_{B^R}[n − e_B(j), k'] ≤ n − e_A(i) then
11:          k_2 ← k'
12:          k_2^B ← k'
13:          break
14:        else if F_{B^R}[n − e_B(j), k'−1] = undefined then
15:          k_2 ← 0
16:          k_2^B ← k'
17:          break
18:        end if
19:      end for
20:    end if
21:  end if
22: end for
```

References

[1] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *FOCS 2015*, pages 59–78, 2015.

[2] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.

[3] Y.-C. Chen and K.-M. Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, 21(3):383–392, Apr 2011.

[4] F. Y. Chin, A. D. Santis, A. L. Ferrara, N. Ho, and S. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4):175 – 179, 2004.

[5] S. Deorowicz. Quadratic-time algorithm for a string constrained LCS problem. *Information Processing Letters*, 112(11):423 – 426, 2012.

[6] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341—-343, 1975.

[7] K. Kuboi, Y. Fujishige, S. Inenaga, H. Bannai, and M. Takeda. Faster STR-IC-LCS computation via RLE. In J. Kärkkäinen, J. Radoszewski, and W. Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPIcs*, pages 20:1–20:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[8] J.-J. Liu, Y.-L. Wang, and Y.-S. Chiu. Constrained Longest Common Subsequences with Run-Length-Encoded Strings. *The Computer Journal*, 58(5):1074–1084, 03 2014.

[9] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[10] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, 18:171–179, 1982.

[11] Y.-T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173 – 176, 2003.

[12] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[13] K. Yamada, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster STR-EC-LCS computation. In *SOFSEM 2020*, volume 12011 of *Lecture Notes in Computer Science*, pages 125–135. Springer, 2020.