Fast Algorithms for the Concatenated Longest Common Subsequence Problem with the Linear-space S-table[*]

Bi-Shiang Lin[a], Kuo-Tsung Tseng[b], Chang-Biau Yang[a†]and Kuo-Si Huang[c]
[a]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
[†]cbyang@cse.nsysu.edu.tw
[b]Department of Shipping and Transportation Management
National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan
[c]Department of Information Management
National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan

Abstract

Given two sequences A and B of lengths m and n, respectively, the consecutive suffix alignment *(CSA) problem is to compute the* longest common subsequence *(LCS) between A and each suffix of B. A two-dimensional S-table is constructed for solving the CSA problem. The linear-space S-table consists of the first row of the S-table and the changes between every two consecutive rows. Suppose that $A = A^{(1)}A^{(2)}$ (concatenation of two substrings), and we are given the S-table of $A^{(2)}$ and B, and the alignment result of $A^{(1)}$ and B. The* concatenated LCS *(CoLCS) problem is to find the alignment result of A and B. By using the linear-space S-table, instead of the 2-D S-table, we first propose an $O(n \log n)$-time algorithm to solve the CoLCS problem. Then, we propose a more efficient algorithm for the CoLCS problem, in $O(n)$ time, with the technique of set find and union.*

1 Introduction

The *longest common subsequence* (LCS) problem [2, 6, 8, 10, 12, 13, 15, 20, 22, 30] is a fundamental method for estimating the similarity between sequences. The LCS problem has been extensively studied for several decades since 1970. The LCS problem can be solved in O(mn) time [13] by the dynamic programming approach, where m and n denote the lengths of the two input sequences, respectively. Lots of variant LCS problems were proposed, such as the *merged longest common subsequence* problem [14, 23, 29], which considers the LCS with the merged sequence, and the *constrained LCS* [4, 5, 9, 24, 27, 28], which computes the LCS with the constrained sequence.

The *consecutive suffix alignment* (CSA) problem is one of the variant LCS problems. Given two sequences A and B, the CSA problem is to compute the LCS between A and each suffix of B [16], where a suffix of a string means a substring starting at a certain position and ending at the last position. The *S-table* can be used to solve the CSA problem. The CSA problem can be used in various applications, such as the common substring alignment problem [18, 19], cyclic string comparison between two strings or between A and each suffix of B [17, 21, 25]. In 2003, Landau *et al.* [18] proposed a linear time algorithm with the given S-table to solve the common substring alignment problem.

In 2004, Landau *et al.* [16] proposed two algorithms to solve the CSA problem. One solves the problem in O(nl) time and space with constant alphabets, and the other solves the problem in O(nl+$n|\log\Sigma|$) time and O(n) space, where $|\Sigma|$, l denote the alphabet size and the length of LCS, respectively. In 2005, Alves *et al.* [3] proposed another algorithm with O(mn) time and O(n) space for the CSA problem. In addition, Alves *et al.* [3] proposed the linear-space S-table, which consists of the first row of the S-table and the changes between every two consecutive rows.

Let $A = A^{(1)}A^{(2)}$ (concatenation of two substrings). And we already have the S-table of $A^{(2)}$ and B, and the alignment result of $A^{(1)}$ and B.

[*]This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 104-2221-E-110-018-MY3.

[†]Corresponding author.

The *concatenated LCS* (CoLCS) problem is defined to calculate the alignment result of A and B. In this paper, we proposes two algorithms in $O(n \log n)$ and $O(n)$ time for solving the CoLCS problem with the linear-space S-table, instead of the 2-D S-table.

The organization of this paper is given as follows. Section 2 introduces the preliminary knowledge of the LCS and CSA problems, and the S-table. Next, our algorithms for the CoLCS problem are proposed in Section 3. Finally, the conclusions are given in Section 4.

2 Preliminaries

A sequence of characters is denoted as an upper-case letter, such as A or B. Taking sequence A as an example, the notations used in this paper are listed below.

- $A = a_1 a_2 \cdots a_m$.
- $|A|$: the length of sequence A.
- a_i: the ith character or element of A.
- $i..j$: an index range from position i to j.
- $A_{i..j}$: the substring of A from index i to j. Note that $A_{i..j} = \emptyset$ if $i > j$.

A subsequence of A is obtained by deleting an arbitrary number of characters (not necessarily consecutive) in A. For example, $A =$ `tctgatggt`, the subsequences of A may be `tctgatggt`, `catt`, `ctga`, `tatgt`, `a`, and so on. The longest common subsequence problem is defined as follows.

Definition 1. (LCS) *Given two sequences A and B with lengths m and n, respectively, the longest common subsequence (LCS) problem is to find the common subsequence between A and B with the maximal length.*

For example, suppose $A =$ `cggattctgt` and $B =$ `tctgatggt`. The LCS of A and B, denoted as $LCS(A, B)$, is `cgatgt` with length 6. The LCS problem can be solved by the grid directed acyclic graph (GDAG) [19] as shown in Figure 1.

Definition 2. ($P_G(i, j)$) *For $0 \le i \le m$ and $0 \le j \le n$, $P_G(i, j)$ is the value of the highest weight path from $G(0, 0)$ to $G(i, j)$.*

With the GDAG, the length of LCS is equal to $P_G(m, n)$. The LCS problem can be solved through the dynamic programming (DP) approach in $O(mn)$ time by Equation 1 [30].

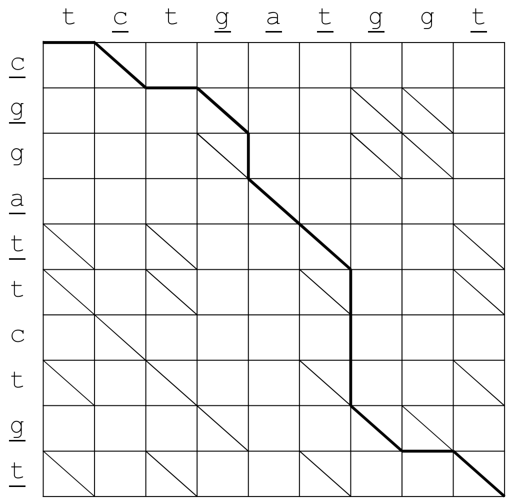

Figure 1: The grid directed acyclic graph (GDAG) for solving the LCS problem with $A =$ `cggattctgt` and $B =$ `tctgatggt`. Here, the path formed with thick lines is the LCS solution (`cgatgt`). Note that diagonal edges with weight 0 are not shown.

$P_G(i, j) =$

$$\max \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ P_G(i-1, j-1) + 1 & \text{if } a_i = b_j, \\ \max \begin{cases} P_G(i-1, j) \\ P_G(i, j-1) \end{cases} & \text{if } a_i \ne b_j. \end{cases} \quad (1)$$

2.1 The Consecutive Suffix Alignment Problem and the S-table

Definition 3. (consecutive suffix alignment) *Given two sequences A and B, the consecutive suffix alignment (CSA) problem is to compute the alignment of A and each suffix of B.*

With the above DP approach for the LCS problem, $|LCS(A, B_{1..j})|$ can be computed in $O(mn)$ time for all $1 \le j \le n$. The naïve method for the CSA problem with the DP approach requires $O(mn^2)$ time by computing each $|LCS(A, B_{i..j})|$, for $0 \le i \le j \le n$. However, it is inefficient. In the GDAG, the CSA problem can be transformed to finding the maximal weight path from $G(0, i)$ to $G(m, j)$ for $0 \le i \le j \le n$.

Definition 4. ($C_G(i, j)$) *For $0 \le i \le j \le n$, $C_G(i, j)$ is the maximal weight of all the paths from $G(0, i)$ to $G(m, j)$.*

Table 1: The matrix C_G with $A = \texttt{ttct}$ and $B = \texttt{tctgatggt}$.

$i \backslash j$	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	3	3	3	3	3	3
1	0	0	1	2	2	2	2	2	2	3
2	0	0	0	1	1	1	2	2	2	3
3	0	0	0	0	0	0	1	1	1	2
4	0	0	0	0	0	0	1	1	1	2
5	0	0	0	0	0	0	1	1	1	2
6	0	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	0

Table 2: The S-table S of $A = \texttt{ttct}$ and $B = \texttt{tctgatggt}$, where the starting index means of the position of B, and the column of D means that the number is the first occurrence in the row.

		Length				
		0	1	2	3	D
Starting index	0	0	1	2	3	
	1	1	2	3	$\underline{9}$	$\underline{9}$
	2	2	3	$\underline{6}$	9	$\underline{6}$
	3	3	6	9	∞	∞
	4	$\underline{4}$	6	9	∞	$\underline{4}$
	5	$\underline{5}$	6	9	∞	$\underline{5}$
	6	6	9	∞	∞	∞
	7	$\underline{7}$	9	∞	∞	$\underline{7}$
	8	$\underline{8}$	9	∞	∞	$\underline{8}$
	9	9	∞	∞	∞	∞

With Definition 4, it is clear that $C_G(i, j) = |LCS(A, B_{i+1..j})|$. Table 1 shows an example of C_G with $A = \texttt{ttct}$ and $B = \texttt{tctgatggt}$.

Some properties in C_G are listed as follows.

- For each row in C_G, the value starts from 0.
- For each row in C_G, the values from left to right are nondecreasing.
- For each row in C_G, the difference between two consecutive values is either 0 or 1.

With the above properties, C_G can be represented by Table 2, denoted as S.

Definition 5. [3] (S-table) *For $0 \leq i \leq n$, $S_{i,0} = i$. For $0 \leq i \leq n$ and $0 < j \leq L$, where L is the maximal value in C_G, $S_{i,j}$ is the minimum of k for $C_G(i, k) = j$. If no such k exists, $S_{i,j} = \infty$.*

The first element in $S_{i,*}$ (row i of S) is $S_{i,0} = i$, and each remaining element in $S_{i,*}$ records the index of the column which is the leftmost of each number appears in row i of C_G. For example, in row 3 of C_G, the leftmost 1 appears at the column 6, so $S_{3,1} = 6$. Alves *et al.* proposed and proved the following property of S-table [3].

Theorem 1. [3] *For $0 \leq i < n$ in S,*

1. *Exactly one element of $S_{i,*}$ does not appear in $S_{i+1,*}$, which is $S_{i,0} = i$.*
2. *At most one element with a finite value in $S_{i+1,*}$ does not appear in $S_{i,*}$.*

Definition 6. [3] (D) *For $0 < i \leq n$, d_i records the element which appears in $S_{i,*}$ but not in $S_{i-1,*}$. If there is no such new element, we set $d_i = \infty$.*

An example of D is shown in the rightmost column of Table 2. The S-table can be constructed from g the $S_{0,*}$ and D. Therefore, the solution of the CSA problem can be represented with the S-table, or $S_{0,*}$ and D [3]. In the following, the *linear-space S-table* means the first row of the S-table ($S_{0,*}$) and D.

2.2 Solving the Concatenated LCS Problem with the S-table

In this subsection, we use an example to explain how to solve the LCS problem with multiple common substrings by the S-table [18, 19]. Suppose we are given two strings $A = \texttt{cggattctgt}$ and $B = \texttt{tctgatggt}$, where A is formed by concatenating three substrings $A^{(1)} = \texttt{cgga}$, $A^{(2)} = \texttt{ttct}$ and $A^{(3)} = \texttt{gt}$. In other situations, $A^{(r)}$ may repeat several times, but not consecutively, to form a longer sequence A. Note the S-table of $A^{(2)}$ and B has been already established in Table 2. Figure 2 shows the GDAG of A and B, which is composed of three subgraphs, corresponding to $A^{(1)}$, $A^{(2)}$ and $A^{(3)}$, respectively. The alignment result of the first subgraph can be viewed as the input of the second subgraph, and the alignment result of the second subgraph can be viewed as the input of the third subgraph.

Let $G^{(r)}$ denote subgraph r, whose input and output are denoted as $I^{(r)}$ and $O^{(r)}$, respectively. In addition, let $S^{(r)}$ denote the S-table of $A^{(r)}$ and B. The goal is to get the output $O^{(r)}$ with the input $I^{(r)}$ and S-table $S^{(r)}$. The following DP formula can be easily obtained [18, 19].

$$O_j^{(r)} = \max\{I_i^{(r)} + C_{G^{(r)}}(i, j)\}, \text{ for } 0 \leq i \leq j. \quad (2)$$

For example, $O_3^{(2)} = 3 = \max\{0 + 3, 0 + 2, 1 + 1, 1 + 0\}$. The value of $O_3^{(2)}$ comes from the input

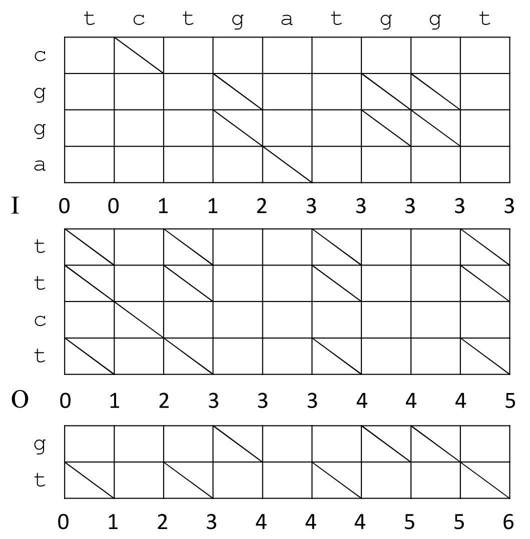

Figure 2: The GDAG composed of three subgraphs with $A^{(1)} = \texttt{cgga}$, $A^{(2)} = \texttt{ttct}$, $A^{(3)} = \texttt{gt}$ and $B = \texttt{tctgatggt}$.

$I^{(2)}$ and $G^{(2)}$. The case from $I_3^{(2)}$ can be ignored since $I_2^{(2)} = I_3^{(2)} = 1$ and $C_{G^{(2)}}(2,3) \geq C_{G^{(2)}}(3,3)$. Similarly, $I_1^{(2)}$ can be ignored since $I_0^{(2)} = I_1^{(2)} = 0$. As another example, $O_6^{(2)} = 4 = \max\{0+3, 0+2, 1+2, 1+1, 2+1, 3+1, 3+0\}$. Thus, only the leftmost position of $I^{(r)}$ with value k is needed to compute $O^{(r)}$. Let PI_k denote the smallest index in $I^{(r)}$ with value k, and PO_i denote the smallest index in $O^{(r)}$ with value i. In this example, $PI = \langle 0, 2, 4, 5\rangle$ and $PO = \langle 0, 1, 2, 3, 6, 9\rangle$. Now, $O^{(r)}$ can be represented as PO in Equation 3 [18].

$$PO_i = \min\{j | k + C_{G^{(r)}}(PI_k, j) = i, \quad 0 \leq k \leq i \text{ and } 0 \leq j \leq n.\} \tag{3}$$

With the S-table $S^{(r)}$, Equation 3 can be transformed into Equation 4.

$$PO_i = \min_{0 \leq k \leq i} \{S^{(r)}_{PI_k, i-k}\}, \text{ if } S^{(r)}_{PI_k, i-k} \text{ exists.} \tag{4}$$

For example, the smallest column index of value 4 in $O^{(2)}$, denoted by PO_4, is obtained by $\min\{S^{(2)}_{PI_0,4}, S^{(2)}_{PI_1,3}, S^{(2)}_{PI_2,2}, S^{(2)}_{PI_3,1}, S^{(2)}_{PI_4,0}\}$ $= \min\{S^{(2)}_{0,4}, S^{(2)}_{2,3}, S^{(2)}_{4,2}, S^{(2)}_{5,1}, S^{(2)}_{\infty,0}\} = \min\{-, 9, 9, 6, -\} = 6$. It means that $|LCS(A^{(1)}, B_{1..2})| + |LCS(A^{(2)}, B_{3..9})| = 1 + 3 = 4$, $|LCS(A^{(1)}, B_{1..4})| + |LCS(A^{(2)}, B_{5..9})| = 2 + 2 = 4$, and $|LCS(A^{(1)}, B_{1..5})| + |LCS(A^{(2)}, B_{6..6})| = 3 + 1 = 4$. And 6 is the leftmost index of B to get LCS length 4.

Definition 7. (M) *$M_{k,i} = S_{PI_k, i-k}$, for $0 \leq k \leq |PI| - 1$ and $k \leq i \leq k + L$ if such $S_{PI_k, i-k}$ exists.*

With the definition of matrix M, $PO_i = \min_{0 \leq k \leq |PI|-1}\{M_{k,i}\}$, for $0 \leq i \leq L$. The matrix M is shown in Table 3. The computation of PO is equivalent to finding the minimum of each column in M.

Table 3: The matrix M, where the row index means that of M, and each number in the bottom row is the column minimum.

M		Length 0	1	2	3	4	5
Row index	0	0	1	2	3		
	1		2	3	6	9	
	2			4	6	9	
	3				5	6	9
Minimum		0	1	2	3	6	9

To find the column minimum, the brute-force method needs O(nl) time to examine all the numbers, where l denotes the length of $LCS(A^{(r-1)} + A^{(r)}, B)$. Note that the symbol + means the concatenation strings $A^{(r-1)}$ and $A^{(r)}$. The matrix M has been proved to be a *totally monotone matrix* [18]. Therefore, a recursive algorithm, named SMAWK and proposed by Aggarwal *et al.* [1], can find the column minimum of a totally monotone matrix in O(l) time. With the S-table $S^{(r)}$ and the input $I^{(r)}$, the alignment of $G^{(r)}$ can be computed in O(l) time, instead of the original DP approach in O(mn) time.

In summary, given two substrings $A^{(1)}$ and $A^{(2)}$ and one string B with the S-table $S^{(2)}$ of $A^{(2)}$ and B, the *concatenated LCS* (CoLCS) problem is to find the LCS length of $A^{(1)} + A^{(2)}$ and B. It can be solved in O(l) time [18].

3 Our Algorithms for the Concatenated LCS Problem

In this section, we propose two new algorithms for solving the CoLCS problem in O($n \log n$) and O(n) time with the linear-space S-table: $S_{0,*}$ and D, instead of using the whole S-table.

3.1 The Alignment with the Linear-Space S-table

For easy explanation, we denote the infinity symbol ∞ mentioned in S-table and D as ∞_1, ∞_2, $\cdots$, and so on. Therefore, Table 2 is modified and shown in Table 4.

Table 4: The modified S-table and D with $A =$ `ttct` and $B =$ `tctgatggt`, where the starting index means of the position of B, and the value in column D means that the number is the first occurrence in the row.

S-table		Length 0	1	2	3	D
	0	0	1	2	3	
	1	1	2	3	$\underline{9}$	$\underline{9}$
	2	2	3	$\underline{6}$	9	$\underline{6}$
	3	3	6	9	$\underline{\infty_1}$	$\underline{\infty_1}$
Starting index	4	$\underline{4}$	6	9	∞_1	$\underline{4}$
	5	$\underline{5}$	6	9	∞_1	$\underline{5}$
	6	6	9	∞_1	$\underline{\infty_2}$	$\underline{\infty_2}$
	7	$\underline{7}$	9	∞_1	∞_2	$\underline{7}$
	8	$\underline{8}$	9	∞_1	∞_2	$\underline{8}$
	9	9	∞_1	∞_2	$\underline{\infty_3}$	$\underline{\infty_3}$

The modified computation matrix M is shown in Table 5(a). Clearly, the same result is obtained if only the finite values are considered when the column minimums in M are computed. The finite values are considered as the output. The minimum of column 6 is ∞_1, so we can ignore it. The output of Table 5(a) is identical to Table 3.

Property 1. *Once a number k appears in $S_{i,*}$, k must appear in $S_{j,*}$ for $i \leq j \leq k$.*

Definition 8. *Let $C_{k,j}$ denote the minimum of $M_{0..k,j}$, for $0 \leq k \leq |PI| - 1$ and $0 \leq j \leq k + L$. And, let h_k denote the maximum of $M_{k,*} \setminus C_{k-1,*}$ ($M_{k,*}$ with excluding $C_{k-1,*}$), where the symbol $\setminus$ denotes the set difference operation.*

For example, the matrix C is shown in Table 5(b). And, $h_1 = 9$, $h_2 = \infty_1$ and $h_3 = 6$. With the above definition, the alignment result $PO = C_{|PI|-1,*} = C_{3,*} = \langle 0, 1, 2, 3, 6, 9, \infty_1 \rangle$. We present a property of two consecutive $C_{k-1,*}$ and $C_{k,*}$ as follows.

Theorem 2. *$C_{k-1,*} \cup \{h_k\} = C_{k,*}$, for $1 \leq k \leq |PI| - 1$.*

Proof. Let j be the smallest index for $C_{k-1,j} > M_{k,j}$. We can divide $C_{k,*}$ into two parts by index j as follows.

1. $0 \leq i < j$. In this case, $C_{k-1,i} \leq M_{k,i}$. Thus, $C_{k,i} = \min\{C_{k-1,i}, M_{k,i}\} = C_{k-1,i}$.
2. $j \leq i \leq k + L$. Because $C_{k-1,j} > M_{k,j}$, we have $C_{k,j} = \min\{C_{k-1,j}, M_{k,j}\} = M_{k,j}$. The value of $C_{k-1,i}$ comes from one number in rows $PI_0, PI_1, \cdots, PI_{k-1}$ of S. Because $C_{k-1,i} \geq PI_k$, with Property 1, $C_{k-1,i}$ appears in $M_{k,*}$ after $M_{k,j}$ for all i. Therefore, $C_{k,i+1} = C_{k-1,i}$.

Thus, $C_{k-1,*} \cup \{h_k\} = C_{k,*}$, where $h_k = M_{k,j}$. □

Table 5: The modified matrix M and C, where $PI = \langle 0, 2, 4, 5 \rangle$. (a) The matrix M, where each number in the bottom is the column minimum. (b) The matrix C.

(a)

M		Length 0	1	2	3	4	5	6
	0	0	1	2	3			
Row index	1		2	3	6	9		
	2			4	6	9	∞_1	
	3				5	6	9	∞_1
Minimum		0	1	2	3	6	9	∞_1

(b)

C		Length 0	1	2	3	4	5	6
	0	0	1	2	3			
Row index	1	0	1	2	3	9		
	2	0	1	2	3	9	∞_1	
	3	0	1	2	3	6	9	∞_1

For example in Table 5, $C_{1,*} = \{0, 1, 2, 3, 9\} = \{0, 1, 2, 3\} \cup \{9\} = C_{0,*} \cup \{9\}$, where $C_{0,*} = S_{0,*}$. $C_{2,*} = \{0, 1, 2, 3, 9\} \cup \{\infty_1\}$ and $C_{3,*} = C_{2,*} \cup \{6\}$, where 6 is the maximum of $M_{3,*} \setminus C_{2,*}$.

With the above properties and $PO = C_{|PI|-1,*}$, we can compute PO from $C_{0,*}$ sequentially where $C_{0,*} = S_{0,*}$. The alignment result PO consists of $S_{0,*}$ and h_k for $1 \leq k \leq |PI| - 1$. Since $S_{0,*}$ has already been given, we focus on finding h_k. We first propose an $O(n \log n)$-time algorithm, and then a linear time algorithm.

3.2 An $O(n \log n)$-time Algorithm

We first find the value of h_k in a sequential method with $k = 1$ to $|PI| - 1$ sequentially.

Lemma 1. *$S_{i,*}$ consists of the L largest numbers in $S_{0,*} \cup D_{1..i}$, where $D_{i..j}$ denotes $\{d_i, d_{i+1}, \cdots, d_j\}$.*

Proof. Each element in $S_{i,*}$ is greater than or equal to i. With Theorem 1, the smallest number of $S_{i-1,*}$ does not appear in $S_{i,*}$. Then, with $|S_{i,*}| = L$, the elements of $S_{i,*}$ are the L largest numbers in $S_{0,*} \cup D_{1..i}$. □

For example in Table 4, $S_{2,*}$ consists of the four largest numbers in $\{0, 1, 2, 3\} \cup \{9, 6\}$.

Theorem 3. *$h_k = \max(D_{1..PI_k} \setminus H_{1..k-1})$, for $1 \le k \le |PI| - 1$.*

Proof. By Definition 7, $M_{k,i} = S_{PI_k,i-k}$. With ignoring the detailed column index and applying the set concept, we have $M_{k,*} = S_{PI_k,*}$. By Definition 8 and Lemma 1, h_k is the maximum of $(S_{0,*} \cup D_{1..PI_k}) \setminus C_{k-1,*}$. Since $C_{0,*} = S_{0,*} \subseteq C_{k-1,*}$, we have that h_k is the maximum of $D_{1..PI_k} \setminus C_{k-1,*}$. By Theorem 2, $C_{k-1,*} \cup \{h_k\} = C_{k,*}$, so $C_{k-1,*} = C_{0,*} \cup \{h_1\} \cup \{h_2\} \cup \cdots \cup \{h_{k-1}\}$. We get

$$\begin{aligned} h_k &= \max(D_{1..PI_k} \setminus \{h_1, h_2, \cdots, h_{k-1}\}) \\ &= \max(D_{1..PI_k} \setminus H_{1..k-1}). \end{aligned} \tag{5}$$

□

By Theorem 3, we can use a sequential method to find h_k by querying the range maximum of D, and remove h_k from D after finding. Take Tables 4 and 5 as an example, where $PI = \langle 0, 2, 4, 5\rangle$ and $M_{0,*} = \langle 0, 1, 2, 3\rangle$. The sequential process is described as follows.

(1) $PI_1 = 2$, so we find h_1 in $\max(D_{1..2}) = 9$, and remove 9.

(2) $PI_2 = 4$ and $\max(D_{1..4}) = \infty_1$, so $h_2 = \infty_1$ and we remove ∞_1.

(3) $PI_3 = 5$ and $\max(D_{1..5}) = 6$, so $h_3 = 6$. Therefore, $PO = C_{3,*} = \langle 0, 1, 2, 3, 6, 9, \infty_1\rangle$.

The range maximum query and single point update (removal) of D with the segment tree structure requires $O(\log n)$ time for each operation [7] . Thus, the problem for finding the LCS of $A^{(r-1)} + A^{(r)}$ and B needs $O(|PI| \log n) = O(n \log n)$ time, when PI, $S^{(r)}_{0,*}$ (row 0 of S-table of $A^{(r)}$ and B) and $D^{(r)}$ are given. The algorithm is presented in Algorithm 1.

Algorithm 1 An O($n \log n$)-time algorithm

```
Input: $PI$, $S_{0,*}$ and $D$
Output: $PO$
1: $PO = S_{0,*}$ // insert each of $S_{0,*}$ into $PO$
2: for $k = 1$ to $|PI| - 1$ do
3:     $i = \max(D_{1..PI_k})$ // $i$ is the index of the range maximum
4:     insert $d_i$ into $PO$ // $h_k = d_i$
5:     $d_i = -\infty$ // remove $d_i$ from $D$
6: return $PO$
```

3.3 An O(n)-time Algorithm

In Section 3.2, we compute h_k with the range maximum of $D_{1..PI_k}$, for $1 \le k \le |PI| - 1$, and remove h_k after finding. Now we focus on whether the number d_i will become the value of one h_k or not.

Definition 9. *$nextPI(d_i)$ is the smallest PI_k such that $i \le PI_k$.*

The $nextPI$ of Table 4 is shown in Table 6, where $PI = \langle 0, 2, 4, 5\rangle$. For example, $nextPI(d_1) = nextPI(9) = 2$ means that the smallest PI_k satisfying $1 \le PI_k$ is 2.

Table 6: An example of $nextPI$. If $nextPI(d_i)$ does not exist, we keep it empty. $PI = \langle 0, 2, 4, 5\rangle$ is underlined in column i.

i	d_i	$nextPI(d_i)$
1	9	2
$\underline{2}$	6	2
3	∞_1	4
$\underline{4}$	4	4
$\underline{5}$	5	5
6	∞_2	
7	7	
8	8	
9	∞_3	

With the preprocessing of $nextPI$, we explain how to compute h_k for $1 \le k \le |PI| - 1$. We check the numbers in $D = \{d_1, d_2, \cdots, d_n\}$ with the decreasing order of the d_i value. If $nextPI(d_i)$ is empty, we ignore it. The computation process is demonstrated as follows.

(1) $nextPI(\infty_3)$ and $nextPI(\infty_2)$ are empty, so we ignore them.

(2) $nextPI(\infty_1) = 4 = PI_2$. ∞_1 appears in the S-table after row $PI_1 = 2$. So ∞_1 should be the new member from $C_{1,*}$ to $C_{2,*}$. In other words, $h_2 = \max(D_{1..PI_2}) = \max(D_{1..4}) = \infty_1$, because we check the numbers of D in decreasing order.

(3) $nextPI(9) = 2 = PI_1$. So $h_1 = 9$.

(4) $nextPI(8)$ and $nextPI(7)$ are empty, so we ignore them.

(5) $nextPI(6) = 2$. We find $PI_1 = 2$, and h_1 has been already determined, so we check next of PI_1. Again, we find $PI_2 = 4$, and h_2 has been already determined, so we check PI_3. Thus, we have $h_3 = 6$.

(6) We finally get $h_1 = 9$, $h_2 = \infty_1$ and $h_3 = 6$, and $PO = S_{0,*} \cup H_{1..3} = \langle 0, 1, 2, 3, 6, 9, \infty_1\rangle$.

Since we check the numbers in D from the largest to the smallest, by Theorem 2, the above process can correctly find which h_k should be of the value d_i.

When we examine d_i, we use the *union-find* data structure [11, 26] to check whether PI_k and

h_k have been determined or not. If PI_k and h_k have been determined, we have to try the next, PI_{k+1} and h_{k+1}. The operations in the union-find data structure are listed as follows.

- $make(x, C)$: Create a new set named C containing exactly x.
- $find(x)$: Find the name of the set containing x.
- $union(x, y, C)$: Unite the set containing x and the set containing y into a new set named C.

In the union-find data structure, each number in PI except PI_0 is initially in a unique set, implemented by $make(PI_k, k)$ for $1 \le k \le |PI| - 1$. We also use $make(\infty, |PI|)$ to set the boundary. Our algorithm for finding PO is presented in Algorithm 2, where D is sorted in decreasing order.

Algorithm 2 An O(n)-time algorithm

```
Input: $PI$, $S_{0,*}$, $D$ and $nextPI$, where $D$ is sorted in decreasing
Output: $PO$
1: $PO = S_{0,*}$ // insert each of $S_{0,*}$ into $PO$
2: for $k = 1$ to $|PI| - 1$ do
3:     $make(PI_k, k)$
4: $make(\infty, |PI|)$ // set the boundary
5: for $d_i \in D$ from the largest to the smallest number do // decreasing order, achieved by bucket sort
6:     if $nextPI(d_i)$ exists and $find(nextPI(d_i)) \ne |PI|$ then
7:         set $k = find(nextPI(d_i))$
8:         insert $d_i$ into $PO$ // $h_k = d_i$
9:         $union(PI_k, PI_{k+1}, find(PI_{k+1}))$
10: return $PO$
```

Figure 3 shows an example of the union-find process, with Table 4 and $PI = \langle 0, 2, 4, 5\rangle$. In this case, $|PI| = 4$ is the boundary number. We start from checking the largest number in D, which is ∞_3. The detailed steps are shown as follows.

1. $nextPI(\infty_3)$ and $nextPI(\infty_2)$ are empty, so skip them.
2. $nextPI(\infty_1) = PI_2 = 4$, and $k = find(4) = 2 \ne |PI| = 4$, so we have $h_2 = \infty_1$ and $union(PI_2, PI_3, find(PI_3)) = union(PI_2, PI_3, 3)$. In this situation, h_2 with PI_2 has been determined. If h_2 is desired to be set next time, $union(PI_2, PI_3, 3)$ guarantees to set h_3 with PI_3, instead of h_2.

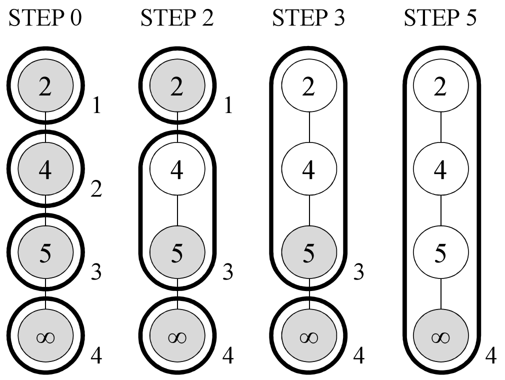

Figure 3: An example of the union-find process. Each thin circle is an element, and each bold circle is a set. The number beside each set is the name of the set.

In other words, when either h_2 or h_3 may be set next time, we always set h_3.

3. $nextPI(9) = PI_1 = 2$, and $k = find(2) = 1 \ne |PI| = 4$, so we have $h_1 = 9$ and $union(PI_1, PI_2, find(PI_2)) = union(PI_1, PI_2, 3)$. After $union(PI_1, PI_2, 3)$ is performed, if one of h_1, h_2 and h_3 is desired to be set next time, we always set h_3.
4. $nextPI(8)$ and $nextPI(7)$ are empty, so skip them.
5. $nextPI(6) = 4$ and $k = find(4) = 3 \ne |PI| = 4$. So $h_3 = 6$, and $union(PI_3, PI_4, find(PI_4)) = union(PI_3, PI_4, 4)$.
6. The algorithm finishes after $H_{1..3}$ are found. If we check the next number $d_i = 5$, $nextPI(5) = PI_3 = 5$, and $find(5) = |PI| = 4$. $d_i = 5$ cannot be the value of any h_k.

Finally, the output is $\langle 0, 1, 2, 3, 6, 9, \infty_1\rangle$, where the elements $\langle 0, 1, 2, 3\rangle$ come from $S_{0,*}$.

For the general union-find problem, the time required for each operation of union or find is $O(\beta)$, where the lower bound of β was proved to be functional inverse of Ackermann's function [26]. The union-find structure we use is a single path tree, and we only unite two consecutive sets. With the definition of static tree set union, the time complexity of each operation is reduced to O(1) [11]. Our algorithm needs O(n) operations of find and union, so the time complexity is O(n). The alignment result can be computed in linear time, when the linear-space S-table is given. In addition to get PO with increasing order, we can collect the elements h_k, and apply the bucket sort on these elements with an array of size n. It needs O(n) time. In summary, the concatenated LCS problem with the linear-space S-table can be solved in linear time.

4 Conclusion

In the previous studies of the S-table, the whole S-table of quadratic space is needed for further applications. Due to the growth of data size, to reduce the required space is an important issue. This paper considers the linear-space S-table, which consists of the first row of the S-table and the changes between every two consecutive rows. New algorithms are proposed to solve the concatenated LCS problem in O($n \log n$) and O(n) time with given the linear-space S-table, instead of the whole S-table reconstruction.

References

[1] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric applications of a matrix searching algorithm," *Proceedings of the Second Annual Symposium on Computational Geometry*, New York, USA, pp. 285–292, ACM, 1986.

[2] L. Allison and T. I. Dix, "A bit-string longest-common-subsequence algorithm," *Information Processing Letters*, Vol. 23, No. 5, pp. 305–310, 1986.

[3] C. E. R. Alves, E. N. Cáceres, and S. W. Song, "An all-substrings common subsequence algorithm," *Electronic Notes in Discrete Mathematics*, Vol. 19, pp. 133–139, 2005.

[4] H.-Y. Ann, C.-B. Yang, and C.-T. Tseng, "Efficient polynomial-time algorithms for the constrained lcs problem with strings exclusion," *Journal of Combinatorial Optimization*, Vol. 28, No. 4, pp. 800–813, Nov. 2014.

[5] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "Fast algorithms for computing the constrained lcs of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, May 2012.

[6] A. Apostolico, "String editing and longest common subsequences," *Handbook of Formal Languages*, pp. 361–398, Springer, 1997.

[7] J. L. Bentley, "Algorithms for klee's rectangle problems." Technical Report, Computer Science Department, Carnegie Mellon University, 1977.

[8] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," *Proceedings of Seventh International Symposium on String Processing and Information Retrieval*, A Coruña, Spain, pp. 39–48, IEEE, 2000.

[9] F. Y. Chin, A. De Santis, A. L. Ferrara, N. Ho, and S. Kim, "A simple algorithm for the constrained sequence problems," *Information Processing Letters*, Vol. 90, No. 4, pp. 175–179, 2004.

[10] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid, "A fast and practical bit-vector algorithm for the longest common subsequence problem," *Information Processing Letters*, Vol. 80, No. 6, pp. 279–285, 2001.

[11] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, New York, USA, pp. 246–251, ACM, 1983.

[12] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, No. 6, pp. 341–343, 1975.

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

[14] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng, "Algorithms for the merged-LCS problem and its variant with block constraint," *Proc. of the 23rd Workshop on Combinatorial Mathematics and Computation Theory*, Chang-Hua, Taiwan, pp. 232–239, 2006.

[15] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[16] G. M. Landau, E. Myers, and M. Ziv-Ukelson, "Two algorithms for LCS consecutive suffix alignment," *Annual Symposium on Combinatorial Pattern Matching*, Istanbul, Turkey, pp. 173–193, Springer, 2004.

[17] G. M. Landau, E. W. Myers, and J. P. Schmidt, "Incremental string comparison,"

SIAM Journal on Computing, Vol. 27, No. 2, pp. 557–582, 1998.

[18] G. M. Landau, B. Schieber, and M. Ziv-Ukelson, “Sparse LCS common substring alignment,” *Annual Symposium on Combinatorial Pattern Matching*, Michoacn, Mexico, pp. 225–236, Springer, 2003.

[19] G. M. Landau and M. Ziv-Ukelson, “On the common substring alignment problem,” *Journal of Algorithms*, Vol. 41, No. 2, pp. 338–359, 2001.

[20] M. Lu and H. Lin, “Parallel algorithms for the longest common subsequence problem,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, pp. 835–848, 1994.

[21] M. Maes, “On a cyclic string-to-string correction problem,” *Information Processing Letters*, Vol. 35, No. 2, pp. 73–78, 1990.

[22] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, Vol. 48, No. 3, pp. 443–453, 1970.

[23] 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, 2010.

[24] Y.-H. Peng, C.-B. Yang, K.-T. Tseng, and K.-S. Huang, “An algorithm and applications to sequence alignment with weighted constraints,” *International Journal of Foundations of Computer Science*, Vol. 21, No. 1, pp. 51–59, Feb. 2010.

[25] J. P. Schmidt, “All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings,” *SIAM Journal on Computing*, Vol. 27, No. 4, pp. 972–992, 1998.

[26] R. E. Tarjan, “Efficiency of a good but not linear set union algorithm,” *Journal of the ACM*, Vol. 22, No. 2, pp. 215–225, 1975.

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

[28] C.-T. Tseng, C.-B. Yang, and H.-Y. Ann, “Efficient algorithms for the longest common subsequence problem with sequential substring constraints,” *Journal of Complexity*, Vol. 29, No. 1, pp. 44–52, Feb. 2013.

[29] K.-T. Tseng, D.-S. Chan, and C.-B. Yang, “An efficient merged longest common subsequence algorithm for similar sequences,” *Proceedings of the 20th World Multi-Conference on Systemics, Cybernetics and Informatics*, Vol. I, Orlando, Florida, USA, pp. 93–98, July 2016.

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