Linear-Space LCS Enumeration for Two Strings

Yoshifumi Sakai
Graduate School of Agricultural Science, Tohoku University, Sendai, Japan

Abstract

Suppose we want to seek the longest common subsequences (LCSs) of two strings as informative patterns that explain the relationship between the strings. The dynamic programming algorithm gives us a table from which all LCSs can be extracted by traceback. However, the need for quadratic space to hold this table can be an obstacle when dealing with long strings. A question that naturally arises in this situation would be whether it is possible to exhaustively search for all LCSs one by one in a time-efficient manner using only a space linear in the LCS length, where we treat read-only memory for storing the strings as excluded from the space consumed. As a part of the answer to this question, we propose an $O(L)$-space algorithm that outputs all distinct LCSs of the strings one by one each in $O(n^2 \log L)$ time, where the strings are both of length n and L is the LCS length of the strings.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases algorithms, longest common subsequence, enumeration

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.2

Funding *Yoshifumi Sakai*: This work was supported by JSPS KAKENHI Grant Number JP23K10975.

1 Introduction

Comparing two strings to find common patterns that would be most informative of their relationships is a fundamental task in parsing them. The longest common subsequences (LCSs) are regarded as such common patterns, and the problem of efficiently finding one of them has long been studied [2, 3, 5, 6, 7, 8, 9, 10, 11, 13]. Here, a subsequence of a string is the sequence obtained from the string by deleting any number of elements at any position that is not necessarily contiguous. Hence, an LCS of two strings is a common subsequence obtained by deleting the least possible number of elements from the strings. For example, if $X =$ `acddadacbcb` and $Y =$ `caccbaadcad` are the target strings, then `caccb` is one of the seven distinct LCSs of X and Y. The six other LCSs are `cacbc`, `accbc`, `acaac`, `acadc`, `acada`, and `acdad`.

Given a pair of strings X and Y both of length n and a string Z of length less than n, it is easy to determine whether Z is a common subsequence of X and Y in $O(n)$ time. In comparison, it is hard to determine whether Z is an LCS of X and Y. The reason is that we need to know the LCS length in advance. It was revealed [1, 4] that unless the strong exponential time hypothesis (SETH) does not hold, no algorithm can determine the LCS length of X and Y in $O(n^{2-\varepsilon})$ time for any positive constant ε. Therefore, we cannot find any LCS in $O(n^{2-\varepsilon})$ time under SETH.

On the other hand, as is well known, finding an arbitrary LCS of X and Y is possible in $O(n^2)$ time and $O(n^2)$ space by the dynamic programming (DP) algorithm [13]. The space of size $O(n^2)$ consumed to store the LCS lengths for all pairs of a prefix of X and a prefix of Y can be treated as constituting a particular directed acyclic graph (DAG) such that any path from the source to the sink represents an LCS of X and Y. This DAG is also useful in applications other than seeking a single arbitrary LCS because each LCS is represented by at least one of the paths from the source to the sink. If one considers only the problem of finding a single arbitrary LCS without intending a data structure representing all LCSs, Hirschberg [7] showed that $O(n)$ space is sufficient to solve the problem in the same $O(n^2)$

time as the DP algorithm. Excluding the read-only memory storing X and Y, his algorithm with a slight modification performs only in $O(L(X,Y))$ space, where $L(X,Y)$ denotes the LCS length of X and Y.

The advantage of $O(L(X,Y))$-space algorithms for finding an arbitrary LCS is that they perform without reserving $O(n^2)$ space, the size of which becomes pronounced when n is large. Thus, a natural question that would come to mind is whether it is possible to design $O(L(X,Y))$-space algorithms that are as accessible to all LCSs as the DP algorithm in addition to the above advantage. As a part of the answer to this question, this article shows that an $O(L(X,Y))$-space algorithm is capable of enumerating all distinct LCSs with a delay time not significantly inferior to the running time of Hirschberg's $O(L(X,Y))$-space algorithm [7] for finding a single arbitrary LCS. More precisely, the algorithm proposed in this article outputs each LCS in $O(n^2 \log L(X,Y))$ time after the previous LCS is output, if any, which is inferior by a logarithmic factor of $L(X,Y)$. Enumeration is done by introducing a DAG, called the all-LCS graph, with each path from the source to the sink corresponding to a distinct LCS and vice versa. Since the size of the all-LCS graph is $O(n^2)$, which exceeds $O(L(X,Y))$, we design the proposed algorithm to somehow perform a depth-first search without explicitly constructing it.

2 Preliminaries

For any sequences S and S', let $S \circ S'$ denote the concatenation of S followed by S'. For any sequence S, let $|S|$ denote the length of S, i.e., the number of elements in S. For any index i with $1 \leq i \leq |S|$, let $S[i]$ denote the ith element of S, so that $S = S[1] \circ S[2] \circ \cdots \circ S[|S|]$. A *subsequence* of S is obtained from S by deleting zero or more elements at any positions not necessarily contiguous, i.e., the sequence $S[i_1] \circ S[i_2] \circ \cdots \circ S[i_\ell]$ for some length ℓ with $0 \leq \ell \leq |S|$ and some ℓ indices $i_1, i_2, \ldots, i_\ell$ with $1 \leq i_1 < i_2 < \cdots < i_\ell \leq |S|$. For any indices i and i' with $1 \leq i \leq i' + 1 \leq |S| + 1$, let $S[i:i']$ denote the contiguous subsequence $S[i] \circ S[i+1] \circ \cdots \circ S[i']$ of S, where $S[i:i']$ with $i = i' + 1$ represents the empty subsequence. If $i = 1$ (resp. $i' = |S|$), then $S[i:i']$ is called a *prefix* (resp. *suffix*) of S.

A string is a sequence of characters over an alphabet set. For any strings X and Y, a *common subsequence* of X and Y is a subsequence of X that is a subsequence of Y. Let $L(X,Y)$ denote the maximum of $|Z|$ over all common subsequences Z of X and Y. Any common subsequence Z of X and Y with $|Z| = L(X,Y)$ is called a *longest common subsequence* (an *LCS*) of X and Y. The following lemma gives a typical recursive expression for the LCS length.

▶ **Lemma 1** (e.g., [13]). *For any strings X and Y, if at least one of X and Y is empty, then $L(X,Y) = 0$; otherwise, if $x = y$, then $L(X,Y) = L(X',Y') + 1$; otherwise, $L(X,Y) = \max(L(X',Y), L(X,Y'))$, where $X = X' \circ x$ and $Y = Y' \circ y$. The same also holds for the case where $X = x \circ X'$ and $Y = y \circ Y'$.*

Let any algorithm that takes certain information, specified later, of an arbitrary pair of non-empty strings X and Y as input and uses only $O(L(X,Y))$ space to output all distinct LCSs of X and Y, each represented by a specific form, one by one be called a *linear-space LCS enumeration algorithm*. We assume that the information of X and Y given to the algorithm consists of $|X|$, $|Y|$, and $O(1)$-time access to check whether $X[i] = Y[j]$ or not for any pair of indices i and j with $1 \leq i \leq |X|$ and $1 \leq j \leq |Y|$ with no space consumption. The worst-case time between outputting one LCS and the next LCS is called the *delay time* of the algorithm. This article aims to propose a linear-space LCS enumeration algorithm with $O(|X||Y| \log L(X,Y))$ delay time.

Let any index pair (i, j) such that $1 \leq i \leq |X|$, $1 \leq j \leq |Y|$, and $X[i] = Y[j]$ be called a *match*. For convenience, we sometimes treat $X[0]$ and $Y[0]$ (resp. $X[|X| + 1]$ and $Y[|Y| + 1]$) as virtual elements of X and Y, respectively, both of which are identical to a virtual character, say ♯ (resp. \$), that appears in neither X nor Y, so that $(0, 0)$ (resp. $(|X| + 1, |Y| + 1)$) can be treated as a virtual match. We use SRC and SNK to denote $(0, 0)$ and $(|X| + 1, |Y| + 1)$, respectively. For any match w, let i_w and j_w denote the indices such that $(i_w, j_w) = w$. Furthermore, let the *diagonal coordinate* of w be $j_w - i_w$, by treating any match as a grid point on a two-dimensional plane. For any matches w and w', let $w < w'$ (resp. $w \leq w'$) mean that both $i_w < i_{w'}$ and $j_w < j_{w'}$ (resp. $i_w \leq i_{w'}$ and $j_w \leq j_{w'}$). For any non-virtual match w, we call the character that is identical to both $X[i_w]$ and $Y[j_w]$ the *character corresponding to* w. Similarly, for any sequence W of non-virtual matches, we call the concatenation of the characters corresponding to $W[k]$ for all indices k from 1 to $|W|$ in this order the *string corresponding to* W.

Let us call any sequence W of $L(X, Y)$ non-virtual matches with $W[1] < W[2] < \cdots < W[L(X, Y)]$ an *LCS-match sequence*, because the string corresponding to W is an LCS of X and Y. For convenience, we sometimes treat SRC $(= (0, 0))$ and SNK $(= (|X| + 1, |Y| + 1))$ as virtual elements $W[0]$ and $W[L(X, Y) + 1]$ of any LCS-match sequence W, respectively. Although any LCS Z of X and Y has at least one LCS-match sequence W with Z as the corresponding string, two or more such LCS-match sequences W may exist. Hence, the problem of enumerating LCSs is not equivalent to the problem of enumerating LCS-match sequences. As the representative of LCS-match sequences W having the same corresponding strings, we consider the one such that for any index k with $1 \leq k \leq L(X, Y)$, $X[1 : i_{W[k]}]$ and $Y[1 : j_{W[k]}]$ are the shortest prefixes of X and Y with the string corresponding to $W[1 : k]$ as a subsequence, respectively. We call any such LCS-match sequence *front-leaning*. Since any LCS Z of X and Y has exactly one front-leaning LCS-match sequence W with Z as the corresponding string, the problem of enumerating LCSs is equivalent to the problem of enumerating front-leaning LCS-match sequences. Based on this observarion, we design the proposed linear-space LCS enumeration algorithm to output all distinct front-leaning LCS-match sequences one by one instead of outputting the corresponding LCSs.

3 Algorithm

This section proposes a linear-space LCS enumeration algorithm that outputs all distinct front-leaning LCS-match sequences of X and Y each in $O(|X||Y| \log L(X, Y))$ time.

Our approach to designing the algorithm considers a particular directed acyclic graph (DAG), which we call the *all-LCS graph*, such that each path from vertex SRC $(= (0, 0))$ to vertex SNK $(= (|X| + 1, |Y| + 1))$ represents a distinct front-leaning LCS-match sequence and vice versa, where each vertex in the graph is a different particular match. Relying on this DAG, we can enumerate LCSs by enumerating all paths from SRC to SNK in a straightforward manner using depth-first search (DFS). The all-LCS graph we introduce is of size $O(|X||Y|)$, which exceeds the size of space allowed for any linear-space LCS enumeration algorithm. Our proposed algorithm overcomes this situation by simulating DFS on the all-LCS graph without explicitly constructing it. We define the all-LCS graph in Section 3.1 and propose a linear-space LCS enumeration algorithm by presenting how to simulate the DFS in Section 3.2.

3.1 All-LCS graph

Conceptually speaking, the all-LCS graph is the union of the path from SRC to SNK that passes through vertices $W[0], W[1], \ldots, W[L(X,Y)+1]$ in this order over all front-leaning LCS-match sequences W, where we recall that $W[0] = \text{SRC}$ and $W[L(X,Y)+1] = \text{SNK}$. The correctness of the definition can be verified because for any front-leaning LCS-match sequences that share an element, exchanging their prefixes ending at the element also yields front-leaning LCS-match sequences. However, since we need to use the all-LCS graph without knowing what front-leaning LCS-match sequences exist, the current definition is not specific enough for our situation. To address this issue, we redefine the all-LCS graph as inductively defined.

For simplicity, for any match w, let $L^-(w)$ and $L^+(w)$ denote $L(X[1:i_w], Y[1:j_w])$ and $L(X[i_w:|X|], Y[j_w:|Y|])$, respectively. The key idea to redefine the all-LCS graph is to focus only on particular matches introduced below.

▶ **Definition 2** (valid match). *Let any match w such that $L^-(w) + L^+(w) = L(X,Y)+1$ be called $L^-(w)$-valid, or simply* valid.

▶ **Definition 3** (follower). *For any valid match u, let any $(L^-(u)+1)$-valid match v that is the only match w with $u < w \leq v$ be called a u-*follower. *Let F_u denote the sequence of all u-followers such that $i_{F_u[1]} > i_{F_u[2]} > \cdots > i_{F_u[|F_u|]}$ (and hence $j_{F_u[1]} < j_{F_u[2]} < \cdots < j_{F_u[|F_u|]}$). Adopting the order of elements in F_u, let the f*th *u-*follower *be $F_u[f]$ and let the* next *u-*follower of $F_u[f]$ *be $F_u[f+1]$, unless $f = |F_u|$.*

Definition 2 is conceived from the fact that for any LCS-match sequence W and any index k with $0 \leq k \leq L(X,Y)+1$, both $L^-(W[k]) = k$ and $L^+(W[k]) = L(X,Y)+1-k$. On the other hand, Definition 3 comes from the observation that for any front-leaning LCS-match sequence W and any index k with $0 \leq k \leq L(X,Y)$, $W[k]$ is k-valid, $W[k+1]$ is $(k+1)$-valid, and $W[k+1]$ is the only match w such that $W[k] < w \leq W[k+1]$. Sequence F_u introduced in Definition 3 makes sense because for any distinct u-followers v and v', if at least $i_v \geq i_{v'}$ or $j_v \leq j_{v'}$, then both $i_v > i_{v'}$ and $j_v < j_{v'}$; otherwise, both $i_v < i_{v'}$ and $j_v > j_{v'}$, due to condition that v (resp. v') is the only match w with $u < w \leq v$ (resp. $u < w \leq v'$). Our inductive definition of the all-LCS graph is as follows (see Figure 1).

▶ **Definition 4** (all-LCS graph). *The* all-LCS graph *is defined inductively to be the DAG such that* SRC *$(= (0,0))$ is a vertex and for any vertex u, all u-followers are vertices and any u-follower v is connected with u by an edge from u to v.*

The following lemma claims that Definition 4 defines the all-LCS graph well.

▶ **Lemma 5.** *The all-LCS graph has a path from vertex* SRC *to vertex* SNK *that passes through vertices* SRC$, w_1, w_2, \ldots, w_\ell,$ SNK *in this order if and only if $w_1 \circ w_2 \circ \cdots \circ w_\ell$ is a front-leaning LCS-match sequence.*

Proof. Let W be an arbitrary front-leaning LCS-match sequence, so that for any index k with $1 \leq k \leq |W|+1$, $W[k]$ is a $W[k-1]$-follower. Since SRC is a vertex of the all-LCS graph, induction proves that for any index k with $1 \leq k \leq |W|+1$, the all-LCS graph has an edge from $W[k-1]$ to $W[k]$, completing the "if" part of the lemma.

Consider the path of the all-LCS graph in the lemma. Since SRC is 0-valid, $W[k]$ is a $W[k-1]$ follower for any index k with $1 \leq k \leq \ell+1$, and SNK is $(L(X,Y)+1)$-valid, we have that $\ell = L(X,Y)$. Induction proves that for any index k with $1 \leq k \leq \ell$, $X[1:i_{w_k}]$ and $Y[1:j_{w_k}]$ are the shortest prefixes of X and Y with the string corresponding to $w_1 \circ w_2 \circ \cdots \circ w_k$ as a subsequence, respectively, completing the proof of the "only if" part. ◀

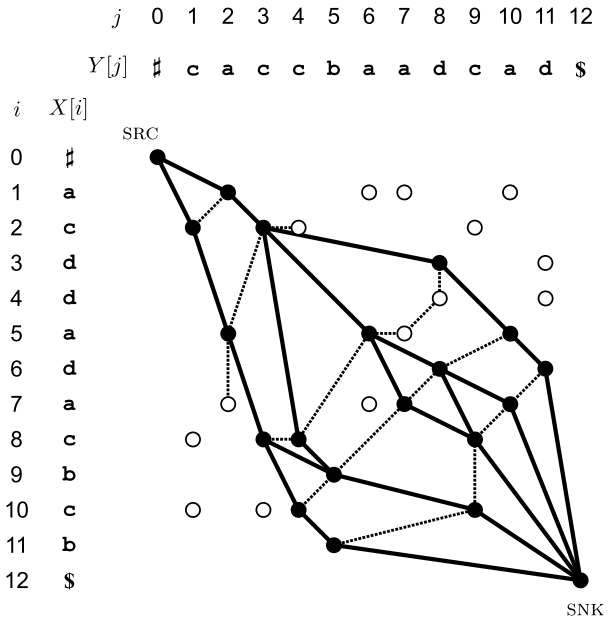

Figure 1 The all-LCS graph for $X = \texttt{acddadacbcb}$ and $Y = \texttt{caccbaadcad}$, having seven LCSs, `caccb`, `cacbc`, `accbc`, `acaac`, `acadc`, `acada`, and `acdad`, where vertices and edges of the graph are indicated by solid bullets and lines, respectively, matches other than the vertices are indicated by open bullets, and all k-valid matches for each index k other than 0 and $L(X,Y)+1$ $(= 6)$ are indicated by a dotted polygonal line connecting them from the lowermost-leftmost one to the uppermost-rightmost one in ascending order of the diagonal coordinate.

The size of the all-LCS graph is $O(|X||Y|)$ as stated in the following lemma.

▶ **Lemma 6.** *The number of edges in the all-LCS graph is at most* $2|X||Y|+1$.

Proof. The number of edges in the all-LCS graph is equal to the sum of $|F_u|$ over all vertices u in the graph, where F_u is the sequence of all u-followers introduced in Definition 3. For any vertex u and any index f with $1 \leq f \leq \lfloor |F_u|/2 \rfloor$, it follows from $i_{F_u[1]} > i_{F_u[2]} > \cdots > i_{F_u[|F_u|]} > i_u$ (resp. $j_u < j_{F_u[1]} < j_{F_u[2]} < \cdots < j_{F_u[|F_u|]}$) that $i_u + f < i_u + ((|F_u|+1)-f) \leq i_{F_u[f]}$ (resp. $j_u + f \leq j_{F_u[f]}$). On the other hand, since $F_u[f]$ is a u-follower, $F_u[f]$ is the only match w with $u < w \leq F_u[f]$. Thus, (i_u+f, j_u+f) is not a match. This implies that each (i_u+f, j_u+f) is a distinct non-match index pair with $1 \leq i_u+f \leq |X|$ and $1 \leq j_u+f \leq |Y|$ because (i_u, j_u) is a match and $(i_u+1, j_u+1), (i_u+1, j_u+2), \ldots, (i_u+f-1, j_u+f-1)$ include no match. In addition, each vertex u having an odd number of u-followers other than SRC is a distinct match with $1 \leq i_u \leq |X|$ and $1 \leq j_u \leq |Y|$. Therefore, the lemma holds because there exist $|X||Y|$ index pairs (i,j) with $1 \leq i \leq |X|$ and $1 \leq j \leq |Y|$. ◀

▶ Remark 7. If an $O(|X||Y|)$ space is available, then as an implementation of the all-LCS graph, it is not difficult to design an $O(|X||Y|)$-time algorithm that constructs the two-dimensional array G such that each element $G[i,j]$ with $0 \leq i \leq |X|+1$ and $0 \leq j \leq |Y|+1$ is the sequence of all (i,j)-followers in ascending order of the diagonal coordinate, if (i,j) is a vertex of the all-LCS graph, or the empty sequence, otherwise. Once G is available, we can enumerate LCSs with $O(L(X,Y))$ delay time by finding each distinct path in the all-LCS graph from SRC to SNK in $O(L(X,Y))$ time by performing DFS.

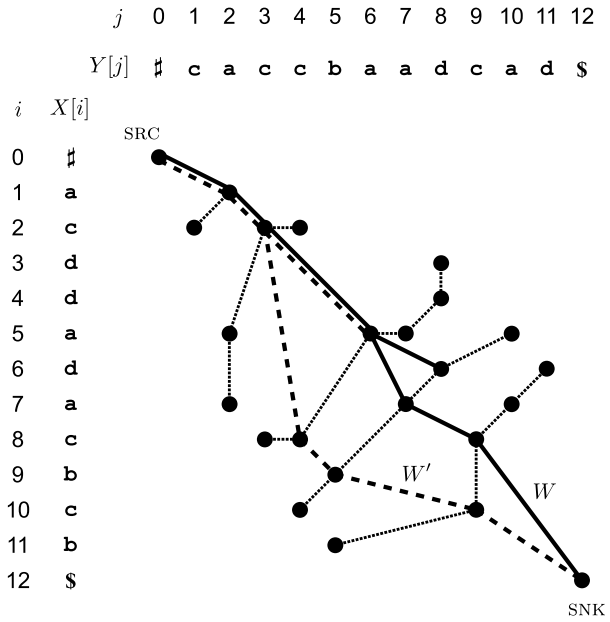

Figure 2 Valid matches with the third (resp. fourth) output front-leaning LCS-match sequences W' (resp. W), indicated by dashed (resp. solid) polygonal line connecting from $\text{SRC} = (0, 0)$ to $\text{SNK} = (12, 12)$, for the same X and Y as Figure 1, where the branching match of W' (resp. W) is indicated by a dashed (resp. solid) line connecting from $W'[\kappa - 1]$ (resp. $W[\kappa - 1]$) for the branching position κ of W' (resp. W) to it.

3.2 Proposed linear-space LCS enumeration algorithm

As mentioned earlier, the linear-space LCS enumeration algorithm we propose simulates DFS on the all-LCS graph without explicitly constructing it, because the algorithm can use only $O(L(X, Y))$ space while the size of the all-LCS graph is $O(|X||Y|)$.

3.2.1 Algorithm overview

The proposed algorithm outputs each distinct front-leaning LCS-match sequence W in lexicographical order of the sequence $f_1 \circ f_2 \circ \cdots \circ f_{L(X,Y)}$, where $W[k]$ is the f_kth $W[k-1]$-follower (see Definition 3). Due to this output order setting, the following index and match immediately provide the subsequent lemma, based on which we design the proposed algorithm.

▶ **Definition 8** (branching position and match). *For any front-leaning LCS-match W, let the greatest index k with $1 \leq k \leq L(X, Y)$ such that $W[k]$ is not the last $W[k-1]$-follower be called the* branching position of W, *if any. In addition, let the next $W[\kappa]$-follower of $W[\kappa]$ be called the* branching match of W, *if the branching position κ of W exists.*

▶ **Lemma 9.** *For any front-leaning LCS-match sequence W, W is the last output one if and only if W has no branching position. If W is the first output one, then for any index k with $1 \leq k \leq L(X, Y)$, $W[k]$ is the first $W[k-1]$-follower. Otherwise, $W[1 : \kappa - 1] = W'[1 : \kappa - 1]$, $W[\kappa]$ is the branching match of W', and for any index k with $\kappa + 1 \leq k \leq L(X, Y)$, $W[k]$ is the first $W[k-1]$-follower, where W' is the last output front-leaning LCS-match sequence before W and κ is the branching position of W'.*

Algorithm 1 enumLCS.

```
1 compute L(X,Y) in O(|X||Y|) time and O(L(X,Y)) space
2 if L(X,Y) ≥ 1 then
3     W ← a match sequence of length L(X,Y); κ ← 0
4     repeat
5         updateW
6         output W as a distinct front-leaning LCS-match sequence
7         findBranch
8     until κ is a dummy index
```

▶ **Example 10.** Consider the third and fourth output front-leaning LCS-match sequences as W' and W, respectively, for the same X and Y as Figure 1 (see Figure 2). Since W' is $(1,2)\circ(2,3)\circ(8,4)\circ(9,5)\circ(10,9)$ with $F_{W'[0]} = F_{(0,0)} = (2,1)\circ(1,2)$, $F_{W'[1]} = F_{(1,2)} = (2,3)$, $F_{W'[2]} = F_{(2,3)} = (8,4)\circ(5,6)\circ(3,8)$, $F_{W'[3]} = F_{(8,4)} = (9,5)$, $F_{W'[4]} = F_{(9,5)} = (10,9)$, and $F_{W'[5]} = F_{(10,9)} = (12,12)$, the branching position of W' is 3 and the branching match of W' is $(5,6)$. Thus, it follows from Lemma 9 that $W[1:2] = W'[1:2] = (1,2)\circ(2,3)$ and $W[3] = (5,6)$. In addition, $W[4] = (7,7)$ due to $F_{W[3]} = F_{(5,6)} = (7,7)\circ(6,8)$, and $W[5] = (8,9)$ due to $F_{W[4]} = F_{(7,7)} = (8,9)$. Hence, W is determined as $(1,2)\circ(2,3)\circ(5,6)\circ(7,7)\circ(8,9)$.

The proposed algorithm denoted enumLCS obtains each distinct front-leaning LCS-match sequence W using variables W and κ. These variables are to maintain W and the existing branching position of W, respectively. After initializing κ to 0, for each front-leaning LCS-match sequence W according to our output order setting, the algorithm executes procedure updateW to set $\mathsf{W}[\kappa+1:L(X,Y)]$ to $W[\kappa+1:L(X,Y)]$, outputs W as W, and executes procedure findBranch to set κ and $W[\kappa]$ to the branching position and match of W, respectively, if existing. If the branching position of W does not exist, then κ is set to a dummy index, indicating that W is the last output one. Due to Lemma 9, induction proves that the algorithm outputs all distinct front-leaning LCS-match sequences of X and Y one after another. A pseudo-code of enumLCS is given as Algorithm 1.

3.2.2 Space-efficient search for locally valid matches

Before presenting the details of procedures updateW and findBranch, we generalize the validity of matches to the local case and develop an algorithm that generates the stream of all the "locally" k-valid matches in ascending order of the diagonal coordinate. The reason is that both updateW and findBranch are designed to simulate this algorithm.

For any valid matches u and v with $u < v$ and any match w with $u < w < v$, let $L_u^-(w)$ denote $L(X[i_u+1:i_w], Y[j_u+1:j_w])$ and let $L_v^+(w)$ denote $L(X[i_w:i_v-1], Y[j_w:j_v-1])$, so that $L^-(w) = L_{(0,0)}^-(w)$ and $L^+(w) = L_{(|X|+1,|Y|+1)}^+(w)$. We define locally valid matches as follows.

▶ **Definition 11** (locally valid match). *For any valid matches u and v with $u < v$, let any match with $u < w < v$ and $L_u^-(w) + L_v^+(w) = L^-(v) - L^-(u)$ be called* (u,v)-locally $(L^-(u) + L_u^-(w))$-valid*, or simply* (u,v)-locally valid.

Note that for any valid matches u, v, and w with $u < w < v$, w is not necessarily (u,v)-locally valid. For example, $u = (5,6)$, $v = (12,12)$, and $w = (6,11)$ for the same X and Y as Figure 2 are valid matches with $u < w < v$ but w is not (u,v)-locally valid. In contrast, any

Algorithm 2 genStream(u, v, l, m, k).

```
1  j ← j_u; i ← i_v; I, J ← the empty sequences; incJ; decI
2  while j ≤ j_v − 1 and i ≥ i_u + 1 do
3      if |I| ≥ k − l, |J| ≥ m − k, i ≥ I[k − l], j ≤ J[m − k], and X[i] = Y[j] then
4          output (i, j)
5      if |I| < k − l or i ≤ I[k − l] then
6          incJ
7      else
8          decI
9  procedure incJ
10     j ← j + 1; p ← |I|
11     foreach i from i_v − 1 down to i_u + 1 do
12         if p > 0 and I[p] = i then p ← p − 1
13         if X[i] = Y[j] then I[p + 1] ← i
14 procedure decI
15     i ← i − 1; p ← |J|
16     foreach j from j_u + 1 to j_v − 1 do
17         if p > 0 and J[p] = j then p ← p − 1
18         if X[i] = Y[j] then J[p + 1] ← j
```

(u, v)-locally k-valid match w is k-valid, because $L^-(w) = k$ and $L^+(w) = L(X, Y) + 1 - k$ due to $L^-(w) \geq L^-(u) + L_u^-(w) = k$, $L^+(w) \geq L_v^+(w) + L^+(v) = L(X, Y) + 1 - k$, and $L^-(w) + L^+(w) \leq L(X, Y) + 1$.

The number of (u, v)-locally k-valid matches is $O((i_v - i_u) + (j_v - j_u))$, which may exceed $O(L^-(v) - L^-(u))$. However, the stream of all (u, v)-locally k-valid matches in ascending order of the diagonal coordinate can be generated by executing the $O(L^-(v) - L^-(u))$-space algorithm genStream presented as Algorithm 2.

▶ **Lemma 12.** *For any indices l, k, and m with $l < k < m$ and any pair of an l-valid match u and an m-valid match v with $u < v$, genStream(u, v, l, m, k) outputs all (u, v)-locally k-valid matches one by one in ascending order of the diagonal coordinate in $O((i_v - i_u)(j_v - j_u))$ time and $O(m - l)$ space.*

Proof. For any index i with $i_u \leq j \leq i_v$ and any index j with $j_u \leq j \leq j_v$, let $L_u^-(i, j)$ and $L_v^+(i, j)$ denote $L(X[i_u + 1 : i], Y[j_u + 1 : j])$ and $L(X[i : i_v - 1], Y[j : j_v - 1])$, respectively, so that for any match w with $u < w < v$, $L_u^-(w) = L_u^-(i_w, j_w)$ and $L_v^+(w) = L_v^-(i_w, j_w)$.

To find all (u, v)-locally k-valid matches, genStream maintains two index variables j and i with $j_u + 1 \leq \mathsf{j} \leq j_v$ and $i_u \leq \mathsf{i} \leq i_v - 1$. After initializing j and i to $j_u + 1$ and $i_v - 1$, respectively, the algorithm repeatedly updates j and i by either increasing j by one or decreasing i by one and outputs (i, j) as a distinct (u, v)-locally k-valid match, if $L_u^-(\mathsf{i}, \mathsf{j}) = k - l$, $L_v^+(\mathsf{i}, \mathsf{j}) = m - k$, and $X[\mathsf{i}] = Y[\mathsf{j}]$, until $\mathsf{j} = j_v$ or $\mathsf{i} = i_u$. Hence, the (u, v)-locally k-valid matches output by the algorithm are in ascending order of the diagonal coordinate (see Table 1).

At each iteration of the algorithm, at least $L_u^-(\mathsf{i}-1, \mathsf{j}) < k-l$ or $L_v^+(\mathsf{i}, \mathsf{j}+1) < m-k$. This is because otherwise there exist matches w^- and w^+ with $u < w^- < w^+ < v$, $L_u^-(w^-) \geq k - l$, and $L_v^+(w^+) \geq m - k$, which implies that $L(X, Y) \geq L^-(w^-) + L^+(w^+) \geq (L^-(u) +$

Table 1 Table of j, I, i, J, and the diagonal coordinate $\mathsf{j}-\mathsf{i}$ of (i,j) just before the rth execution of line 2 of GENSTREAM(u,v,l,m,k) implemented as Algorithm 2, excluding the last execution to terminate the algorithm, for the same X and Y as Figure 2, $u=(5,6)$ with $l=3$, $v=(12,12)$ with $m=6$, and $k=5$, where the last column indicates the (u,v)-locally k-valid matches (i,j) output just after the rth execution of line 3.

r	j	I	i	J	$\mathsf{j}-\mathsf{i}$	output matches
1	7	7	11	empty	-4	
2	8	6	11	empty	-3	
3	9	$6\circ 8$	11	empty	-2	
4	9	$6\circ 8$	10	9	-1	$(10,9)$
5	9	$6\circ 8$	9	9	0	
6	9	$6\circ 8$	8	9	1	$(8,9)$
7	10	$6\circ 7$	8	9	2	
8	10	$6\circ 7$	7	$10\circ 7$	3	$(7,10)$
9	11	$6\circ 7$	7	$10\circ 7$	4	

$L_u^-(w^-))+(L_v^+(w^+)+L^+(v))\geq(l+(k-l))+((m-k)+(L(X,Y)+1-m))=L(X,Y)+1$, a contradiction. Therefore, if $L_u^-(\mathsf{i}-1,\mathsf{j})<k-l$, then any match (i,j) with $i_u+1\leq i\leq \mathsf{i}-1$ is not (u,v)-locally k-valid due to $L_u^-(i,\mathsf{j})\leq L_u^-(\mathsf{i}-1,\mathsf{j})$; otherwise, any match (i,j) with $\mathsf{j}+1\leq j\leq j_v-1$ is not (u,v)-locally k-valid due to $L_v^+(\mathsf{i},j)\leq L_v^+(\mathsf{i},\mathsf{j}+1)$. Based on this fact, in each iteration of the algorithm, if $L_u^-(\mathsf{i}-1,\mathsf{j})<k-l$, then j is increased by one; otherwise, i is decreased by one.

The algorithm accomplishes the above by maintaining a sequence variable I of at most $m-l$ indices so that $|\mathsf{I}|=L_u^-(i_v-1,\mathsf{j})$ and for any index p with $1\leq p\leq|\mathsf{I}|$, $\mathsf{I}[p]$ is the least index i with $i_u+1\leq i\leq i_v-1$ such that $L_u^-(i,\mathsf{j})=p$. Analogously, a sequence variable J of at most $m-l$ indices is maintained so that $|\mathsf{J}|=L_v^+(\mathsf{i},j_u+1)$ and for any index p with $1\leq p\leq|\mathsf{J}|$, $\mathsf{J}[p]$ is the greatest index j with $j_u+1\leq j\leq j_v-1$ such that $L_v^+(\mathsf{i},j)=p$. The algorithm executes procedure INCJ (lines 9 through 13 of Algorithm 2) to increase j by one and update I in a straightforward way based on Lemma 1 in $O(i_v-i_u)$ time and executes procedure DECI (lines 14 through 18) to decrease i by one and update J in $O(j_v-j_u)$ time symmetrically.

It follows from $L_u^-(\mathsf{i},\mathsf{j})+L_v^+(\mathsf{i},\mathsf{j})\leq m-l$ that both $L_u^-(\mathsf{i},\mathsf{j})=k-l$ and $L_v^+(\mathsf{i},\mathsf{j})=m-k$ hold if and only if all of $|\mathsf{I}|\geq k-l$, $|\mathsf{J}|\geq m-k$, $\mathsf{i}\geq\mathsf{I}[k-l]$, and $\mathsf{j}\leq\mathsf{J}[m-k]$ hold. On the other hand, $L_u^-(\mathsf{i}-1,\mathsf{j})<k-l$ if and only if $|\mathsf{I}|<k-l$ or $\mathsf{i}\leq\mathsf{I}[k-l]$. Thus, all distinct (u,v)-locally k-valid matches are output by the algorithm successfully in ascending order of the diagonal coordinate. Since INCJ is executed at most j_v-j_u times and DECI is executed at most i_v-i_u times, the algorithm runs in $O((i_v-i_u)(j_v-j_u))$ time and $O(m-l)$ space. ◀

3.2.3 Ideas for speeding up the naive method

In what follows, for simplicity, we use κ to represent the index maintained by ENUMLCS, when W is output as W by line 6 of Algorithm 1, where W is an arbitrary front-leaning LCS-match sequence. Hence, if W is the first output one, then $\kappa=0$; otherwise, κ is the branching position of the last output front-leaning LCS-match sequence before W. When mentioning GENSTREAM(u,v,l,m,k), it is often the case that l and m are obvious from u and v, respectively. In such cases, for the sake of readability, l and m are omitted from the specification, and we denote it by GENSTREAM(u,v,k).

A naive $O(L(X,Y))$-space procedure completes the task of updateW and findBranch simultaneously in $O(L(X,Y)|X||Y|)$ time as follows. For each index k from 1 to $L(X,Y)$ in this order, by simulating genStream$(W[k-1], \text{SNK}, k)$ in $O(|X||Y|)$ time, we can obtain the existing next $W[k-1]$-follower of $W[k]$ as a candidate of the branching match of W, if $k \leq \kappa$, or otherwise, we can obtain both the first and existing second $W[k-1]$-followers of $W[k-1]$ as $W[k]$ and a candidate of the branching match, respectively. The existing last found candidate is the branching match of W, based on which the branching position κ of W can also be determined.

We reduce the execution time of the naive procedure to $O(|X||Y| \log L(X,Y))$ by considering an approximation $W^\star$ of W that can be obtained faster than W and transformed into W easily. The definition of $W^\star$ is as follows.

▶ **Definition 13** (approximation $W^\star$ of W)**.** *For any front-leaning LCS-match sequence W, let $W^\star$ denote the sequence such that $W^\star[1:\kappa] = W[1:\kappa]$ and for any index k with $\kappa+1 \leq k \leq L(X,Y)$, $W^\star[k]$ is the $(W[\kappa], \text{SNK})$-locally k-valid match with the least diagonal coordinate.*

▶ **Example 14.** If W is the forth output front-leaning LCS-match sequence for the same X and Y as Figure 2, which is $(1,2) \circ (2,3) \circ (5,6) \circ (7,7) \circ (8,9)$ with $\kappa = 3$ (see Example 10), then $W^\star$ is $(1,2) \circ (2,3) \circ (5,6) \circ (7,7) \circ (10,9)$. Note that $W^\star[5]$ is not identical to $W[5]$ because the $((5,6),(12,12))$-locally 5-valid match with the least diagonal coordinate is not $(8,9)$ but $(10,9)$ (see Table 1).

The difference between W and $W^\star$ that allows us to obtain $W^\star$ faster than W is that each element $W[k]$ of W with $\kappa+1 \leq k \leq L(X,Y)$ is defined dependent on $W[k-1]$ while the corresponding element $W^\star[k]$ of W^* is defined independent of $W^\star[k-1]$. That is, unlike the case of W, where each element $W[k]$ of $W[\kappa+1 : L(X,Y)]$ must be determined in ascending order of k from $\kappa+1$ to $L(X,Y)$, each element $W^\star[k]$ of $W^\star[\kappa+1 : L(X,Y)]$ can be determined in any order. Procedure updateW obtains $W^\star[\kappa+1 : L(X,Y)]$ by recursively decomposing the problem of determining $W^\star[l+1 : m-1]$ into the problems of determining $W^\star[l+1 : k-1]$ and determining $W^\star[k+1 : m-1]$, which is done by simulating genStream$(W^\star[l], W^\star[m], k)$ until the first match is output to determine $W^\star[k]$. The following lemma guarantees that our approach works by claiming that $W^\star[l] < W^\star[l+1] < \cdots < W^\star[m]$ because this implies that $W^\star[k]$ is $(W^\star[l], W^\star[m])$-locally k-valid.

▶ **Lemma 15.** *For any front-leaning LCS-match sequence W, $W^\star$ is an LCS-match sequence.*

Proof. Since $W^\star[1] < W^\star[2] < \cdots < W^\star[\kappa+1]$, it suffices to show that for any index k with $\kappa+2 \leq k \leq L(X,Y)$, $W^\star[k-1] < W^\star[k]$. Since $W^\star[k-1]$ is $(W^\star[\kappa], \text{SNK})$-locally $(k-1)$-valid, there exists a $(W^\star[\kappa], \text{SNK})$-locally k-valid match w with $W^\star[k-1] < w$. It follows from $j_{W^\star[k]} - i_{W^\star[k]} \leq j_w - i_w$ and neither $W^\star[k] < w$ nor $w < W^\star[k]$ that $i_w \leq i_{W^\star[k]}$. This implies that $i_{W^\star[k-1]} < i_{W^\star[k]}$ because $i_{W^\star[k-1]} < i_w$. Symmetrically, there exists a $(W^\star[\kappa], \text{SNK})$-locally $(k-1)$-valid match w with $w < W^\star[k]$, which satisfies that $j_{W^\star[k-1]} \leq j_w < j_{W^\star[k]}$. Thus, $W^\star[k-1] < W^\star[k]$. ◀

The conversion from $W^\star$ to W is possible in $O(|X| - i_{W[\kappa]})$ time by determining $W[k]$ using $W[k-1]$ and $W^\star[k]$ for each index k from $\kappa+1$ to $L(X,Y)$ in this order based on the following Lemma.

▶ **Lemma 16.** *For any front-leaning LCS-match sequence W and any index k with $\kappa+1 \leq k \leq L(X,Y)$, $W[k]$ is the match $(i, j_{W^\star[k]})$ with the least index i that is greater than $i_{W[k-1]}$.*

Table 2 Table of sequences J_k maintained by GENSTREAM$_k$ for the same X, Y, and W as Figure 2, where each underline indicates the prefix of J_1 that represents J_k with $i_{W[k-1]} + 1 \leq \mathsf{i} \leq i_{W[k]} - 1$.

i	J_5	J_4	J_3	J_2	J_1
1					$11 \circ 10 \circ 8 \circ 4 \circ 2$
2				$11 \circ 10 \circ 8 \circ 4$	$11 \circ 10 \circ 8 \circ 4 \circ 1$
3			$11 \circ 10 \circ 8$	$11 \circ 10 \circ 8$	$\underline{11 \circ 10 \circ 8} \circ 2$
4			$11 \circ 10 \circ 8$	$11 \circ 10 \circ 8$	$\underline{11 \circ 10 \circ 8} \circ 2$
5			$11 \circ 10 \circ 7$	$11 \circ 10 \circ 7$	$11 \circ 10 \circ 7 \circ 2$
6		$11 \circ 8$	$11 \circ 8 \circ 4$	$11 \circ 8 \circ 4$	$\underline{11 \circ 8} \circ 4 \circ 2$
7		$10 \circ 7$	$10 \circ 7 \circ 4$	$10 \circ 7 \circ 4$	$10 \circ 7 \circ 4 \circ 2$
8	9	9	$9 \circ 5 \circ 4$	$9 \circ 5 \circ 4$	$9 \circ 5 \circ 4$
9	9	9	$9 \circ 5$	$9 \circ 5$	$9 \circ 5$
10	9	9	$9 \circ 4$	$9 \circ 4$	$9 \circ 4$
11	empty	empty	5	5	5

Proof. It follows from Lemma 15 that $W[\kappa] = W^\star[\kappa] < W^\star[\kappa + 1]$. Due to definition of $W^\star[k]$, there exists no k-valid match w with $W[\kappa] < w < (|X| + 1, |Y| + 1)$ such that $j_w < j_{W^\star[k]}$. Therefore, if $W[k-1] < W^\star[k]$, then the match $(i, j_{W^\star[k]})$ in the lemma is $W[k]$, i.e., the first $W[k-1]$-follower. In addition, $i \leq i_{W^\star[k]}$ because $W^\star[k]$ is a match. This implies from $W^\star[k] < W^\star[k+1]$ due to Lemma 15 that if $(i, j_{W^\star[k]}) = W[k]$, then $W[k] < W^\star[k+1]$. Thus, induction proves the lemma. ◀

The reason for splitting the naive procedure into UPDATEW and FINDBRANCH is that once W is available, we can efficiently determine the existing branching match of it as follows. As long as searching only for the branching match, iterative updates of (i, j) in execution of GENSTREAM$(W[k-1], \text{SNK}, k)$ can start from $(i_{W[k]} - 1, j_{W[k]} + 1)$ rather than from $(|X|, j_{W[k-1]} + 1)$, if I and J are appropriately maintained. Let GENSTREAM$_k$ denote execution of the modified GENSTREAM as above and let J_k denote J maintained in GENSTREAM$_k$. A key observation for improving time efficiency is that J_1 can be used as J_k for any index k because J_k is a prefix of J_1 (see Table 2). This immediately implies that adopting J_1 as J maintained in GENSTREAM$_k$ instead of J_k and doing GENSTREAM$_k$ for each k from $L(X, Y)$ to 1 in descending order allow GENSTREAM$_k$ to initialize J not from scratch but staring from the resulting J after doing GENSTREAM$_{k+1}$, unless $k = L(X, Y)$. Another crucial observation is that only the first element $\mathsf{I}[1]$ of I is sufficient to be maintained in each GENSTREAM$_k$ and update of $\mathsf{I}[1]$ can be done by scanning $X[i_{W[k-1]} + 1 : i_{W[k]} - 1]$ instead of $X[i_{W[k-1]} + 1 : |X|]$. Based on these observations, we design FINDBRANCH as a modification of GENSTREAM$(\text{SRC}, \text{SNK}, 1)$.

3.2.4 Procedures updateW and findBranch

Based on Lemmas 15 and 16, UPDATEW obtains $W^\star[\kappa + 1 : L(X, Y)]$ and transform it to $W[\kappa + 1 : L(X, Y)]$ for each front-leaning LCS-match sequence W. Algorithm 3 presents a pseudo-code of UPDATEW, in which recursive decomposition of the problem of determining $W^\star[l + 1 : m - 1]$ is implemented as a sequential process. The following lemma assures that this procedure works well.

▶ **Lemma 17.** *For any front-leaning LCS-match sequence W, if $W[\kappa]$ is given as $\mathsf{W}[\kappa]$, then* UPDATEW *determines $W[\kappa + 1 : L(X, Y)]$ as $\mathsf{W}[\kappa + 1 : L(X, Y)]$ in $O(|X||Y| \log L(X, Y))$ time and $O(L(X, Y))$ space.*

Algorithm 3 updateW.

```
1  foreach d from ⌊log2(L(X,Y) − κ)⌋ + 1 down to 1 do
2      foreach p from 0 to ⌊(L(X,Y) − κ)/2^d⌋ do
3          l ← κ + 2^d · p
4          m ← min(l + 2^d, L(X,Y) + 1)
5          k ← l + 2^(d−1)
6          if k < m then
7              simulate genStream(W[l], W[m], l, m, k) until the first match w is output
8              W[k] ← w
9  foreach k from κ + 1 to L(X,Y) do
10     i ← i_{W[k−1]}
11     repeat i ← i + 1 until X[i] = Y[j_{W[k]}]
12     W[k] ← (i, j_{W[k]})
```

Proof. Lines 1 through 8 of updateW implemented as Algorithm 3 determines $W^\star[k]$, i.e., the $(W[\kappa], \mathrm{SNK})$-locally k-valid match having the least diagonal coordinate, as $\mathsf{W}[k]$ for each index k with $\kappa + 1 \leq k \leq L(X,Y)$. This can be verified because it follows from Lemma 15 that $W^\star[k]$ is the $(W^\star[l], W^\star[m])$-locally k-valid match with the least diagonal coordinate, implying that $W^\star[k]$ can be obtained as the first match output by $\mathrm{genStream}(W^\star[l], W^\star[m], k)$ due to Lemma 12. Execution of lines 1 through 8 completes in $O((|X| - i_{W[\kappa]})(|Y| - j_{W[\kappa]}) \log(L(X,Y) - \kappa))$ time because the number of iterations of lines 2 through 8 is $O(\log(L(X,Y) - \kappa))$ and it follows from Lemma 12 that each iteration executes in $O((|X| - i_{W[\kappa]})(|Y| - j_{W[\kappa]}))$ time. Once $W^\star[\kappa + 1 : L(X,Y)]$ is available as $\mathsf{W}[\kappa + 1 : L(X,Y)]$, for each index k from $\kappa + 1$ to $L(X,Y)$ in this order, lines 10 through 12 determine $W[k]$ as $\mathsf{W}[k]$ from $W[k-1]$ and $W^\star[k]$ in $O(i_{W[k]} - i_{W[k-1]})$ time based on Lemma 16. ◀

As mentioned in Section 3.2.3, we adopt a modification of $\mathrm{genStream}(\mathrm{SRC}, \mathrm{SNK}, 1)$ as findBranch to determine the existing branching match of W in $O(|X||Y|)$ time. A pseudo-code of findBranch is presented as Algorithm 4. Note that incJ in findBranch (lines 12 through 15) maintains only the first element $\mathsf{I}[1]$ of I maintained by $\mathrm{genStream}(W[k-1], \mathrm{SNK}, k)$ in a straightforward way. In contrast, decI in findBranch (lines 16 through 20) is the same as decI in $\mathrm{genStream}(\mathrm{SRC}, \mathrm{SNK}, 1)$, which implies that J maintained by $\mathrm{genStream}(W[k-1], \mathrm{SNK}, k)$ is a prefix of J maintained by findBranch. The following lemma claims that findBranch works successfully.

▶ **Lemma 18.** *For any front-leaning LCS-match sequence W given as W, findBranch determines the branching position and match of W as κ and $\mathsf{W}[\kappa]$, respectively, if existing, or sets κ to a dummy index, otherwise, in $O(|X||Y|)$ time and $O(L(X,Y))$ space.*

Proof. After initialization of variables i, I, J, and κ by line 1 of findBranch implemented as Algorithm 4, for each index k from $L(X,Y)$ to 1 in this order, lines 3 through 11 search for the next $W[k-1]$-follower of $W[k]$. In this process for k, i is initialized to $i_{W[k]} - 1$ by line 3 and j is initialized to $j_{w[k]} + 1$ by line 4. Since $\mathsf{I}[1]$ and J are also updated appropriately by lines 3 and 4, respectively, lines 5 through 11 simulate execution of $\mathrm{genStream}(W[k-1], \mathrm{SNK}, k)$ until the $(W[k-1], \mathrm{SNK})$-locally k-valid match (i, j) with $\mathsf{i} < i_{W[k]}$ and $\mathsf{j} > j_{W[k]}$ having the least diagonal coordinate is found. If found, then line 7 terminates the execution of

Algorithm 4 findBranch.

```
1  i ← |X| + 1; I, J ← the empty sequences; κ ← a dummy index
2  foreach k from L(X,Y) down to 1 do
3      while i ≥ i_{W[k]} do decI
4      j ← j_{W[k]}; I[1] ← i_{W[k]}; incJ
5      while j ≤ |Y| and i ≥ i_{W[k−1]} + 1 do
6          if |J| ≥ (L(X,Y) + 1) − k, i ≥ I[1], j ≤ J[(L(X,Y) + 1) − k], and X[i] = Y[j]
            then
7              κ ← k; W[κ] ← (I[1], j); halt
8          if i ≤ I[1] then
9              incJ
10         else
11             decI
12 procedure incJ
13     j ← j + 1; i ← i_{W[k−1]}
14     repeat i ← i + 1 until X[i] = Y[j] or i = I[1]
15     I[1] ← i
16 procedure decI
17     i ← i − 1; p ← |J|
18     foreach j from 1 to |Y| do
19         if p > 0 and J[p] = j then p ← p − 1
20         if X[i] = Y[j] then J[p + 1] ← j
```

findBranch after setting κ to k as the branching position of W and setting $\mathsf{W}[k]$ to $(\mathsf{I}[1], \mathsf{j})$, instead of (i, j), as the branching match of W due to an argument similar to the proof of Lemma 16.

For any index k with $1 \leq k \leq L(X,Y)$, the process for k implemented by lines 2 though 11 executes incJ $O(|Y| - j_{W[k]})$ times each in $O(i_{W[k]} - i_{W[k-1]})$ time. On the other hand, for each index i with $0 \leq \mathsf{i} \leq |X|$, decI is executed at most once in $O(|Y|)$ time throughout the execution of findBranch. Thus, findBranch runs in $O(|X||Y|)$ time and $O(L(X,Y))$ space. ◀

It immediately follows from Lemmas 17 and 18 that the following theorem holds.

▶ **Theorem 19.** *Algorithm* enumLCS *with procedures* updateW *and* findBranch *works as a linear-space LCS enumeration algorithm that outputs all distinct front-leaning LCS-match sequences of X and Y each in $O(|X||Y| \log L(X,Y))$ time.*

4 Concluding remarks

This article proposed an algorithm that takes strings X and Y as input and uses $O(L(X,Y))$ space, excluding the space for storing X and Y, to enumerate all distinct LCSs of X and Y each in $O(|X||Y| \log L(X,Y))$ time. The all-LCS graph, of size $O(|X||Y|)$, introduced in Section 3.1 allows us to determine all distinct LCSs each in $O(L(X,Y))$ time. Given this, whether we can remove the logarithmic factor of $L(X,Y)$ from the delay time achieved by the proposed algorithm is a natural question arising from a space-delay tradeoff perspective.

The author recently claimed in [12] that adopting a variant of the linear-space LCS-finding algorithm of Hirschberg [7] as an alternative of our $O(|X||Y| \log L(X,Y))$-time procedure UPDATEW can resolve this question in the affirmative.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 59–78. IEEE, 2015. doi:10.1109/FOCS.2015.14.

2 Alberto Apostolico. Improving the worst-case performance of the hunt-szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, 23(2):63–69, 1986. doi:10.1016/0020-0190(86)90044-X.

3 Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987. doi:10.1007/BF01840365.

4 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97. IEEE, 2015. doi:10.1109/FOCS.2015.15.

5 Yo-lun Chin, Chung-kwong Poon, et al. *A fast algorithm for computing longest common subsequences of small alphabet size*. University of Hong Kong, Department of Computer Science, 1990.

6 Jun-Yi Guo and Frank K Hwang. An almost-linear time and linear space algorithm for the longest common subsequence problem. *Information processing letters*, 94(3):131–135, 2005. doi:10.1016/j.ipl.2005.01.002.

7 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.

8 James W Hunt and Thomas G Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977. doi:10.1145/359581.359603.

9 Costas S Iliopoulos and M Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory of Computing Systems*, 45(2):355–371, 2009. doi:10.1007/s00224-008-9101-6.

10 William J Masek and Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.

11 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982. doi:10.1007/BF00264437.

12 Yoshifumi Sakai. Linear-space LCS enumeration with quadratic-time delay for two strings. *arXiv preprint arXiv:2504.05742*, 2025. doi:10.48550/arXiv.2504.05742.

13 Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974. doi:10.1145/321796.321811.