Computing longest common square subsequences

Takafumi Inoue
Department of Informatics, Kyushu University, Japan

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Heikki Hyyrö
Faculty of Natural Sciences, University of Tampere, Finland
heikki.hyyro@uta.fi

Hideo Bannai
Department of Informatics, Kyushu University, Japan
bannai@inf.kyushu-u.ac.jp
https://orcid.org/0000-0002-6856-5185

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

A *square* is a non-empty string of form YY. The *longest common square subsequence* (*LCSqS*) problem is to compute a longest square occurring as a subsequence in two given strings A and B. We show that the problem can easily be solved in $O(n^6)$ time or $O(|\mathcal{M}|n^4)$ time with $O(n^4)$ space, where n is the length of the strings and $\mathcal{M}$ is the set of matching points between A and B. Then, we show that the problem can also be solved in $O(\sigma|\mathcal{M}|^3 + n)$ time and $O(|\mathcal{M}|^2 + n)$ space, or in $O(|\mathcal{M}|^3 \log^2 n \log\log n + n)$ time with $O(|\mathcal{M}|^3 + n)$ space, where σ is the number of distinct characters occurring in A and B. We also study lower bounds for the LCSqS problem for two or more strings.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases squares, subsequences, matching rectangles, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.15

Acknowledgements The authors thank the anonymous referees for correcting errors involved in an earlier version of this paper.

1 Introduction

Computing the *longest common subsequence* (*LCS*) of given strings is the fundamental way to compare the strings. Given two strings A and B of length n each, the basic dynamic programming solution computes the LCS of A and B in $O(n^2)$ time and space [27]. While faster solutions for the LCS problem exist, such as those running in $O(n^2/\log^2 n)$ time for constant-size alphabets [22], and in $O(n^2(\log\log n)^2/\log^2 n)$ time or in $O(n^2 \log\log n/\log^2 n)$ time for non constant-size alphabets [5, 12] [1], no strongly sub-quadratic $O(n^{2-\epsilon})$-time

[1] Grabowski's method [12] works when the length m of one string is at least $\log^2 n$, where n is the length of the other string.

solutions are known for any constant $\epsilon > 0$. Difficulty in breaking this barrier is supported by recent studies on *conditional lower bounds* for string similarity measures: It is shown in [1] that if there is an $O(n^{2-\epsilon})$-time solution for the LCS problem with a constant $\epsilon > 0$, then the famous *strong exponential time hypothesis* (*SETH*) fails.

To reflect a priori knowledge to the solution to be found, many variants of the LCS problem where some *constraints* are introduced in the solution have been considered (see e.g. [7, 2, 14, 20, 9, 10, 28, 11, 29, 30, 8, 16, 19]).

This paper considers a new variant of the LCS problem where the solution must be a *square* (of form YY with some string Y), called the *longest common square subsequence* (*LCSqS*) problem defined as follows: Given two strings A and B of length n, compute (the length of) a longest square which appears as a subsequence in A and B. For instance, for $A =$ `babcabdbaca` and $B =$ `dbcacbbcacd`, their LCSqSs are `bacbac` and `bcabca` of length 6.

We propose several solutions for the LCSqS problem. We first show that there is a simple $O(n^6)$-time $O(n^4)$-space solution for the LCSqS problem. The algorithm is also improved to $O(|\mathcal{M}|n^4)$-time by using the set $\mathcal{M}$ of matching points between the two input strings. Albeit $\mathcal{M}$ can be as large as $O(n^2)$ in the worst case, it can be smaller in many cases. We then give two more sophisticated algorithms based on the set $\mathcal{R}$ of *matching rectangles*: one runs in $O(\sigma|\mathcal{M}||\mathcal{R}| + n) = O(\sigma|\mathcal{M}|^3 + n)$ time with $O(|\mathcal{M}|^2 + n)$ space, and the other in $O(|\mathcal{M}||\mathcal{R}| \log^2 \log\log n + |\mathcal{M}|^3 + n) = O(|\mathcal{M}|^3 \log^2 \log\log n + n)$ time with $O(|\mathcal{M}|^3 + n)$ space, where σ denotes the number of distinct characters that appear in both strings. These two solutions are faster than the simple $O(n^6)$-time or $O(|\mathcal{M}|n^4)$-time solutions when $\mathcal{M}$ is sparse. Note e.g. that under uniformly distributed random text $|\mathcal{M}| \approx n^2/\sigma$ and $|\mathcal{R}| \approx |\mathcal{M}|^2/\sigma \approx n^4/\sigma^3$, in which case the *expected* running times of our three algorithms would be $O(n^6/\sigma)$, $O(n^6/\sigma^3)$ and $O(n^6(\log^2 \log\log n + \sigma)/\sigma^4)$ respectively.

The set $\mathcal{M}$ of matching points can easily be computed in $O(|\mathcal{M}| + n)$ time under a common assumption that the input strings are over an integer alphabet of size $n^{O(1)}$.

We also study hardness of the LCSqS problem for two or more strings. The k-LCSqS problem is to compute the LCSqS of given $k \geq 2$ strings. We show that the k-LCSqS problem is at least as hard as the $2k$-LCS problem which asks to compute the LCS of $2k$ given strings. This implies that for unfixed k the k-LCSqS problem is NP-hard, and that for fixed k it seems hard to solve the k-LCSqS problem in $O(n^{k-\epsilon})$ time for any constant $\epsilon > 0$.

Related work

It is known that one can compute (the length of) a *longest square subsequence* (*LSqS*) of a single string of length n in $O(n^2)$ time and $O(n)$ space [18]. Also, it is shown in [1] that if there is an $O(n^{2-\epsilon})$-time solution for the LSqS problem with a constant $\epsilon > 0$, then the famous *strong exponential time hypothesis* (*SETH*) fails. Our results for the LCSqS problem can be seen as a generalization of these results for the LSqS problem.

Technically speaking, our results for the LCSqS problem are most related to those for the *longest common palindromic subsequence* (*LCPS*) problem, where the task is to find a longest palindrome that appears as a subsequence in both of the two strings A and B. Chowdhury et al. [8] were the first to consider the LCPS problem, giving an $O(n^4)$-time solution and an $O(|\mathcal{M}|^2 \log^2 n \log\log n + n)$-time solution[2]. Inenaga and Hyyrö [16] proposed another

[2] Our careful analysis reveals that Chowdhury et al.'s algorithm [8] uses at least $\Omega(\min\{|\mathcal{M}|^2 n^2 \log n, n^3\})$ space (and hence time), but it can be fixed to run in $O(|\mathcal{M}|^2 \log^2 n \log\log n + n)$ time using our technique proposed in Section 3.

algorithm which solves the LCPS problem in $O(\sigma|\mathcal{M}|^2 + n)$ time and $O(|\mathcal{M}|^2 + n)$ space. Very recently, Bae and Lee [3] showed how to solve the LCPS problem in $O(|\mathcal{M}|^2 + n)$ time. Inenaga and Hyyrö [16] also showed that the LCPS problem for two strings is at least as hard as the LCS problem for four strings, implying that it seems hard to solve the LCPS problem in $O(n^{4-\epsilon})$ time for any constant $\epsilon > 0$.

2 Preliminaries

Let Σ be the alphabet. An element X of Σ^* is called a string. The length of string X is denoted by $|X|$. For any $1 \leq i \leq |X|$, $X[i]$ denotes the ith character of X. For any $1 \leq i \leq j \leq |X|$, $X[i..j]$ denotes the substring of X beginning at position i and ending at position j.

A string X is said to be a *subsequence* of another string Y if there exists a sequence $1 \leq i_1 < \cdots < i_{|X|} \leq |Y|$ of increasing positions of Y such that $X = Y[i_1] \cdots Y[i_{|X|}]$. In other words, a subsequence of Y can be obtained by removing zero or more characters from Y. The *k-LCS problem* is to compute the length of a *longest common subsequence* (*LCS*) of given k strings, where $k \geq 2$. Let $LCS(A_1, \ldots, A_k)$ denote the length of a longest common subsequence of k strings $A_1, \ldots, A_k$. A non-empty string X of length $2k$ is called a *square* if there exists a string Y of length k such that $X = YY$. A square S is called a *square subsequence* of another string Y if square S is a subsequence of Y. Let $LCSqS(A, B)$ denote the length of a *longest common square subsequence* (*LCSqS*) of strings A and B. This paper deals with the problem of computing $LCSqS(A, B)$ for two given strings A and B. For simplicity, we assume that the input strings A and B are of the same length and let $n = |A| = |B|$. Our algorithms can easily be extended to the case where $|A| \neq |B|$ as well as to the case where we wish to compute one longest common square subsequence of A and B.

For two strings A and B, a pair (i, j) of positions $1 \leq i \leq |A|$ and $1 \leq j \leq |B|$ is said to be a *matching point* if $A[i] = B[j]$. The set of all matching positions of A and B is denoted by $\mathcal{M}(A, B)$, namely, $\mathcal{M}(A, B) = \{(i, j) \mid 1 \leq i \leq |A|, 1 \leq j \leq |B|, A[i] = B[j]\}$. We will abbreviate $\mathcal{M}(A, B)$ as $\mathcal{M}$ when it is clear from the context.

3 Algorithms

In this section, we present several algorithms for computing $LCSqS(A, B)$. In order to avoid processing unnecessary characters, we will assume that the input strings A and B have been already preprocessed by an alphabet reduction technique [16] as follows: First, we compute the lexicographical ranks of the characters in A and B. Assuming that A and B are drawn from an integer alphabet of size $n^{O(1)}$, this can be done in $O(n)$ time with radix sort. We then replace each character in A and B with its rank, turning A and B into strings over the integer alphabet $[1, 2n]$. Then we remove every character that appears only either in A or in B. It is clear that this preprocessing essentially preserves common subsequences between the original A and B and thus has no negative effect on computing $LCSqS(A, B)$. Note that $n \leq \mathcal{M}$ holds after alphabet reduction, while $\mathcal{M} = O(n^2)$ still also holds.

3.1 Simple Algorithm

Our first algorithm considers $\Theta(n^2)$ pairs of partitioning of A and B. Namely, we have that

$$LCSqS(A, B) = \max_{1 \leq i < n, 1 \leq j < n} \{2 \times LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n])\}.$$

This immediately implies an $O(n^6)$-time $O(n^4)$-space algorithm for computing $LCSqS(A, B)$, since the LCS of four strings can be computed in $O(n^4)$ time and space by standard DP.

The $O(n^6)$-time complexity can be improved as follows. For any matching point $(i, j) \in \mathcal{M}$, let i' (resp. j') be the smallest position such that $i < i'$, $j < j'$, and $(i', j') \in \mathcal{M}$. If such (i', j') does not exist, then let $i' = j' = n$.

▶ **Observation 1.** *For any $i \leq k < i'$ and $j \leq h < j'$, $LCS(A[1..k], A[k+1..n], B[1..h], B[h+1..n] = LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n]$.*

By Observation 1, it is sufficient for us to consider only $|\mathcal{M}|$ partition points between A and B. Hence, we can compute $LCSqS(A, B)$ in $O(|\mathcal{M}|n^4)$ time and $O(n^4)$ space.

3.2 $O(\sigma|\mathcal{M}|^3 + n)$-time algorithm

Here we present our $O(\sigma|\mathcal{M}|^3 + n)$-time algorithm for computing $LCSqS(A, B)$, where σ is the number of distinct characters occurring in A and B. This algorithm is based on Inenaga and Hyyrö's algorithm [16] which computes (the length of) a *longest palindromic common subsequence* of two given strings in $O(\sigma|\mathcal{M}|^2 + n)$ time. Consider a 2D plain where the string A corresponds to the vertical axis upward (i.e., $A[1]$ is on the bottom and $A[n]$ is on the top), and the string B corresponds to the horizontal axis rightward (i.e., $B[1]$ is on the left end and $B[n]$ is on the right end). Our key idea is to represent each common square subsequence of strings A and B by matching rectangles defined as follows: For $1 \leq i < j \leq n$ and $1 \leq k < l \leq n$, a tuple $r = (i, j, k, l)$ is said to be a *matching rectangle* iff $A[i] = A[j] = B[k] = B[l]$, and more specifically a c-matching rectangle iff $A[i] = A[j] = B[k] = B[l] = c$. For a matching rectangle $r = (i, j, k, l)$, (i, k) is said to be the left-bottom corner of r, and (j, l) is said to be the right-upper corner of r. Let $\mathcal{R}$ denote the set of matching rectangles of A and B. Notice $|\mathcal{R}| = O(|\mathcal{M}|^2)$. For two matching rectangles $r = (i, j, k, l)$ and $r' = (i', j', k', l')$, let

$$\begin{aligned} r = r' &\iff i = i', j = j', k = k', \text{ and } l = l' \\ r < r' &\iff i < i', j < j', k < k', \text{ and } l < l' \\ r \lhd r' &\iff i \leq i', j \leq j', k \leq k', l \leq l', \text{ and } r \neq r'. \end{aligned}$$

For two c-matching rectangles $r = (i, j, k, l)$ and $r' = (i', j', k', l')$, let

$$r \preceq r' \iff i \leq i', j \leq j', k \leq k' \text{ and } l \leq l'.$$

A sequence $\langle r_1, \ldots, r_m \rangle$ of matching rectangles is said to be a sequence of *diagonally overlapping matching rectangles* (*DOMRs*) iff $r_x < r_{x+1}$ for all $1 \leq x < m$, $i_m < j_1$ and $k_m < l_1$, where we use the notation $r_h = (i_h, j_h, k_h, l_h)$ for all $h = 1, \ldots, m$. The *size* of a sequence $\langle r_1, \ldots, r_m \rangle$ of DOMRs is the number m of overlapping rectangles in it.

The following observation lays the foundation to the algorithms of this subsection (and to the one of the following subsection as well):

▶ **Observation 2.** *There is a common square subsequence T of length $2m$ of strings A and B iff there exists a sequence $\langle r_1, \ldots, r_m \rangle$ of DOMRs of length m.*

See Figure 1 which depicts the relationship between common square subsequences and DOMRs for two strings A and B. By Observation 2, the problem of computing $LCSqS(A, B)$ reduces to the problem of finding a longest sequence of DOMRs.

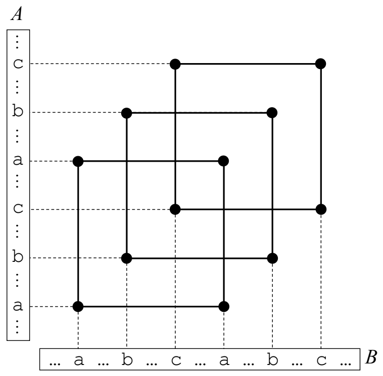


Figure 1 Illustration of the relationship between common square subsequences and DOMRs.

The basic idea of our algorithm is to extend a given sequence $S = \langle r_1, \ldots, r_m \rangle$ of DOMRs by adding a new matching rectangle to its right-end. We say that a c-matching rectangle $r = (i, j, k, l)$ is a *c-extension* of S if $\langle r_1, \ldots, r_m, r \rangle$ is a sequence of DOMRs. A c-extension r of S is *dominant* if the condition $r \preceq r'$ holds between r and any c-extension r' of S. The algorithms in this subsection are based on the following lemmas.

▶ **Lemma 3.** *Let $S = \langle r_1, \ldots, r_m \rangle$ be any sequence of DOMRs. If S has at least one c-extension, then S has a unique dominant c-extension r'. It is furthermore possible to compute any such r' in $O(1)$ time after initial preprocessing of A and B in $O(\sigma n)$ time and space.*

Proof. Consider $r' = (i', j', k', l')$, where $i' = \min(\{i \mid i_m < i < j_1, A[i] = c\} \cup \{n + 1\})$, $j' = \min(\{j \mid j_m < j, A[j] = c\} \cup \{n + 1\})$, $k' = \min(\{k \mid k_m < k < l_1, B[k] = c\} \cup \{n + 1\})$ and $l' = \min(\{l \mid l > l_m, B[l] = c\} \cup \{n+1\})$. If any of i', j', k' and l' holds the sentinel value $n + 1$ that corresponds to non-existence of a further suitable match with c, then S cannot have any c-extension. Otherwise $A[i'] = A[j'] = B[k'] = B[l'] = c$ and r' is a c-matching rectangle. Furthermore $i_m < i'$, $j_m < j'$, $k_m < k'$, $l_m < l'$, $i' < j_1$ and $k' < l_1$, so r' is a c-extension of S. If we assume the existence of another c-extension r' of S such that $r'' \preceq r'$ does not hold, then at least one of the definitions of i', j', k' and l' above is contradicted. Hence r' must be dominant. Finally, r' must clearly be unique: if also $r'' \neq r'$ is a dominant c-extension, then both $r' \preceq r''$ and $r'' \preceq r'$ must hold, but this is possible only if $r'' = r'$.

The values i' and j' can be computed in $O(1)$ time by using a precomputed table P_A of size $\sigma \times n$ that holds the values $P_A[c, h] = \min(\{i \mid h < i, A[h] = c\} \cup \{n + 1\})$ for all $c \in \Sigma$ and $1 \leq h \leq n$. The values k' and l' can be computed in $O(1)$ time by using an analogous precomputed table P_B with values $P_B[c, h] = \min(\{i \mid h < i, B[h] = c\} \cup \{n + 1\})$. Both tables can be precomputed in $O(\sigma n)$ time and space in a straight-forward manner. ◀

Note that the proof of Lemma 3 refers only to r_1 and r_m when determining the unique dominant extension of $\langle r_1, \ldots, r_m \rangle$: any inner rectangle r_i for $1 < i < m$ does not need to be considered. Thus all sequences of DOMRs that begin with the rectangle r_1 and end with the rectangle r_m share the same unique dominant extensions.

▶ **Lemma 4.** *Let $S = \langle r_1, \ldots, r_m \rangle$ be any sequence of at least two DOMRs. If any c-matching rectangle r_h with $1 < h \leq m$ is replaced by the dominant c-extension of $\langle r_1, \ldots, r_{h-1} \rangle$, also the resulting sequence of matching rectangles is a sequence of DOMRs.*

Proof. The lemma clearly holds if $h = m$, so consider the case $1 < h < m$. Let (i', j', k', l') be the dominant c-extension of $\langle r_1, \ldots, r_{h-1} \rangle$, and let $S' = \langle r'_1, \ldots, r'_m \rangle$ denote the sequence obtained from S by replacing r_h with (i', j', k', l'). S is a sequence of DOMRs, and thus

$i'_m = i_m < j_1 = j'_1$, $k'_m = k_m < l_1 = l'_1$, and $r_x < r_{x+1}$ for $1 \le x < m$. On the other hand $r'_{h-1} < r'_h$, as also $\langle r'_1, \dots, r'_h \rangle = \langle r_1, \dots, r_{h-1}, (i', j', k', l') \rangle$ is a sequence of DOMRs. Because r'_h is dominant, we have $r'_{h-1} < r'_h \preceq r_h < r_{h+1} = r'_{h+1}$, which in turn implies that $r'_h < r'_{h+1}$ for $1 \le h < m$, and hence S' fulfills all conditions of a sequence of DOMRs. ◀

Basic algorithm. The basic principle of our first rectangle-based algorithm, Algorithm 1, is to fix the first left-bottom matching rectangle r_b, and then try to extend it as long as possible to the right-upper direction. For each such starting rectangle r_b, we compute a dynamic programming table DP_{r_b} of size $O(|\mathcal{M}|^2)$ such that $DP_{r_b}[r_e]$ will finally store the length of the longest sequence of DOMRs beginning with r_b and ending with r_e, where r_e is either r_b itself or a dominant extension. In more detail, Algorithm 1 works as follows:

```
Algorithm 1:
  Preprocessing: Compute a list $L$ of all matching rectangles sorted according to $<$ and
  $\preceq$ by radix sorting all rectangles $(i, j, k, l)$ as 4-digit numbers.

  Compute longest sequence of DOMRs: For each matching rectangle $r_b$ (in any
  order), perform the following:
  (1) For each $r_e$ ($\neq r_b$), we initialize $DP_{r_b}[r_e] \leftarrow 0$. We let $DP_{r_b}[r_b] \leftarrow 2$.
  (2) Suppose $r_b$ is the $i$th element of $L$. For each $j = i + 1, \dots, |L|$ in increasing order,
      let $r \leftarrow L[j]$ and attempt to extend a sequence $\langle r_b, \dots, r \rangle$ of DOMRs as follows:
      (a) If $DP_{r_b}[r] = 0$, then no sequence of DOMRs of form $\langle r_b, \dots, r \rangle$ exists.
      (b) Otherwise, for each character $c$, try to compute the unique dominant $c$-extension
          $r'$ of any sequence $\langle r_b, \dots, r \rangle$ of DOMRs which begins with $r_b$ and ends with $r$.
          If such $r'$ exists, set $DP_{r_b}[r'] \leftarrow \max\{DP_{r_b}[r'], DP_{r_b}[r] + 2\}$.
  (3) If the maximum value in $DP_{r_b}$ exceeds the current best solution, then update it.
```

Let us explain the correctness of Algorithm 1. Lemma 4 guarantees that an optimal sequence of DOMRs can be constructed by considering only dominant extensions. Consider any such optimal sequence of DOMRs $S = \langle r_1, \dots, r_m \rangle$. The outer loop of Algorithm 1 will at some point select $r_b = r_1$. As $r \leftarrow L[j]$ are processed in increasing order of j, the sorting order of L guarantees that rectangles r_i of S will be selected as the current r in the order $i = 1, \dots, m$. For each such $r = r_i$, the algorithm uses Lemma 3 to consider all possible dominant extensions, including also the extension r_{i+1} if $i < m$. A simple inductive argument shows that the values $DP_{r_1}[r_i]$ will become correctly computed in the order $i = 1, \dots, m$.

Let us analyze the efficiency of Algorithm 1. Constructing the tables P_A and P_B takes $O(\sigma n)$ time and space. Note that alphabet reduction guarantees that $O(\sigma n) = O(\sigma|\mathcal{M}|)$. Since $1 \le i, j, k, l \le n$ for each matching rectangle (i, j, k, l), we obtain a sorted list L of all $O(|\mathcal{M}|^2)$ matching rectangles in $O(|\mathcal{M}|^2 + n)$ time and space by radix sort. Hence the preprocessing takes $O(|\mathcal{M}|^2 + n)$ total time and space. We test no more than σ characters for any cell $DP_{r_b}[r]$ of the dynamic programming table DP_{r_b}. By Lemma 3, we can compute a unique dominant c-extension in $O(1)$ time, if it exists. Since there are $O(|\mathcal{M}|^2)$ candidates for r_b and $O(|\mathcal{R}|) = O(|\mathcal{M}|^2)$ candidates for r, Algorithm 1 takes overall $O(\sigma|\mathcal{M}|^4 + n)$ time and $O(|\mathcal{M}|^2 + n)$ space.

Improved algorithm. Now we show how to reduce the number of candidates for the starting rectangle r_b. We give proof for Lemma 5. Lemmas 6 and 7 can be proven similarly.

▶ **Lemma 5.** *Let $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$ and $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$ be any matching rectangles s.t. $i_{b_1} < i_{b_2}$, $j_{b_1} = j_{b_2}$, $k_{b_1} = k_{b_2}$, and $l_{b_1} = l_{b_2}$. Let ℓ_1 and ℓ_2 be the lengths of LCSqS of A and B whose corresponding sequences of DOMRs begin with r_{b_1} and r_{b_2}, respectively. Then, $\ell_1 \geq \ell_2$.*

Proof. See Figure 2 for illustration. It follows from $j_{b_1} = j_{b_2}$, $k_{b_1} = k_{b_2}$, and $l_{b_1} = l_{b_2}$ that the two matching rectangles r_{b_1} and r_{b_2} correspond to the same character. Let $\langle r_{b_2,1}, r_{b_2,2}, \ldots, r_{b_2,\ell_2} \rangle$ be any sequence of DOMRs which begins with r_{b_2} and represents a common square subsequence of length ℓ_2, namely $r_{b_2} = r_{b_2,1}$. Since $i_{b_1} < i_{b_2}$, $j_{b_1} = j_{b_2}$, $k_{b_1} = k_{b_2}$, and $l_{b_1} = l_{b_2}$, $\langle r_{b_1}, r_{b_2,2}, \ldots, r_{b_2,\ell_2} \rangle$ is a sequence of DOMRs which begins with r_{b_1} and represents a common square subsequence of length ℓ_2. This implies that $\ell_1 \geq \ell_2$. ◀

▶ **Lemma 6.** *Let $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$ and $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$ be any matching rectangles s.t. $i_{b_1} = i_{b_2}$, $j_{b_1} = j_{b_2}$, $k_{b_1} < k_{b_2}$, and $l_{b_1} = l_{b_2}$. Let ℓ_1 and ℓ_2 be the lengths of LCSqS of A and B whose corresponding sequences of DOMRs begin with r_{b_1} and r_{b_2}, respectively. Then, $\ell_1 \geq \ell_2$.*

▶ **Lemma 7.** *Let $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$ and $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$ be any matching rectangles such that $i_{b_1} < i_{b_2}$, $j_{b_1} = j_{b_2}$, $k_{b_1} < k_{b_2}$, and $l_{b_1} = l_{b_2}$. Let ℓ_1 and ℓ_2 be the lengths of longest common square subsequences of A and B whose corresponding sequences of DOMRs begin with r_{b_1} and r_{b_2}, respectively. Then, $\ell_1 \geq \ell_2$.*

It follows from Lemmas 5–7 that it suffices to consider only all right-upper corners (j_b, l_b) instead of all matching rectangles $r_b = (i_b, j_b, k_b, l_b)$. Namely, for each arbitrarily fixed right-upper corner (j_b, l_b) such that $A[j_b] = B[l_b] = c$, we can always use $(i_{\min}, k_{\min})$ as its left-bottom corner, where $i_{\min}$ and $k_{\min}$ are respectively the left-most occurrences of character c in A and B. The following is our improved algorithm.

Algorithm 2:

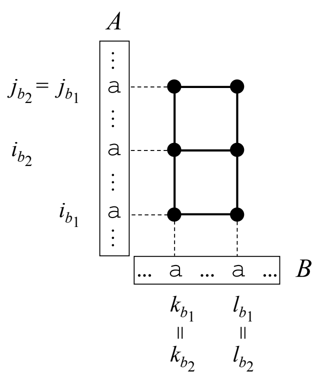
Preprocessing: As in Algorithm 1, but now also precompute positions $i_b = \min\{i \mid A[i] = c\}$ and $k_b = \min\{k \mid B[k] = c\}$ for each character c that appears in A and B.

Computing longest sequence of DOMRs: For each matching point $p_b = (j_b, l_b) \in \mathcal{M}$ we perform the following:

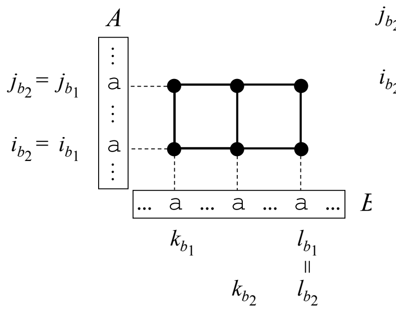
- **(i)** Let $c = A[j_b] = B[l_b]$. We compute $i_b = \min\{i \mid A[i] = c\}$ and $k_b = \min\{k \mid B[k] = c\}$, and let $r_b \leftarrow (i_b, j_b, k_b, l_b)$. If $i_b = j_b$ or $k_b = l_b$, then we stop processing the current matching point and proceed to the next matching point in $\mathcal{M}$.
- **(ii)** Perform the same procedures (1)–(3) as in Algorithm 1.
- **(iii)** If the maximum value in DP_{r_b} exceeds the current best solution, then update it.

The correctness of Algorithm 2 follows from that of Algorithm 1 and Lemmas 5-7.

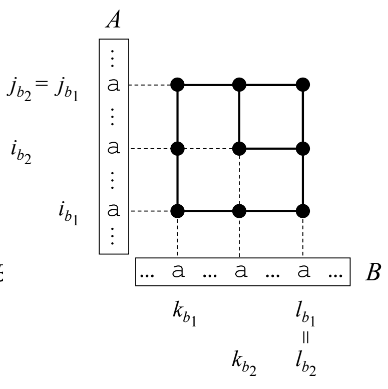
Let us analyze the efficiency of Algorithm 2. For all characters c, we can precompute $i_b = \min\{i \mid A[i] = c\}$ and $k_b = \min\{k \mid B[k] = c\}$ in total $O(n)$ time and space. The other preprocessing steps are the same as in Algorithm 1 and take $O(\sigma|\mathcal{M}| + n)$ total time and space. There are $O(|\mathcal{M}|)$ candidates for the right-upper corner $p_b = (j_b, l_b)$ of the first matching rectangle from which considered sequences of DOMRs begin. For each $p_b = (j_b, l_b)$, its left-bottom corner (i_b, k_b) can be retrieved in $O(1)$ time. We again test no more than σ characters for any cell $DP_{r_b}[r]$, and Lemma 3 allows to check each unique dominant c-extension in $O(1)$ time. Since there are $O(|\mathcal{M}|)$ candidates for r_b and $O(|\mathcal{R}|) = O(|\mathcal{M}|^2)$



■ **Figure 2** Illustration for Lemma 5.



■ **Figure 3** Illustration for Lemma 6.



■ **Figure 4** Illustration for Lemma 7.

candidates for r, the whole algorithm takes overall $O(\sigma|\mathcal{M}|^3 + n)$ time and $O(|\mathcal{M}|^2 + n)$ space. We have shown the following theorem:

▶ **Theorem 8.** *We can compute $LCSqS(A, B)$ in $O(\sigma|\mathcal{M}|^3 + n)$ time and $O(|\mathcal{M}|^2 + n)$ space.*

3.3 $O(|\mathcal{M}|^3 \log^2 n \log\log n + n)$-time algorithm

In this section we propose an $O(|\mathcal{M}|^3 \log^2 n \log\log n + n)$-time and $O(|\mathcal{M}|^3 + n)$-space algorithm for computing $LCSqS(A, B)$.

For any $1 \le i < s \le j \le n$ and $1 \le k < t \le l \le n$, let $LCSqS_{s,t}(i, j, k, l) = 2 \times LCS(A[1..i], A[s..j], B[1..k], B[t..l])$.

By definition, $LCSqS(A, B) = \max_{1 \le i < s \le j \le n, 1 \le k < t \le l \le n, (s,t) \in \mathcal{M}} \{LCSqS_{s,t}(i, j, k, l)\}$.

Now, let $(s, t) \in \mathcal{M}$ be an arbitrarily fixed matching point between A and B. This corresponds to Observation 1. A recurrence for computing $LCSqS_{s,t}(i, j, k, l)$ is given as follows:

$$LCSqS_{s,t}(i,j,k,l) = \begin{cases} \max_{(i',j',k',l') < (i,j,k,l)} \{LCSqS_{s,t}(i',j',k',l')\} + 2 & \begin{array}{l}((i,j,k,l) \in \mathcal{R},\\ 1 \le i < s \le j \le n,\\ 1 \le k < t \le l \le n)\end{array} \\ \max_{(i',j',k',l') \lhd (i,j,k,l)} \{LCSqS_{s,t}(i',j',k',l')\} & \begin{array}{l}((i,j,k,l) \notin \mathcal{R},\\ 1 \le i < s \le j \le n,\\ 1 \le k < t \le l \le n)\end{array} \\ 0 & \text{(otherwise)} \end{cases} \tag{1}$$

Our technique for computing $LCSqS_{s,t}(i, j, k, l)$ is similar to Chowdhury et al.'s method [8] for computing longest common palindromic subsequences, which uses the following well-known van Emde Boas tree data structure: Let $\mathcal{S}$ be a set of integers from the universe $[1, U]$. The van Emde Boas tree for $\mathcal{S}$ takes $\Theta(U)$ space and supports predecessor/successor queries and insertion/deletion operations on $\mathcal{S}$ in $O(\log\log U)$ time each [26].

Let $(s, t) \in \mathcal{M}$ be an arbitrary fixed matching point. We plot a point (i, j, k) on the 3D grid $[1..n] \times [1..n] \times [1..n]$ if and only if there is a matching rectangle of form $(i, j, k, *)$, namely, one having i, j, k as its first three coordinates. This 3D point (i, j, k) will finally be associated with $\max_{(i,j,k,l) \in \mathcal{R}} \{LCSqS_{s,t}(i, j, k, l)\}$.

Now we show how to compute those associated values for all the 3D points. We consider the permuted tuples (l, i, j, k) and sort them as 4-digit numbers, like we did for L in Section 3.2. We process the permuted tuples in this sorted order. Suppose we are to process a permuted tuple (l, i, j, k) such that its original tuple (i, j, k, l) is in $\mathcal{R}$. It is now guaranteed that we have processed all tuples $(l', *, *, *)$ with $l' < l$. Therefore, if z is the maxima among the associated values of all 3D points in the range $[1..i-1] \times [1..j-1] \times [1..k-1]$, then we have that $\mathit{LCSqS}_{s,t}(i, j, k, l) = z + 2$ (see also the recurrence (1) above). We maintain these 3D points with a variant of the 3D range tree [4]. Then, the maxima z can be efficiently retrieved by querying the point with the maximum associated value in the range $[1..i-1] \times [1..j-1] \times [1..k-1]$. If there is no existing 3D point (i, j, k), then we insert this point with the associated value $z + 2$. Otherwise, we update the associated value of the already existing 3D point (i, j, k) with $z + 2$.

The 3D range tree is a three layered data structure: The top layer tree maintains the first i-coordinate $[1..n]$, and each of its nodes is associated with a middle layer tree. Each middle layer tree maintains the second j-coordinate $[1..n]$, and each of its nodes is associated with a bottom layer tree. Each bottom layer tree maintains the third k-coordinate $[1..n]$. Since each bottom layer tree can contain $O(n)$ nodes, each middle layer tree can contain at most $O(n)$ nodes, and the top layer can contain at most $O(n)$ nodes, the total size of the 3D range tree data structure is trivially bounded by $O(n^3) = O(|\mathcal{M}|^3)$. Since at most $O(|\mathcal{M}|^2)$ points are inserted to the 3D range tree and since $|\mathcal{M}| = O(n^2)$, the 3D range tree supports range maxima queries and insertions of new points in $O(\log^3(|\mathcal{M}|^2)) = O(\log^3 n)$ time.

Next, we improve the query and update times from $O(\log^3 n)$ to $O(\log^2 n \log\log n)$. Chowdhury et al. [8] claimed that using the technique from [15] it is possible to replace each 1D range tree on the bottom layer with a van Emde Boas tree data structure [26], leading to $O(\log^2 n \log\log n)$ query and update times. However, the way how van Emde Boas trees are used in the approach of [15] indeed requires to maintain a set of integers in the universe of size $\Theta(n^2)$. This implies that each van Emde Boas tree requires $\Theta(n^2)$ space. Since the total size of the top layer tree and the middle layer trees is $O(n^2)$, and since each node of a middle layer tree maintains a van Emde Boas tree of size $O(n^2)$, it takes $O(n^4)$ space[3]. This is, however, prohibitive since it can exceed our target time bound $O(|\mathcal{M}|^3 \log^2 n \log\log n)$ when the set $\mathcal{M}$ of matching points is sparse (e.g., when $|\mathcal{M}| = \Theta(n)$). Below, we will reduce the space requirement for the van Emde Boas trees used in our data structure.

Space efficient 3D range tree with van Emde Boas trees. We briefly recall how the algorithm of [15] computes the maxima in a given range using a van Emde Boas tree. Let $D[1..n]$ be an array of monotonically non-decreasing non-negative integers from $[0..n]$, namely, $0 \leq D[k] \leq n$ for all $1 \leq k \leq n$ and $D[k] \leq D[k+1]$ for all $1 \leq k < n$. We will store in D the associated values of 3D points in increasing order of positions, and in the sequel we assume that $D[k+1] - D[k] \in \{0, 2\}$. Let $\mathit{RMQ}_S(1, k)$ denote a query to return the maxima in the sub-array $D[1..k]$ for $1 \leq k \leq n$. For any integer val $(1 \leq \mathsf{val} \leq n)$, if some entry of D stores val, then we insert the pair $(\mathsf{pos}, \mathsf{val})$ s.t. pos is the rightmost position in D that stores val. For instance, if $D = [0, 0, 2, 4, 4, 6]$, then the van Emde Boas tree maintains the set $\{(2, 0), (3, 2), (5, 4), (6, 6)\}$ of integer pairs. However, since a van Emde Boas tree is an integer data structure, we convert each pair $(\mathsf{pos}, \mathsf{val})$ to integer $\mathsf{pos} \times (n+1) + \mathsf{val}$ and insert

[3] A more careful analysis reveals that the total size of this variant of the 3D range tree with van Emde Boas bottom layer trees is $O(|\mathcal{M}|^2 n^2 \log n)$, however, this can also exceed $O(|\mathcal{M}|^3 \log^2 n \log\log n)$ when $\mathcal{M}$ is sparse.

it to the van Emde Boas tree. Now, observe that computing $RMQ_S(1, k)$ reduces to finding the successor for the pair $(k-1, n)$.

The value of $LCSqS_{s,t}(i, j, k, l)$ is monotonically non-decreasing as i, j, k, l grow, for fixed s and t. Also, val in our case is in range $[0, n]$. Hence, we can use the above approach in our algorithm. The remaining problem is that the universe size is $\Theta(n^2)$, meaning that each van Emde Boas tree above takes $\Theta(n^2)$ space.

To reduce the space requirement, we maintain only pos's in our van Emde Boas tree, and store val's in an array V of size n so that $V[\mathsf{pos}] = \mathsf{val}$. We let $V[i] = -1$ if i does not exist in the van Emde Boas tree. Let us denote by **Pos_vEB** and **ValPos_vEB** the van Emde Boas trees which store pos's only and pairs $(\mathsf{pos}, \mathsf{val})$, respectively. Namely, the former is ours and the latter is the method from [15]. It is sufficient for **ValPos_vEB** to support insertions, deletions, and successor queries. These operations and queries can be simulated by our **Pos_vEB** as follows: When a pair $(\mathsf{pos}, \mathsf{val})$ is inserted to **ValPos_vEB**, then we insert pos to **Pos_vEB** and set $V[\mathsf{pos}] \leftarrow \mathsf{val}$. Notice that at any moment **ValPos_vEB** never maintains two pairs $(\mathsf{pos}_1, \mathsf{val})$ and $(\mathsf{pos}_2, \mathsf{val})$ with $\mathsf{pos}_1 \neq \mathsf{pos}_2$ for the same associated value val, since otherwise we get $\operatorname{argmax}\{i \mid D[i] = \mathsf{val}\} = \mathsf{pos}_1 \neq \mathsf{pos}_2 = \operatorname{argmax}\{i \mid D[i] = \mathsf{val}\}$, a contradiction. Therefore, we can simulate insertions on **ValPos_vEB** with **Pos_vEB** and V as above. When we delete a pair $(\mathsf{pos}, \mathsf{val})$ from **ValPos_vEB**, then we delete pos from **Pos_vEB** and modify the value stored in $V[\mathsf{pos}]$ accordingly. When we query the successor $(\mathsf{pos}, \mathsf{val})$ of $(k-1, n)$ on **ValPos_vEB**, then we query the successor pos of $k-1$ on **Pos_vEB**, and retrieve $\mathsf{val} = V[\mathsf{pos}]$. This way, we can simulate **ValPos_vEB** with **Pos_vEB** of $O(n)$ total space, retaining $O(\log\log n)$ time efficiency for insertion/deletion operations and successor queries. Since the total number of **ValPos_vEB**'s is linear in the number of nodes in the top and middle layer trees, our version of 3D range tree, named **New_vEB_3DRangeTree**, takes a total of $O(n^3)$ space and supports range maxima queries in $O(\log^2 n \log\log n)$ time for query ranges of form $[1..i] \times [1..j] \times [i..k]$. The whole algorithm is the following:

Algorithm 3:

Preprocessing: For all matching rectangles $(i, j, k, l) \in \mathcal{R}$, sort the permuted tuples (l, i, j, k) as 4-digit numbers. Initialize **New_vEB_3DRangeTree**, so that no points are inserted and every entry of array V in each **Pos_vEB** stores 0.

Compute $LCSqS_{s,t}(i, j, k, l)$**:** For each matching point $(s, t) \in \mathcal{M}$, perform the following:

(1) Process each permuted tuple (l, i, j, k) in the sorted order. Compute $LCSqS_{s,t}(i, j, k, l)$ according to recurrence (1): For each different value of l, let $\mathcal{PT}_l$ denote the list of permuted tuples whose first elements are l. For each permuted tuple $q = (l, i, j, k) \in \mathcal{PT}_l$, perform the following:
- If $i < s < j$ and $k < t < l$, then using **New_vEB_3DRangeTree** find a 3D point with the maximum associated value z_q in range $[1..i-1] \times [1..j-1] \times [1..k-1]$.
- After computing $LCSqS_{s,t}(i, j, k, l)$ for all permuted tuples $q = (l, i, j, k) \in \mathcal{PT}_l$, insert $z_q + 2$ in (i, j, k) to **New_vEB_3DRangeTree** for all such permuted tuples in $\mathcal{PT}_\ell$.

(2) If some value $LCSqS_{s,t}(i, j, k, l)$ exceeds the currently stored maxima, we update it. Then, delete all existing 3D points from **New_vEB_3DRangeTree**.

Let us recall recurrence (1) to see why Algorithm 3 correctly computes $LCSqS_{s,t}(i,j,k,l)$. The rule for the second case (where $(i,j,k,l) \in \mathcal{R}$) requires $(i',j',k',l') < (i,j,k,l)$. To reflect this, Algorithm 3 processes all permuted tuples in $\mathcal{PT}_l$ for each difference value of l and in increasing order of l. After processing all permuted tuples $q = (l.i,j,k) \in \mathcal{PT}_l$, we can safely insert the value $z_q + 2$ in the corresponding 3D point (i,j,k) for all such tuples q, and can proceed to the permuted tuples with larger first values.

Let us analyze the efficiency of Algorithm 3. For preprocessing, we use $O(n)$ time and space for alphabet reduction, for sorting the permuted tuples (l,i,j,k), and for initializing **New_vEB_3DRangeTree**. For each $(s,t) \in \mathcal{M}$, we compute $LCSqS_{s,t}(i,j,k,l)$ with each $(i,j,k,l) \in \mathcal{R}$, by querying and updating **New_vEB_3DRangeTree**. Each query and update here take $O(\log^2 n \log\log n)$ time. After computing all $LCSqS_{s,t}(i,j,k,l)$ for the current matching point (s,t), we delete all 3D points from **New_vEB_3DRangeTree**. Thus it takes $O(|\mathcal{R}| \log^2 n \log\log n)$ time for each $(s,t) \in \mathcal{M}$. **New_vEB_3DRangeTree** uses $O(n^3) = O(|\mathcal{M}|^3)$ space (recall that $n \leq |\mathcal{M}|$ holds after alphabet reduction). Since $|\mathcal{R}| = O(|\mathcal{M}|^2)$, Algorithm 3 takes a total of $O(|\mathcal{M}||\mathcal{R}| \log^2 n \log\log n + |\mathcal{M}|^3 + n) = O(|\mathcal{M}|^3 \log^2 n \log\log n + n)$ time and $O(|\mathcal{M}|^3 + n)$ space.

We have shown the following theorem:

▶ **Theorem 9.** *We can compute $LCSqS(A,B)$ in $O(|\mathcal{M}|^3 \log^2 n \log\log n + n)$ time and $O(|\mathcal{M}|^3 + n)$ space.*

4 Hardness results on the LCSqS problem

The k-LCSqS problem is to compute an LCSqS of k given strings. For simplicity, we assume that each given string is of length n.

▶ **Lemma 10.** *For any $k \geq 2$, the k-LCS problem can be reduced in linear time to the $\lceil k/2 \rceil$-LCSqS problem.*

Proof. Our proof uses an idea similar to [6] and [16]. We first consider the case where k is even. Let $A_1, \ldots, A_k$ be the input strings for the k-LCS problem. For each $1 \leq i \leq k/2$, we construct a string B_i of length $4n+2$ such that $B_i = A_{2i-1}\$^{n+1}A_{2i}\$^{n+1}$, where \$ is a special character which does not appear in $A_1, \ldots, A_k$. Let Z be any LCSqS of $B_1, \ldots, B_{k/2}$. Since each A_j $(1 \leq j \leq k)$ is of length n, Z must be of form $X\$^{n+1}X\$^{n+1}$. Then, clearly the string X is a longest common subsequence of the original strings $A_1, \ldots, A_k$.

For odd k, it suffices to consider the same strings B_i for $1 \leq i \leq \lfloor k/2 \rfloor$ and one additional string $B_{\lceil k/2 \rceil} = A_k\$^{n+1}A_k\$^{n+1}$. This completes the proof. ◀

By Lemma 10, the k-LCSqS problem is NP-hard for an unfixed k. For an arbitrarily fixed k, Abboud et al. [1] showed that if there exist a constant $\epsilon > 0$, an integer $k \geq 2$, and an algorithm which solves the k-LCS problem for an alphabet of size $O(k)$ in $O(n^{k-\epsilon})$ time, then the famous *strong exponential time hypothesis* (*SETH*) is false. This suggests that it seems hard to compute $LCSqS(A,B)$ in $O(n^{4-\epsilon})$ time for any $\epsilon > 0$.

5 Discussions

We observe that it seems difficult to shave the $|\mathcal{M}|^3$ term in the time complexity of any matching-rectangle-based algorithm for computing the LCSqS: For instance, in both Algorithm 2 and Algorithm 3, we first fix a matching point in $\mathcal{M}$, and this indeed corresponds to the $|\mathcal{M}|$ term in the $O(|\mathcal{M}|n^4)$-time complexity of the simple solution for computing

$LCSqS(A,B)$. The rest of all these algorithms exactly computes the LCS of the four strings obtained by partitioning A and B at a given matching point using at least $O(|\mathcal{M}|^2)$ or $O(n^4)$ time. This seems almost best possible, since it is widely believed that there is no algorithm which computes the LCS of four strings in $O(n^{4-\epsilon})$ time for any $\epsilon > 0$ (recall Section 4).

Can we break the $O(|\mathcal{M}|^3)$ or $O(n^6)$ barrier? The only hope seems to generalize an *incremental LCS computation* algorithm for two strings ([21, 24, 17, 23, 25, 13]) to the case of four strings. This would help us update a data structure for $LCS(A[1..i-1], A[i..n], B[1..j-1], B[j..n])$ to that for $LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n])$ in faster than $O(n^4)$ time. However, this seems difficult, too. We investigated whether Kim and Park's method [17], the simplest incremental LCS algorithm for two strings, can be generalized to more strings. Their algorithm uses the differential encoding of the 2-dimensional DP tables (for two strings) before and after the first character of one string is deleted, and they showed that only $O(n)$ entries of the differential encoding need to be updated. However, our preliminary experiments for 3-dimensional DP tables (i.e. for three strings) already suggested that there would be more than $O(n^2)$ entries in the differential encoding that need to be updated.

Overall, it is an intriguing open question how one can close the (almost) quadratic gap between the upper and lower bounds for the LCSqS problem.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proc. FOCS 2015*, pages 59–78, 2015.

2 Abdullah N. Arslan. Regular expression constrained sequence alignment. *J. Disc. Algo.*, 5(4):647–661, 2007.

3 Sang Won Bae and Inbok Lee. On finding a longest common palindromic subsequence. *Theor. Comput. Sci.*, 710:29–34, 2018.

4 Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.

6 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proc. FOCS 2015*, pages 79–97, 2015.

7 Francis Y. L. Chin, Alfredo De Santis, Anna Lisa Ferrara, N. L. Ho, and S. K. Kim. A simple algorithm for the constrained sequence problems. *Inf. Process. Lett.*, 90(4):175–179, 2004.

8 Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. *Fundam. Inform.*, 129(4):329–340, 2014.

9 Sebastian Deorowicz. Quadratic-time algorithm for a string constrained LCS problem. *Inf. Process. Lett.*, 112(11):423–426, 2012.

10 Effat Farhana and M. Sohel Rahman. Doubly-constrained LCS and hybrid-constrained LCS problems revisited. *Inf. Process. Lett.*, 112(13):562–565, 2012.

11 Effat Farhana and M. Sohel Rahman. Constrained sequence analysis algorithms in computational biology. *Inf. Sci.*, 295:247–257, 2015.

12 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016.

13 Heikki Hyyrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *J. Disc. Algo.*, 34:2–17, 2015.

14 Costas S. Iliopoulos and Mohammad Sohel Rahman. New efficient algorithms for the LCS and constrained LCS problems. *Inf. Process. Lett.*, 106(1):13–18, 2008.

15 Costas S. Iliopoulos and Mohammad Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory Comput. Syst.*, 45(2):355–371, 2009.

16 Shunsuke Inenaga and Heikki Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Inf. Process. Lett.*, 129:11–15, 2018.

17 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *J. Disc. Algo.*, 2:302–312, 2004.

18 Adrian Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *Proc. SPIRE 2004*, pages 93–100, 2004.

19 Keita Kuboi, Yuta Fujishige, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster str-ic-lcs computation via rle. In *Proc. CPM 2017*, page 25:1–25:12, 2017.

20 Gregory Kucherov, Tamar Pinhas, and Michal Ziv-Ukelson. Regular language constrained sequence alignment revisited. *J. Computational Biology*, 18(5):771–781, 2011.

21 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comp.*, 27(2):557–582, 1998.

22 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

23 Yoshifumi Sakai. An almost quadratic time algorithm for sparse spliced alignment. *Theory Comput. Syst.*, 48(1):189–210, 2011.

24 Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application in finding all approximate repeats in strings. *SIAM J. Comp.*, 27(4):972–992, 1998.

25 Alexandre Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2007. URL: `http://arxiv.org/abs/0707.3619`.

26 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. FOCS 1975*, pages 75–84, 1975.

27 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

28 Daxin Zhu and Xiaodong Wang. A simple algorithm for solving for the generalized longest common subsequence (LCS) problem with a substring exclusion constraint. *Algorithms*, 6(3):485–493, 2013.

29 Daxin Zhu, Yingjie Wu, and Xiaodong Wang. An efficient algorithm for a new constrained LCS problem. In *Proc. ACIIDS 2016*, pages 261–267, 2016.

30 Daxin Zhu, Yingjie Wu, and Xiaodong Wang. An efficient dynamic programming algorithm for STR-IC-STR-EC-LCS problem. In *Proc. GPC 2016*, pages 3–17, 2016.