Maximal Common Subsequence Algorithms

Yoshifumi Sakai
Graduate School of Agricultural Science, Tohoku University
468-1, Aza-Aoba, Aramaki, Aoba-ku, Sendai 980-0845, Japan
yoshifumi.sakai.c7@tohoku.ac.jp

Abstract

A common subsequence of two strings is maximal, if inserting any character into the subsequence can no longer yield a common subsequence of the two strings. The present article proposes a (sub)linearithmic-time, linear-space algorithm for finding a maximal common subsequence of two strings and also proposes a linear-time algorithm for determining if a common subsequence of two strings is maximal.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases algorithms, string comparison, longest common subsequence, constrained longest common subsequence

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.1

1 Introduction

A subsequence of a string of characters is obtained from the string by deleting any number of not necessarily contiguous characters at any position. A common subsequence of two strings can be though of as a pattern common to the strings. A common subsequence is maximal, if inserting any character into the subsequence can no longer yield a common subsequence. Hence, any common subsequence can be found as a subsequence of some maximal common subsequence. The present article considers the problem of finding a maximal common subsequence of two strings both of length $O(n)$ over an alphabet set of $O(n)$ characters for some positive integer n and also considers the problem of determining if a given common subsequence of the two strings is maximal.

A longest one of maximal common subsequences is called a longest common subsequence (an LCS). It is well known that the dynamic programming algorithm of Wagner and Fisher [10] finds an LCS of two $O(n)$-length strings in $O(n^2)$ time and $O(n^2)$ space. Moreover, the divide-and-conquer version developed by Hirschberg [6] reduces the required space to $O(n)$ without increasing the asymptotic execution time. On the other hand, Abboud et al. [1] revealed that, for any positive constant ϵ, there exist no $O(n^{2-\epsilon})$-time algorithms for computing the LCS length, unless the strong exponential time hypothesis (SETH) [7, 8] is false. This immediately implies that, under assumption of SETH, neither an LCS can be found nor whether a common subsequence is an LCS can be determined in $O(n^{2-\epsilon})$ time. Problems of finding a conditional LCS have also been considered. The constrained LCS (CLCS) problem [9, 3] (also called the SEQ-IC-LCS problem [2]) and the restricted LCS (RLCS) problem [5] (also called the SEQ-EC-LCS problem [2]) are such problems. Given a common subsequence P as essentially "relevant" (resp. "irrelevant") to relationship between the two strings, the CLCS (RLCS) problem consists of finding an LCS that has (resp. does not have) P as a subsequence and was shown to be solvable in $O(n^3)$ time [3] (resp. [5, 2]). From definition, the CLCS found is maximal. In contrast, the RLCS found is not necessarily maximal and, unless maximal, the RLCS might not be very informative in certain applications because it is just obtained from some common subsequence, which has the "irrelevant" P perfectly as a subsequence, only by deleting a single character.

The reason why it takes at least an almost quadratic time to find an LCS or a conditional LCS as a pattern common to the two strings is due to condition that the pattern to be found should have a maximum length. Possibly for an analogous reason, the best asymptotic running time known for finding a shortest maximal common subsequence of two strings remains cubic [4]. The present article shows that, ignoring such conditions with respect to the length of a maximal common subsequence to be found, we can find a maximal common subsequence much faster, by proposing an $O(n \log\log n)$-time, $O(n)$-space algorithm. This algorithm can also be used to find a constrained maximal common subsequence, which hence has P as a subsequence, in the same asymptotic time and space, where P is an arbitrary common subsequence given as a "relevant" pattern. It is also shown that we can determine whether any given common subsequence, such as an RLCS, is maximal further faster, by proposing an $O(n)$-time algorithm.

This article is organized as follows. Section 2 defines notations and terminology used in this article. Section 3 proposes an $O(n \log\log n)$-time, $O(n)$-space algorithm that finds a maximal common subsequence of two strings of length $O(n)$. Section 4 modifies the above algorithm so as to output a maximal common subsequence having a given common subsequence as a subsequence in the same asymptotic time and space. Section 5 proposes an $O(n)$-time algorithm that determines if a given common subsequence is maximal. Section 6 concludes this article.

2 Preliminaries

For any sequences S and S', let $S \circ S'$ denote the concatenation of S followed by S'. Let ε denote the empty sequence. For any sequence S, let $|S|$ denote the length of S. For any index i with $1 \le i \le |S|$, 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 a sequence obtained from S by deleting elements at any position, i.e., $S[i_1] \circ S[i_2] \circ \cdots \circ S[i_l]$ for some indices $i_1, i_2, \dots, i_l$ with $0 \le l \le |S|$ and $1 \le i_1 < i_2 < \cdots < i_l \le |S|$. For any sequences S and S', we say that S contains S', if S' is a subsequence of S. For any indices i' and i with $0 \le i' \le i \le |S|$, let $S(i', i]$ denote the contiguous subsequence of S consisting of all elements at position between $i' + 1$ and i, i.e., $S[i' + 1] \circ S[i' + 2] \circ \cdots \circ S[i]$. Note that $S(i, i] = \varepsilon$. We call $S(i', i]$ a prefix (resp. suffix) of S, if $i' = 0$ (resp. $i = |S|$).

Let $\Sigma = \{c_1, c_2, \cdots, c_{|\Sigma|}\}$ be an alphabet set of $|\Sigma|$ characters, which are totally ordered. A string is a sequence of characters over Σ. For any strings X and Y, a common subsequence of X and Y is a subsequence of X that is also a subsequence of Y. We say that X and Y are disjoint, if they have no non-empty common subsequences. Let a common subsequence W of X and Y be maximal, if inserting any character into W can no longer yield a common subsequence of X and Y.

3 Algorithm for finding a maximal common subsequence

This section proposes an $O(n \log\log n)$-time algorithm that outputs, for any strings X and Y of length $O(n)$ with $|\Sigma| = O(n)$ given as input, a maximal common subsequence of X and Y.

For technical reasons, we assume without loss of generality that $X[1] = Y[1] = c_1$, $X[|X|] = Y[|Y|] = c_{|\Sigma|}$, which will work as sentinels, and neither c_1 nor $c_{|\Sigma|}$ appears in $X(1, |X|-1]$ and also in $Y(1, |Y|-1]$. Note that $W[1] = c_1$, $W[|W|] = c_{|\Sigma|}$, and $W(1, |W|-1]$ is a maximal common subsequence of $X(1, |X|-1]$ and $Y(1, |Y|-1]$ for any maximal common subsequence W of X and Y.

We also assume that the array $\mathcal{I}$ (resp. $\mathcal{J}$) of arrays I_c (resp. J_c) for all characters c in Σ is available, where I_c (resp. J_c) is an appropriate data structure supporting queries of the following index, indicating the nearest occurrence of a specific character c in X (resp. Y) from a specific position i (resp. j).

▶ **Definition 1.** For any character c in Σ and any index i with $0 \le i \le |X|$, let $I_c^{\prec}(i)$ (resp. $I_c^{\succ}(i)$) denote the least (greatest) index such that c does not appear in $X(I_c^{\prec}(i), i]$ (resp. $X(i, I_c^{\succ}(i)]$). Define index $J_c^{\prec}(j)$ (resp. $J_c^{\succ}(j)$) analogously with respect to Y.

In what follows, we adopt as data structure I_c (resp. J_c) the y-fast trie [11] maintaining all indices i (resp. j) with $X[i] = c$ (resp. $Y[j] = c$), because array $\mathcal{I}$ (resp. $\mathcal{J}$) is constructible in $O(n \log\log n)$ time and $O(n)$ space and supports $O(\log\log n)$-time queries of any index introduced above. However, in implementation for practical use, if n is not very large, then due to hidden constant factors in big-O notation, adopting as I_c (resp. J_c) the array consisting of the same indices as the y-fast trie in ascending order, supporting $O(\log n)$-time queries based on a binary search of the array, might be more suitable. Furthermore, if $|\Sigma|$ is a small constant, then we can adopt as I_c (resp. J_c) the table of indices $I_c^{\prec}(i)$ and $I_c^{\succ}(i)$ (resp. $J_c^{\prec}(i)$ and $J_c^{\succ}(j)$) for all indices i (resp. j), which supports $O(1)$-time queries.

We design the proposed algorithm based on the following property of a common subsequence W, which is naturally derived from the fact that W is not maximal if and only if inserting some character between some prefix and the remaining suffix of W still yields a common subsequence of X and Y.

▶ **Lemma 2.** *For any common subsequence W of X and Y, W is maximal if and only if X_k and Y_k are disjoint for any index k with $0 \le k \le |W|$, where X_k (resp. Y_k) is the remaining substring obtained from X (resp. Y) by deleting both the shortest prefix containing $W(0, k]$ and the shortest suffix containing $W(k, |W|]$.*

Proof. The lemma follows from the fact that, for any index k with $0 \le k \le |W|$ and any character c in Σ, $W(0, k] \circ c \circ W(k, |W|]$ is a common subsequence of X and Y if and only if c appears in both X_k and Y_k. ◀

The proposed algorithm solves the problem using string variable W, which is initially set to $c_1 \circ c_{|\Sigma|}$ and is eventually updated to a maximal common subsequence of X and Y. For any index k with $0 \le k \le |W|$, let X_k and Y_k be the substrings in Lemma 2. The algorithm updates W by iteratively replacing it by $W(0, k] \circ c \circ W(k, |W|]$, where k is the least index such that X_k and Y_k are not disjoint and c is a certain character appearing both in X_k and Y_k, until X_k and Y_k become disjoint for all indices k with $0 \le k \le |W|$. Note that the resulting string W is a maximal common subsequence of X and Y due to Lemma 2. The algorithm adopts as c the character that appears both in the shortest possible suffix of X_k or Y_k and the entire string of the other. As shown later, this choice is crucial to executing the algorithm in $O(n \log\log n)$ time.

In order to execute the above, the algorithm maintains a sequence variable, $\hat{W} = (i_1, j_1) \circ (i_2, j_2) \circ \cdots \circ (i_{|W|}, j_{|W|})$, consisting of $|W|$ index pairs so that $X(i_k, i_{k+1}]$ and $Y(j_k, j_{k+1}]$ are respectively certain prefixes of X_k and Y_k such that they are disjoint if and only if X_k and Y_k are disjoint. The character to be inserted at position between $W(0, k]$ and $W(k, |W|]$ is searched for by iteratively updating $\hat{W}$ by replacing (i_{k+1}, j_{k+1}) by $(i_{k+1} - 1, j_{k+1} - 1)$. If i_{k+1} becomes i_k or j_{k+1} becomes j_k, then, since X_k and Y_k are disjoint, the algorithm updates $\hat{W}$ by replacing $\hat{W}[k+1]$ by (i', j') and then updates k to $k+1$, where i' (resp. j') is the index such that $X(0, i']$ (resp. $Y(0, j']$) is the shortest prefix of X (resp. Y) containing $W(0, k+1]$, i.e. $i' = I_{W[k+1]}^{\succ}(i_k) + 1$ (resp. $j' = J_{W[k+1]}^{\succ}(j_k) + 1$). Otherwise,

Algorithm 1: Algorithm findMCS

```
 1:  $W \leftarrow c_1 \circ c_{|\Sigma|}$;
 2:  $\hat{W} \leftarrow (1,1) \circ (|X|-1, |Y|-1)$;
 3:  $k \leftarrow 1$;
 4:  while $k < |W|$,
 5:      $(i', j') \leftarrow \hat{W}[k]$;
 6:      $(i, j) \leftarrow \hat{W}[k+1]$;
 7:      while $i' < i$, $j' < j$, $J^{\prec}_{X[i]}(j) \leq j'$, and $I^{\prec}_{Y[j]}(i) \leq i'$,
 8:          $\hat{W}[k+1] \leftarrow (i-1, j-1)$;
 9:          $(i, j) \leftarrow \hat{W}[k+1]$;
10:      if $i = i'$ or $j = j'$, then
11:          $\hat{W}[k+1] \leftarrow (I^{\succ}_{W[k+1]}(i') + 1, J^{\succ}_{W[k+1]}(j') + 1)$;
12:          $k \leftarrow k + 1$,
13:      otherwise, if $J^{\prec}_{X[i]}(j) > j'$, then
14:          $W \leftarrow W(0,k] \circ X[i] \circ W(k, |W|]$;
15:          $\hat{W} \leftarrow \hat{W}(0,k] \circ (i-1, J^{\prec}_{X[i]}(j) - 1) \circ \hat{W}(k, |\hat{W}|]$,
16:      otherwise,
17:          $W \leftarrow W(0,k] \circ Y[j] \circ W(k, |W|]$;
18:          $\hat{W} \leftarrow \hat{W}(0,k] \circ (I^{\prec}_{Y[j]}(i) - 1, j - 1) \circ \hat{W}(k, |\hat{W}|]$;
19:  output $W$.
```

if $X[i_{k+1}]$ appears in $Y(j_k, j_{k+1}]$ (i.e., if $J^{\prec}_{X[i_{k+1}]}(j_{k+1}) > j_k$), then the algorithm updates W to $W(0,k] \circ X[i_{k+1}] \circ W(k, |W|]$ and also updates $\hat{W}$ to $\hat{W}(0,k] \circ (i,j) \circ \hat{W}(k, |\hat{W}|]$, where i (resp. j) is the index such that $X(i, |X|]$ (resp. $Y(j, |Y|]$) is the shortest suffix of X (resp. Y) containing $X[i_{k+1}] \circ W(k, |W|]$, i.e., $i = i_{k+1} - 1$ (resp. $j = J^{\prec}_{X[i_{k+1}]}(j_{k+1}) - 1$). Otherwise, since $Y[j_{k+1}]$ appears in $X(i_k, i_{k+1}]$, the algorithm updates W and $\hat{W}$ in a symmetric manner with respect to $Y[j_{k+1}]$.

A pseudocode of the proposed algorithm is given as Algorithm findMCS in Algorithm 1, where we assume that, by an $O(n \log\log n)$-time preprocessing, arrays $\mathcal{I}$ and $\mathcal{J}$ are available as data structures supporting $O(\log\log n)$-time queries of any of indices $I^{\prec}_c(i)$, $I^{\succ}_c(i)$, $J^{\prec}_c(j)$, and $J^{\succ}_c(j)$. In this pseudocode, variables i', j', i, and j are respectively used to represent indices i_k, j_k, i_{k+1}, and j_{k+1}, where $(i_k, j_k) = \hat{W}[k]$ and $(i_{k+1}, j_{k+1}) = \hat{W}[k+1]$. A concrete example of how this algorithm works is presented in Figure 1.

As mentioned earlier, the following condition holds at any execution of line 7 of this algorithm and also at the last execution of line 4.

▶ **Definition 3.** For any string W, any index pair sequence $\hat{W} = (i_1, j_1) \circ (i_2, j_2) \circ \cdots \circ (i_{|W|}, j_{|W|})$, and any index k, let $C(W, \hat{W}, k)$ denote the condition that

- W is a common subsequence of X and Y containing $c_1 \circ c_{|\Sigma|}$,
- $1 \leq k \leq |W|$,
- for any index k' with $1 \leq k' \leq k-1$, $X(i^{\dashv}_{k'}, i^{\vdash}_{k'+1}]$ and $Y(j^{\dashv}_{k'}, j^{\vdash}_{k'+1}]$ are disjoint,
- $(i_k, j_k) = (i^{\dashv}_k, j^{\dashv}_k)$, and
- for any index k' with $k \leq k' \leq |W|-1$,
  - $i_k \leq i_{k'+1} \leq i^{\vdash}_{k'+1}$,
  - $j_k \leq j_{k'+1} \leq j^{\vdash}_{k'+1}$,
  - $X(i_{k'+1}, i^{\vdash}_{k'+1}]$ and $Y(j_k, j^{\vdash}_{k'+1}]$ are disjoint, and

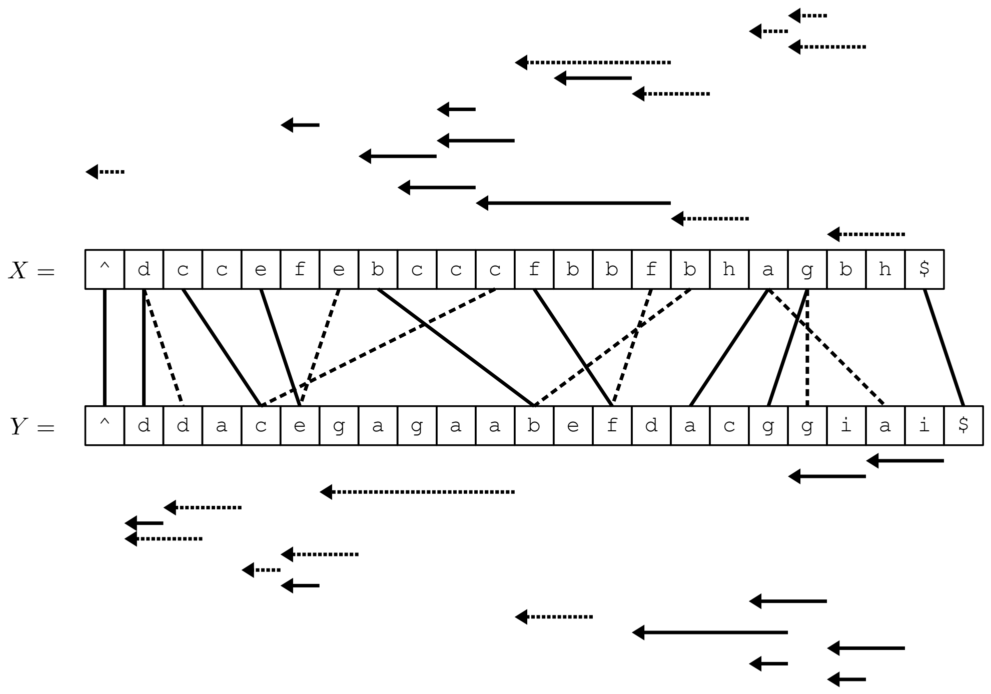
X =
^ d c c e f e b c c c f b b f b h a g b h $
Y =
^ d d a c e g a g a a b e f d a c g g i a i $

Figure 1 A maximal common subsequence $W =$ ^dcebfag$ of $X =$ ^dccefebcccfbbfbhagbh$ and $Y =$ ^ddacegagaabefdacggiai$ with $c_1 =$ ^ and $c_{|\Sigma|} = \$$, which is output by Algorithm findMCS. Lines 5 through 18 of the algorithm are executed fifteen times and for each number t with $1 \le t \le 15$, the tth most inner pair of arrows (one solid and the other dotted, which are of the same length) indicates which index pairs (i, j) are considered by line 7 throughout the tth iteration of lines 5 through 18, where the dotted arrow is chosen so as to show that the sum of the length of all dotted arrows is at most $2(|X| + |Y|)$. Each dashed line between $X[i]$ and $Y[j]$ indicates that $\hat{W}$ is replaced by $\hat{W}(0, k] \circ (i-1, j-1) \circ \hat{W}(k, |\hat{W}|]$ by either line 15 or line 18. Each solid line between $X[i']$ and $Y[j']$, other than the leftmost one, indicates that $\hat{W}[k+1]$ is set to index pair (i', j') by line 11.

- $X(i_k, i^{\vdash}_{k'+1}]$ and $Y(j_{k'+1}, j^{\vdash}_{k'+1}]$ are disjoint,

where, for any index k' with $0 \le k' \le |W|$, $i^{\dashv}_{k'}$ (resp. $j^{\dashv}_{k'}$) is the least index such that $X(0, i^{\dashv}_{k'}]$ (resp. $Y(0, j^{\dashv}_{k'}]$) contains $W(0, k']$ and $i^{\vdash}_{k'+1}$ (resp. $j^{\vdash}_{k'+1}$) is the greatest index such that $X(i^{\vdash}_{k'+1}, |X|]$ (resp. $Y(j^{\vdash}_{k'+1}, |Y|]$) contains $W(k', |W|]$.

▶ **Lemma 4.** *Condition $C(W, \hat{W}, k)$ holds at any execution of line 7 of Algorithm* findMCS *and also at the last execution of line 4.*

Proof. The lemma is proven by induction. Since $C(c_1 \circ c_{|\Sigma|}, (1,1) \circ (|X|-1, |Y|-1), 1)$ holds at the first execution of line 7, assume that $C(W, \hat{W}, k)$ holds at an arbitrary execution of line 7. Let $(i', j') = \hat{W}[k]$ and let $(i, j) = \hat{W}[k+1]$. If $i' < i$, $j' < j$, $J^{\prec}_{X[i]}(j) \le j'$, and $I^{\prec}_{Y[j]}(i) \le i'$, then $C(W, \hat{W}(0, k] \circ (i-1, j-1) \circ \hat{W}(k+1, |W|], k)$ holds, because $i' \le i-1$, $j' \le j-1$, $X[i]$ does not appear in $Y(j', j]$, and $Y[j]$ does not appear in $X(i', i]$. If $i = i'$ or $j = j'$, then $C(W, \hat{W}(0, k] \circ (I^{\succ}_{W[k+1]}(i') + 1, J^{\succ}_{W[k+1]}(j') + 1) \circ \hat{W}(k+1, |W|], k+1)$ holds, because $X(i^{\dashv}_k, i^{\vdash}_{k+1}]$ and $Y(j^{\dashv}_k, j^{\vdash}_{k+1}]$ are disjoint. If $i' < i$, $j' < j$, and $J^{\prec}_{X[i]}(j) > j'$, then $C(W(0, k] \circ X[i] \circ W(k, |W|], \hat{W}(0, k] \circ (i-1, J^{\prec}_{X[i]}(j) - 1) \circ \hat{W}(k, |W|], k)$ holds, because $X(i-1, |X|]$ (resp. $Y(J^{\prec}_{X[i]}(j) - 1, |Y|]$) is the shortest suffix of X (resp. Y)

that contains $X[i] \circ W(k, |W|]$. Analogously, if $i' < i$, $j' < j$, and $I^{\prec}_{Y[j]}(i) > i'$, then $C(W(0,k] \circ Y[j] \circ W(k,|W|], \hat{W}(0,k] \circ (I^{\prec}_{Y[j]}(i)-1, j-1) \circ \hat{W}(k,|W|], k)$ holds. ◀

The following simple lemma plays a key role in estimating execution time of the algorithm. This lemma claims, for example, that the situation where any solid line other than the leftmost and rightmost ones in Figure 1 shares at least one of endpoints with a unique dotted line is inevitable.

▶ **Lemma 5.** *At least one of $I^{\succ}_{W[k+1]}(i') = i^{\vdash}_{k+1}$ or $J^{\succ}_{W[k+1]}(j') = j^{\vdash}_{k+1}$ holds at any execution of line 11 in Algorithm* findMCS, *where $i^{\vdash}_{k+1}$ and $j^{\vdash}_{k+1}$ are the indices in Definition 3.*

Proof. Since $X(i', i^{\vdash}_{k+1}]$ and $Y(j', j^{\vdash}_{k+1}]$ are disjoint due to Lemma 4, $W[k+1]$ does not appear in at least one of $X(i', i^{\vdash}_{k+1}]$ or $Y(j', j^{\vdash}_{k+1}]$. ◀

▶ **Theorem 6.** *For any strings X and Y of length $O(n)$ with $|\Sigma| = O(n)$, Algorithm* findMCS *outputs a maximal common subsequence of X and Y in $O(n \log\log n)$ time and $O(n)$ space.*

Proof. Since $C(W, \hat{W}, |W|)$ holds at the last execution of line 4 of the algorithm due to Lemma 4, it follows from Lemma 2 that W output by the algorithm is a maximal common subsequence of X and Y.

Execution time of the algorithm is estimated as follows. Let V be the eventual string W output by line 19. For any index k with $0 \le k \le |V|$, let $g^{\dashv}_k$ (resp. $h^{\dashv}_k$) denote the least index such that $X(0, g^{\dashv}_k]$ (resp. $Y(0, h^{\dashv}_k]$) contains $V(0,k]$. Let k be an arbitrary index with $1 \le k \le |V|$ and consider W and $\hat{W}$ just before execution of line 11. Let $i^{\dashv}_k$, $j^{\dashv}_k$, $i^{\vdash}_{k+1}$, and $j^{\vdash}_{k+1}$ be the indices in Definition 3. Let $(i', j') = \hat{W}[k]$ and let $(i,j) = \hat{W}[k+1]$. Note that $i' = i^{\dashv}_k$ and $j' = j^{\dashv}_k$ due to Lemma 4. Since $i = i'$ or $j = j'$, $\hat{W}[k+1]$ is obtained from $(i^{\vdash}_{k+1}, j^{\vdash}_{k+1})$ by executing either line 15 or line 18 and then executing line 8 iteratively $\min(i^{\vdash}_{k+1} - i^{\dashv}_k, j^{\vdash}_{k+1} - j^{\dashv}_k)$ times. This implies that execution time of the algorithm is $O(\sum_{k=1}^{|V|-1} \min(i^{\vdash}_{k+1} - i^{\dashv}_k, j^{\vdash}_{k+1} - j^{\dashv}_k) \log n)$. Since $W(0,k] = V(0,k]$, both $i^{\dashv}_k = g^{\dashv}_k$ and $j^{\dashv}_k = h^{\dashv}_k$ hold. Similarly, since $W(0,k+1] = V(0,k+1]$, both $I^{\succ}_{W[k+1]}(i') + 1 = g^{\dashv}_{k+1}$ and $J^{\succ}_{W[k+1]}(j') + 1 = h^{\dashv}_{k+1}$ hold. Therefore, from Lemma 5, $i^{\vdash}_{k+1} + 1 = g^{\dashv}_{k+1}$ or $j^{\vdash}_{k+1} + 1 = h^{\dashv}_{k+1}$ holds and hence we have that $\min(i^{\vdash}_{k+1} - i^{\dashv}_k, j^{\vdash}_{k+1} - j^{\dashv}_k) \le \max(g^{\dashv}_{k+1} - g^{\dashv}_k, h^{\dashv}_{k+1} - h^{\dashv}_k)$. Since $g^{\dashv}_1 = 1$, $h^{\dashv}_1 = 1$, $g^{\dashv}_{|V|} = |X|$, and $h^{\dashv}_{|V|} = |Y|$, $\sum_{k=1}^{|V|-1} \max(g^{\dashv}_{k+1} - g^{\dashv}_k, h^{\dashv}_{k+1} - h^{\dashv}_k) \le |X| + |Y| = O(n)$. Thus, $\sum_{k=1}^{|V|-1} \min(i^{\vdash}_{k+1} - i^{\dashv}_k, j^{\vdash}_{k+1} - j^{\dashv}_k) = O(n)$, implying that the algorithm outputs V in $O(n \log\log n)$ time.

The algorithm uses variables W, $\hat{W}$, k, i', j', i, and j, together with data structures $\mathcal{I}$ and $\mathcal{J}$, which all require $O(n)$ space. ◀

4 Algorithm for finding a constrained maximal common subsequence

This section modifies Algorithm findMCS so as to output, for any common subsequence P of X and Y given as an additional input string, a maximal common subsequence of X and Y that contains P in $O(n \log\log n)$ time and $O(n)$ space, where we assume the same condition of X and Y as in Section 3 and also assume that $P[1] = c_1$, $P[|P|] = c_{|\Sigma|}$ and neither c_1 nor $c_{|\Sigma|}$ appears in $P(1, |P|-1]$. Note that $W[1] = c_1$, $W[|W|] = c_{|\Sigma|}$, and $W(1, |W|-1]$ is a maximal common subsequence of $X(1, |X|-1]$ and $Y(1, |Y|-1]$ containing $P(1, |P|-1]$ for any maximal common subsequence W of X and Y containing P.

The only difference of the modified algorithm from the original algorithm is to initialize W to P, instead of $c_1 \circ c_{|\Sigma|}$, and $\hat{W}$ to a certain index pair sequence $\hat{P}$ satisfying $C(P, \hat{P}, 1)$,

Algorithm 2: Algorithm findCMCS

```
1:  $W \leftarrow P$;
2:  $\hat{W} \leftarrow \varepsilon$;
3:  $i \leftarrow |X| - 1$; $j \leftarrow |Y| - 1$;
4:  for each index $k$ from $|P| - 1$ down to 1,
5:      while $X[i+1] \neq P[k+1]$,
6:          $i \leftarrow i - 1$;
7:      while $Y[j+1] \neq P[k+1]$,
8:          $j \leftarrow j - 1$;
9:      $\hat{W} \leftarrow (i, j) \circ \hat{W}$;
10:     $i \leftarrow i - 1$; $j \leftarrow j - 1$;
11: $\hat{W} \leftarrow (1, 1) \circ \hat{W}$;
12: $k \leftarrow 1$;
13: do the same as lines 4 through 19 of Algorithm findMCS.
```

instead of $(1,1) \circ (|X|-1, |Y|-1)$. Since lines 4 through 19 of the original algorithm delete no characters from W, the modified algorithm eventually outputs a maximal common subsequence of X and Y that contains P in $O(n \log\log n)$ time after initialization of $(W, \hat{W}, 1)$ to $(P, \hat{P}, 1)$. For any index k with $1 \leq k \leq |P|$, let $i^{\vdash}_{k+1}$ (resp. $j^{\vdash}_{k+1}$) be the greatest index such that $X(i^{\vdash}_{k+1}, |X|]$ (resp. $Y(j^{\vdash}_{k+1}, |Y|]$) contains $P(k, |P|]$. Then, Definition 3 immediately suggests that $\hat{P}$ can be set to $(1,1) \circ (i^{\vdash}_2, j^{\vdash}_2) \circ (i^{\vdash}_3, j^{\vdash}_3) \circ \cdots \circ (i^{\vdash}_{|P|}, j^{\vdash}_{|P|})$. Thus, we have Algorithm findCMCS presented in Algorithm 2 as an $O(n \log\log n)$-time algorithm for finding a maximal common subsequence of X and Y containing P.

▶ **Theorem 7.** *For any strings X and Y of length $O(n)$ with $|\Sigma| = O(n)$ and any common subsequence P of X and Y, Algorithm* findCMCS *outputs a maximal common subsequence of X and Y containing P in $O(n \log\log n)$ time and $O(n)$ space.*

Proof. It is easy to verify by induction that, for any index k with $1 \leq k \leq |P| - 1$, $X(i, |X|]$ (resp. $Y(j, |Y|]$) at execution of line 9 of the algorithm are the shortest suffix of X (resp. Y) that contains $P(k, |P|]$. Therefore, W, $\hat{W}$, and k just after execution of line 12 satisfy $C(W, \hat{W}, k)$. Since lines 1 through 12 are executed in $O(n)$ time, the theorem can be proven in a way similar to the proof of Theorem 6. ◀

5 Algorithm for determining if a common subsequence is maximal

This section proposes an $O(n)$-time algorithm that determines, for any strings X and Y of length $O(n)$ with $|\Sigma| = O(n)$ and any common subsequence W of X and Y given as input, whether W is maximal or not.

The proposed algorithm is based on Lemma 2. Using an array of $|\Sigma|$ bits, each being used to indicate if a distinct character in Σ appears in Y_k, we can determine if X_k and Y_k are disjoint in $O(|X_k| + |Y_k|)$ time for any index k with $0 \leq k \leq |W|$, where X_k and Y_k are the substrings of X and Y in Lemma 2, respectively. However, this naive approach provides only an $O(n^2)$-time algorithm, because both $|X_k|$ and $|Y_k|$ can be $\Theta(n)$ for all indices k and $|W|$ can also be $\Theta(n)$. In order to reduce this execution time to $O(n)$, the algorithm exploits the fact that if X_{k-1} and Y_{k-1} are disjoint, then the prefix $X^{\lhd}_k$ of X_k overlapping X_{k-1} and the prefix $Y^{\lhd}_k$ of Y_k overlapping Y_{k-1} are also disjoint; otherwise, W is not maximal due to Lemma 2, where $X_{-1} = X(0,0]$ and $Y_{-1} = Y(0,0]$. From this fact, if X_{k-1} and Y_{k-1} are

Algorithm 3: Algorithm determineIfMCS

```
1:  $i^{\dashv}_0 \leftarrow 0$; $j^{\dashv}_0 \leftarrow 0$; $i^{\vdash}_{|W|+1} \leftarrow |X|$; $j^{\vdash}_{|W|+1} \leftarrow |Y|$;
2:  $k \leftarrow 1$;
3:  for each index $i$ from 1 to $|X|$,
4:      if $k \leq |W|$ and $X[i] = W[k]$, then
5:          $i^{\dashv}_k \leftarrow i$;
6:          $k \leftarrow k + 1$;
7:  $k \leftarrow |W| - 1$;
8:  for each index $i$ from $|X|$ down to 1,
9:      if $k \geq 1$ and $X[i] = W[k+1]$, then
10:         $i^{\vdash}_{k+1} \leftarrow i - 1$;
11:         $k \leftarrow k - 1$;
12: $k \leftarrow 1$;
13: for each index $j$ from 1 to $|Y|$,
14:     if $k \leq |W|$ and $Y[j] = W[k]$, then
15:         $j^{\dashv}_k \leftarrow j$;
16:         $k \leftarrow k + 1$;
17: $k \leftarrow |W| - 1$;
18: for each index $j$ from $|Y|$ down to 1,
19:     if $k \geq 1$ and $Y[j] = W[k+1]$, then
20:         $j^{\vdash}_{k+1} \leftarrow j - 1$;
21:         $k \leftarrow k - 1$;
22: for each character $c$ in $\Sigma$,
23:     $i_c \leftarrow 0$;
24:     $j_c \leftarrow 0$;
25: for each index $k$ from 0 to $|W|$,
26:     for each index $i$ from $i^{\vdash}_k + 1$ to $i^{\vdash}_{k+1}$, where $i^{\vdash}_0 = 0$,
27:         $i_{X[i]} \leftarrow i$;
28:         if $j_{X[i]} > j^{\dashv}_k$, then
29:             output "not maximal" and halt;
30:     for each index $j$ from $j^{\vdash}_k + 1$ to $j^{\vdash}_{k+1}$, where $j^{\vdash}_0 = 0$,
31:         $j_{Y[j]} \leftarrow j$;
32:         if $i_{Y[j]} > i^{\dashv}_k$, then
33:             output "not maximal" and halt;
34: output "maximal".
```

disjoint, then whether X_k and Y_k are disjoint can be determined only by checking if $X^{\triangleright}_k$ and $Y^{\triangleleft}_k$ are disjoint as well as checking if X_k and $Y^{\triangleright}_k$ are disjoint, where $X^{\triangleright}_k$ (resp. $Y^{\triangleright}_k$) are the remaining suffix of X_k (resp. Y_k) after deleting prefix $X^{\triangleleft}_k$ (resp. $Y^{\triangleleft}_k$). Note however that, as long as using the array of $|\Sigma|$ bits, it still takes $O(|X_k| + |Y_k|)$ time to determine if X_k and Y_k are disjoint. The algorithm reduces this execution time to $O(|X^{\triangleright}_k| + |Y^{\triangleright}_k|)$ by using, instead of the bit array, a pair of arrays of $|\Sigma|$ indices. Each index in one (resp. the other) of the arrays in the pair is used to represent the last position at which a distinct character in Σ appears in the prefix of Y (resp. X) having $Y^{\triangleleft}_k$ (resp. X_k) as a suffix. This index array allows the algorithm to determine if any character in $X^{\triangleright}_k$ (resp. $Y^{\triangleright}_k$) appears in $Y^{\triangleleft}_k$ (resp. X_k) in $O(1)$ time. Furthermore, since the prefix of Y (resp. X) having $Y^{\triangleleft}_k$ (resp. X_k) as a suffix is the concatenation of the prefix of Y (resp. X) having $Y_{k-1}^{\triangleleft}$ (resp. X_{k-1}) as a suffix followed by $Y_{k-1}^{\triangleright}$ (resp. $X_k^{\triangleright}$), for each k, this index array can be updated appropriately in $O(|Y_{k-1}^{\triangleright}|)$ (resp. $O(|X_k^{\triangleright}|)$) time.

We show that Algorithm determineIfMCS presented in Figure 3 works as the proposed algorithm.

▶ **Theorem 8.** *For any strings X and Y of length $O(n)$ with $|\Sigma| = O(n)$ and any common subsequence W of X and Y, Algorithm* determineIfMCS *outputs message "not maximal", if W is not a maximal common subsequence of X and Y, or outputs message "maximal", otherwise, in $O(n)$ time.*

Proof. For any index k with $0 \leq k \leq |W|$, let X_k and Y_k be the strings in Lemma 2 and let indices $i_k^{\dashv}$, $i_{k+1}^{\vdash}$, $j_k^{\dashv}$, and $j_{k+1}^{\vdash}$ be such that $X_k = X(i_k^{\dashv}, i_{k+1}^{\vdash}]$ and $Y_k = Y(i_k^{\dashv}, i_{k+1}^{\vdash}]$. Furthermore, let $X_k^{\triangleleft} = X(i_k^{\dashv}, \max(i_k^{\vdash}, i_k^{\dashv})]$ and let $X_k^{\triangleright} = X(\max(i_k^{\vdash}, i_k^{\dashv}), i_{k+1}^{\vdash}]$, where $i_0^{\vdash} = 0$. Similarly, let $Y_k^{\triangleleft} = Y(j_k^{\dashv}, \max(j_k^{\vdash}, j_k^{\dashv})]$ and let $Y_k^{\triangleright} = Y(\max(j_k^{\vdash}, j_k^{\dashv}), j_{k+1}^{\vdash}]$, where $j_0^{\vdash} = 0$. The algorithm uses index variable i_c (resp. j_c) for any character c in Σ. Let I (resp. J) denote the array consisting of variables i_c (resp. j_c) for all characters c in Σ. For any index k with $-1 \leq k \leq |W|$, let $C_I(k)$ (resp. $C_J(k)$) denote the condition that, for any character c in Σ, $X(i_c, i_{k+1}^{\vdash}]$ (resp. $Y(j_c, j_{k+1}^{\vdash}]$) is the longest suffix of $X(0, i_{k+1}^{\vdash}]$ (resp. $Y(0, j_{k+1}^{\vdash}]$) in which c does not appear.

After computing indices $i_k^{\dashv}$, $j_k^{\dashv}$, $i_k^{\vdash}$, and $j_k^{\vdash}$ for all indices k with $0 \leq k \leq |W|$ by lines 1 through 21 of the algorithm, lines 22 through 24 initialize variables i_c and j_c so that $C_I(-1)$ and $C_J(-1)$ hold. Then, for any index k from 0 to $|W|$, lines 25 through 33 check if X_k and Y_k are disjoint as follows. Since either $k = 0$ or X_{k-1} and Y_{k-1} are disjoint, $X_k^{\triangleleft}$ and $Y_k^{\triangleleft}$ are disjoint. Therefore, it suffices to check if $X_k^{\triangleright}$ and $Y_k^{\triangleleft}$ are disjoint and check if X_k and $Y_k^{\triangleright}$ are disjoint. Lines 27 through 29 update I so as to satisfy $C_I(k)$ by iteratively executing line 27 and also check if $X_k^{\triangleright}$ and $Y_k^{\triangleleft}$ are disjoint by iteratively executing line 28 using array J satisfying $C_J(k-1)$. If $X_k^{\triangleright}$ and $Y_k^{\triangleleft}$ are not disjoint, then, since $j_{X[i]} > j_k^{\dashv}$ holds for some index i with $i_k^{\vdash} + 1 \leq i \leq i_{k+1}^{\vdash}$ due to $C_J(k-1)$, line 29 outputs message "not maximal" and terminates the algorithm; otherwise, line 29 is never executed also due to $C_J(k-1)$ and hence lines 30 through 33 are executed. Lines 30 through 33 update array J so as to satisfy $C_J(k)$ and check if X_k and $Y_k^{\triangleright}$ are disjoint using array I satisfying $C_I(k)$ in a similar manner. Thus, the algorithm works correctly.

It is easy to verify that the algorithm runs in $O(n)$ time. ◀

6 Conclusion

The present article proposed an $O(n \log \log n)$-time, $O(n)$-space algorithm that finds a maximal common subsequence of two $O(n)$-length strings over an alphabet set of $O(n)$ characters, which are totally ordered, where n is an arbitrary positive integer and a common subsequence is maximal, if inserting any character into it can no longer yields a common subsequence. It is also shown that, without increasing asymptotic time and space complexities, this algorithm can be used to find a constrained maximal common subsequence, which contains a common subsequence given arbitrarily as a "relevant" pattern, after an appropriate initialization of some variables. Furthermore, an $O(n)$-time algorithm that determines if a given common subsequence is maximal was also proposed.

There remain some questions to be solved, which are related to the problems considered in the present article. Our algorithms run much faster than those proposed so far (and also all possible algorithms under SETH) for the LCS-related problems corresponding to ours. One reason for this difference is that any common subsequence is certainly a subsequence of some maximal common subsequence but is not necessarily a subsequence of any LCS. This fact naturally poses a question whether we can find a restricted maximal common subsequence, which does not contain a common subsequence given as an “irrelevant” pattern, in $O(n \log \log n)$ time and $O(n)$ space, because some restricted non-maximal common subsequences are not necessarily subsequences of any restricted maximal common subsequence. The gap between asymptotic execution time of the proposed algorithms for finding a maximal common subsequence and for determining if a common subsequence given is maximal immediately poses another natural question whether we can find a maximal common subsequence in $O(n)$ time.

References

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

2 Yi-Ching Chen and Kun-Mao Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, 21(3):383–392, 2011.

3 Francis YL Chin, Alfredo De Santis, Anna Lisa Ferrara, NL Ho, and SK Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4):175–179, 2004.

4 Campbell B Fraser, Robert W Irving, and Martin Middendorf. Maximal common subsequences and minimal common supersequences. *Information and Computation*, 124(2):145–153, 1996.

5 Zvi Gotthilf, Danny Hermelin, Gad M Landau, and Moshe Lewenstein. Restricted LCS. In *International Symposium on String Processing and Information Retrieval*, pages 250–257. Springer, 2010.

6 Daniel S Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

7 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

8 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

9 Yin-Te Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.

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

11 Dan E Willard. Log-logarithmic worst-case range queries are possible in space Θ(N). *Information Processing Letters*, 17(2):81–84, 1983.