Theoretical Computer Science 793 (2019) 132–139

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Maximal common subsequence algorithms ☆

Yoshifumi Sakai

Graduate School of Agricultural Science, Tohoku University, 468-1, Aza-Aoba, Aramaki, Aoba-ku, Sendai 980-0845, Japan

ARTICLE INFO

Article history:
Received 21 November 2018
Received in revised form 3 June 2019
Accepted 6 June 2019
Available online 2 July 2019
Communicated by R. Giancarlo

Keywords:
Algorithms
String comparison
Longest common subsequence

ABSTRACT

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

1. Introduction

A subsequence of a string is obtained from the string by deleting any number of characters at any position. A common subsequence of two strings is a subsequence of one string that is also a subsequence of the other. Searching for certain kinds of common subsequences is important in some applications because we can think of a common subsequence as a pattern common to the strings. Let a common subsequence be maximal if inserting a character into it no longer yields a common subsequence. The present article considers the problem of determining if a given common subsequence is maximal, and also considers the problem of finding a maximal common subsequence (an MCS) that contains the given common subsequence as a subsequence. These problems may arise when we search for informative patterns common to two strings. For example, consider the situation where we want to find a common subsequence that is essentially different from a given common subsequence P. Here, a common subsequence W is essentially different from P if W is not a subsequence of any common subsequence that is similar to P under some similarity criterion. Suppose that we have a common subsequence W that is not similar to P as a candidate. If W is maximal, then we can immediately conclude that W is essentially different from P. In contrast, for the case in which W is not maximal, an arbitrary MCS Z that has W as a subsequence tells us the following. If Z is not similar to P, then Z is essentially different from P; otherwise, W is not essentially different from P (and hence we can shift our focus to another possible candidate, if any).

A longest common subsequence (an LCS) is a common subsequence that has the greatest possible length. Any (non-conditional) LCS is hence maximal. Since the length of an arbitrary LCS of two strings can be used to measure the similarity between the strings, the problem of finding an LCS is one of the classic problems in computer science and has been widely investigated. It is well known that the dynamic programming algorithm of Wagner and Fisher [15] finds an LCS of two strings in $O(n^2)$ time and $O(n^2)$ space. Here, we use n to denote the sum of the length of the strings. The divide-and-conquer technique of Hirschberg [10] reduces this required space to $O(n)$ without increasing the asymptotic execution

☆ A preliminary version [13] of this paper was presented at the 29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018) at Qingdao, China, in July 2018.

E-mail address: yoshifumi.sakai.c7@tohoku.ac.jp.

https://doi.org/10.1016/j.tcs.2019.06.020
0304-3975/

time. On the other hand, it was revealed [1,3] that, for any positive constant ϵ, there exist no $O(n^{2-\epsilon})$-time algorithms for computing the LCS length if the strong exponential time hypothesis (SETH) [11,12] is true. This immediately implies that, under the assumption of the 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 studied. The constrained LCS (CLCS) problem [14,5,4] and the restricted LCS (RLCS) problem [8,4] are such problems. The CLCS (resp. RLCS) problem consists of finding an LCS that has (resp. does not have) P as a subsequence. Here, P is a common subsequence possibly given as a pattern that is relevant (resp. irrelevant) to the relationship between the two strings in some sense. This problem was shown to be solvable in $O(n^3)$ time [5] (resp. [8,4]). A variant of the CLCS problem, which asks to find an arbitrary LCS that has P as a substring (i.e., a contiguous subsequence), is also considered [4,6] and was shown to be solvable in $O(n^2)$ time [6]. Note that the conditional LCS found by solving the RLCS problem or the variant of the CLCS problem is not necessarily maximal. In particular, if we want to find a common subsequence W that is essentially different from an irrelevant pattern P in the sense that W is not a subsequence of any common subsequence that has P as a subsequence, then the RLCS found is not necessarily satisfactory. This is because, unless the RLCS is maximal, we can insert a character into this RLCS to obtain an LCS that has P as a subsequence.

The reason why it takes at least almost quadratic time to find an LCS or a conditional LCS as a pattern common to the two strings is due to the condition that the pattern to be found should be of the maximum length. Possibly for an analogous reason, the best asymptotic running time for finding an arbitrary shortest MCS remains cubic [7]. The present article shows that ignoring such conditions on the length of an MCS to be found, we can find an MCS much faster by proposing an $O(n\sqrt{\log n/\log\log n})$-time, $O(n)$-space algorithm. This algorithm can also be used to find an MCS that has a given common subsequence P as a subsequence in the same asymptotic time and space. It is also shown that we can determine whether a 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 $O(n\sqrt{\log n/\log\log n})$-time, $O(n)$-space algorithm that finds an MCS of two strings. Section 4 modifies the above algorithm so as to output an MCS that has 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 T, let $S \circ T$ denote the concatenation of S followed by T. Let ε denote the empty sequence. For any sequence S, let $|S|$ denote the length of 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 a sequence obtained from S by deleting an arbitrary number of elements at any position, i.e., $S[i_1] \circ S[i_2] \circ \cdots \circ S[i_l]$ for some indices $i_1, i_2, \ldots, i_l$ with $0 \leq l \leq |S|$ and $1 \leq i_1 < i_2 < \cdots < i_l \leq |S|$. For any sequences S and T, we say that S contains T, if T is a subsequence of S. For any indices g and i with $1 \leq g \leq i+1 \leq |S|+1$, let $S[g,i]$ denote the contiguous subsequence of S consisting of all elements at position between g and i, i.e., $S[g] \circ S[g+1] \circ \cdots \circ S[i]$. For simplicity, we use $S[i+1,i]$ to denote ε. We call $S[g,i]$ a prefix (resp. suffix) of S, if $g = 1$ (resp. $i = |S|$).

Let Σ be an alphabet set of a finite number of ordered characters. We call any sequence of characters over Σ a string. We say that two strings are disjoint if they have no non-empty common subsequences, i.e., if any character appearing in one string does not appear in the other. A common subsequence of two strings is a subsequence of one string that is also a subsequence of the other. Let a common subsequence of two strings be maximal, if inserting any character into the subsequence no longer yields a common subsequence of the two strings. We call any common subsequence of two strings that is maximal a maximal common subsequence (an MCS) of the strings.

The algorithms we will propose in the subsequent sections take as input two strings X and Y, possibly together with a common subsequence of X and Y. We use n to denote the sum of the length of X and Y, i.e., $n = |X| + |Y|$. Furthermore, we assume that any character in Σ appears in at least one of X or Y. When Σ does not satisfy this condition, we can remove as preprocessing all the irrelevant characters each appearing in neither X nor Y from Σ, for example, in $O(n)$ time using radix sorting, if Σ is of size polynomial in n, or otherwise in $O(n \log\log n)$ time and $O(n)$ space using the integer sorting of Han [9]. However, for simplicity, we do not consider computational resources required to execute any such preprocessing.

3. Algorithm for finding an MCS

This section proposes an $O(n\sqrt{\log n/\log\log n})$-time, $O(n)$-space algorithm that outputs an MCS of X and Y.

We design the proposed algorithm using the following two lemmas. The first lemma presents an immediate recurrence. Based on this recurrence, the algorithm finds an MCS by determining each of its elements from the last one to the first successively. The second lemma allows the algorithm to do so time-efficiently. We present these lemmas using the notations introduced below.

Definition 1. For any non-empty common subsequence W of X and Y, let g_W (resp. h_W) denote the index such that $X[1, g_W]$ (resp. $Y[1, h_W]$) is the shortest prefix of X (resp. Y) that contains W. Furthermore, let i_W (resp. j_W) denote the index such that $X[i_W, |X|]$ (resp. $Y[j_W, |Y|]$) is the shortest suffix of X (resp. Y) that contains $W[|W|]$.

Lemma 1. *For any non-empty common subsequence W of X and Y, if $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ are disjoint, then the concatenation of any MCS of $X[1,i_W-1]$ and $Y[1,j_W-1]$ that contains $W[1,|W|-1]$ followed by $W[|W|]$ is an MCS of X and Y.*

Proof. Suppose that $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ are disjoint and let W' be the concatenation in the lemma. Since W' contains W, $g_W \le g_{W'}$ and $h_W \le h_{W'}$. This implies that $X[g_{W'}+1,|X|]$ and $Y[h_{W'}+1,|Y|]$ are also disjoint. Therefore, appending any character to W' does not yield a common subsequence of X and Y. On the other hand, due to $W'[|W'|]=W[|W|]$, $i_{W'}=i_W$ and $j_{W'}=j_W$. This implies that $W'[1,|W'|-1]$ is an MCS of $X[1,i_{W'}-1]$ and $Y[1,j_{W'}-1]$. Therefore, inserting any character into W' at any position before $W'[|W'|]$ does not yield a common subsequence of X and Y. □

Lemma 2. *For any non-empty common subsequence W of X and Y, if $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ are disjoint, then $i_W=g_W$ or $j_W=h_W$.*

Proof. Otherwise, $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ would have $W[|W|]$ as a common character, a contradiction. □

For example, if $X=$ `eeeaeecebebcd`, $Y=$ `fafffaffbddcfa`, and $W=$ `ac`, then $X[1,g_W]=X[1,7]=$ `eeeaeec`, $Y[1,h_W]=Y[1,12]=$ `fafffaffbddc`, $X[i_W,|X|]=X[12,13]=$ `cd`, and $Y[j_W,|Y|]=Y[12,14]=$ `cfa`. Since $X[g_W+1,|X|]=X[8,13]=$ `ebebcd` and $Y[h_W+1,|Y|]=Y[13,14]=$ `fa` are disjoint, Lemma 1 guarantees that `abc` is an MCS of X and Y. This is because `ab` is an MCS of $X[1,i_W-1]=X[1,11]=$ `eeeaeecebeb` and $Y[1,j_W-1]=Y[1,11]=$ `fafffaffbdd` that contains $W[1,|W|-1]=$ `a`. As claimed by Lemma 2, h_W and j_W in this concrete example represent the same index, which is 12, because g_W $(=7)$ and i_W $(=12)$ do not represent the same index.

An outline of the proposed algorithm is as follows. The algorithm finds an MCS of X and Y as the output by applying Lemma 1 recursively. That is, the algorithm obtains the last element of the output by finding a common subsequence W of X and Y such that $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ are disjoint and adopting the last element of W. The algorithm finds such W by repeatedly appending a common character of $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ to W until $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ become disjoint, where W is initially set to the empty string. The common character to be appended is searched for by checking if each character in any of $X[g_W+1,|X|]$ or $Y[h_W+1,|Y|]$ appears in the other. After that, the algorithm regards $X[1,i_W-1]$, $Y[1,j_W-1]$, and $W[1,|W|-1]$ as X, Y, and the initial W, respectively, to obtain the second last element of the output in the same manner. To obtain each of the other elements in the output from the third last one to the first successively, the algorithm iterates the same process until X and Y become disjoint. From initialization of W in each iteration, Lemma 1 guarantees by induction that the algorithm works correctly.

Due to the outline presented above, the execution time of the algorithm depends on the number of characters checked to search for a common character of $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$. Our key observation to save this number is that check of any character can be skipped if the character is already checked in a previous iteration with the same W. This is because $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ in the present iteration are respectively prefixes of those in the previous iteration. For each distinct W, the algorithm checks each of characters $X[g_W+1]$, $Y[h_W+1]$, $X[g_W+2]$, $Y[h_W+2]$, $X[g_W+3]$, and so on alternately in this order until the last element of any of $X[g_W+1,|X|]$ or $Y[h_W+1,|Y|]$ is checked. This process is interrupted whenever a common character of $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ is found, and restarted with shorter X and Y just after the algorithm adopts the common character as an element of the output. The number of characters checked for this W is hence at most $2\min\{|X|-g_W,|Y|-h_W\}$, where X and Y are the eventual ones such that $X[g_W+1,|X|]$ and $Y[h_W+1,|Y|]$ are disjoint. As shown later, Lemma 2 guarantees that the sum of this value over all distinct strings W is bounded from above by $O(n)$.

As an example, Fig. 1 demonstrates how the proposed algorithm finds string `abc` as an MCS of $X=$ `eeeaeecebebcd` and $Y=$ `fafffaffbddcfa`. Execution of the algorithm for this concrete example of X and Y consists of seven stages, Stages (a) through (g) performed in this order as follows. Stage (a) with $|X|=13$, $|Y|=14$, $W=\varepsilon$, $g_W=0$, and $h_W=0$ finds a common character $\mathtt{a}=X[4]=Y[2]$ of $X[1,13]=$ `eeeaeecebebcd` and $Y[1,14]=$ `fafffaffbddcfa` by checking each of elements $X[1]$, $Y[1]$, $X[2]$, and $Y[2]$ in this order to pass $W=$ `a`, $g_W=4$, and $h_W=2$ to Stage (b). Stage (b) finds a common character $\mathtt{a}=X[7]=Y[12]$ of $X[5,13]=$ `eecebebcd` and $Y[3,14]=$ `fffaffbddcfa` by checking each of elements $X[5]$, $Y[3]$, $X[6]$, $Y[4]$, and $X[7]$ in this order to pass $W=$ `ac`, $g_W=7$, and $h_W=12$ to Stage (c). Stage (c) confirms that $X[8,13]=$ `ebebcd` and $Y[13,14]=$ `fa` are disjoint by checking each of elements $X[8]$, $Y[13]$, $X[9]$, and $Y[14]$ in this order to obtain character `c` as the last element of the output and passes $|X|=11$, $|Y|=11$, $W=$ `a`, $g_W=4$, and $h_W=2$ to Stage (d). Stage (d) finds a common character $\mathtt{b}=X[9]=Y[9]$ of $X[5,11]=$ `eecebeb` and $Y[3,11]=$ `fffaffbdd` by checking each of elements $Y[5]$, $X[8]$, $Y[6]$, and $X[9]$ in this order to pass $W=$ `ab`, $g_W=9$, and $h_W=9$ to Stage (e). Note that checking elements in $X[5,7]$ and elements in $Y[3,4]$ is skipped due to the previous stage with the same W, which is Stage (b). Stage (e) confirms that $X[10,11]=$ `eb` and $Y[10,11]=$ `dd` are disjoint by checking each of elements $X[10]$, $Y[10]$, and $X[11]$ in this order to obtain character `b` as the second last element of the output and passes $|X|=10$, $|Y|=8$, $W=$ `a`, $g_W=4$, and $h_W=2$ to Stage (f). Stage (f) confirms that $X[5,10]=$ `eecebe` and $Y[3,8]=$ `fffaff` are disjoint by checking each of elements $Y[7]$ and $X[10]$ in this order to obtain character `a` as the third last element of the output and passes $|X|=3$, $|Y|=5$, $W=\varepsilon$, $g_W=0$, and $h_W=0$ to Stage (g). Note that checking elements in $X[5,9]$ and elements in $Y[3,6]$ is skipped due to the previous stages with the same W, which are Stages (b) and (d). Finally, Stage (g) confirms that $X[1,3]=$ `eee` and $Y[1,5]=$ `fafff` are disjoint by checking $X[3]$ to conclude that the entire string of the output has

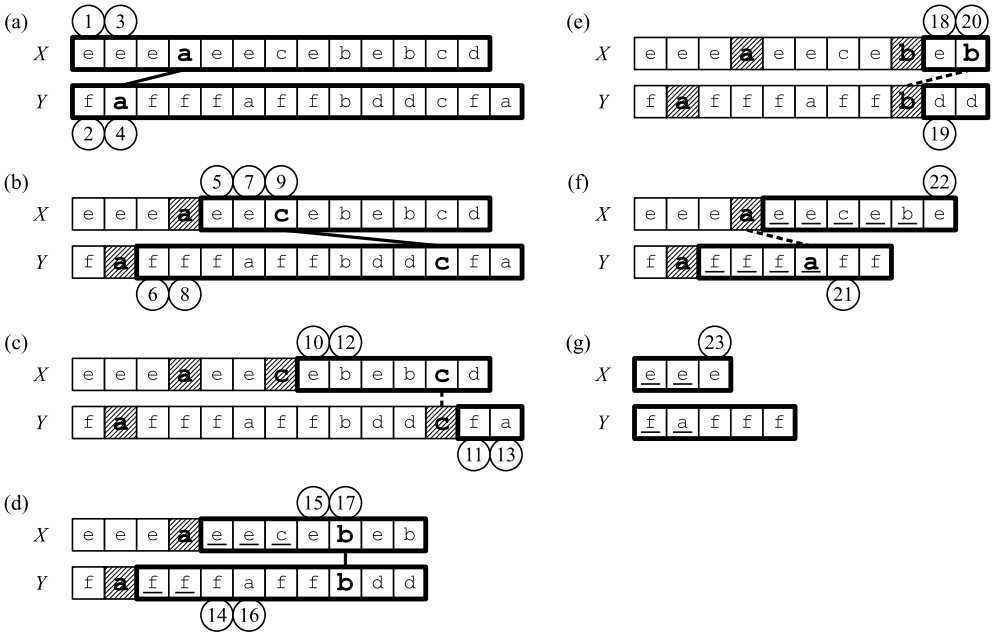

Fig. 1. How the proposed algorithm finds an MCS of a concrete example of X and Y is presented, where we consider $X = \texttt{eeeaeecebebcd}$ and $Y = \texttt{fafffaffbddcfa}$. The algorithm executes Stages (a) through (g) in this order. The regions of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ in each stage are indicated by thick boxes. The shaded elements in X (resp. Y) compose W, i.e., strings W in Stages (a) through (g) are ε, a, ac, a, ab, a, and ε, respectively. The algorithm checks if each element of X or Y indicated by an encircled number is a common character of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ in ascending order with respect to the number. Each element of $X[g_W + 1, |X|]$ or $Y[h_W + 1, |Y|]$ already checked in a previous stage with the same W is indicated by an underline. Each solid line, appearing in Stage (a), (b), or (d), indicates the common character of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ to be appended to W. Each dashed line, appearing in Stage (c), (e), or (f), connects $X[i_W]$ and $Y[j_W]$.

already been obtained. Note that checking elements in $X[1, 2]$ and elements in $Y[1, 2]$ is skipped due to the previous stage with the same W, which is Stage (a).

To design methods to determine indices g_W, h_W, i_W, and j_W in Lemma 1 as well as methods to check if each character in one of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ appears in the other, it is useful to introduce the following notations.

Definition 2. For any index g (resp. i) with $0 \le g \le |X|$ (resp. $1 \le i \le |X| + 1$) and any character c in Σ, let $\mathsf{next}_X(g, c)$ (resp. $\mathsf{prev}_X(i, c)$) denote the greatest (resp. least) index such that c does not appear in $X[g + 1, \mathsf{next}_X(g, c) - 1]$ (resp. $X[\mathsf{prev}_X(i, c) + 1, i - 1]$). Define indices $\mathsf{next}_Y(h, c)$ and $\mathsf{prev}_Y(j, c)$ analogously with respect to Y.

The proposed algorithm utilizes the data structure of Beame and Fich [2], which is $O(n)$-time constructible and supports $O(\sqrt{\log n / \log\log n})$-time queries of index $\mathsf{next}_X(g, c)$ (resp. $\mathsf{next}_Y(h, c)$) defined above. (If the size of Σ is constant, then we can adopt the lookup table, which is constructible in $O(n)$ time and supports $O(1)$-time queries, to let the proposed algorithm run in $O(n)$ time.)

A pseudocode of the proposed algorithm is presented as Algorithm findMCS in Fig. 2. In this implementation, the algorithm uses variable Z to store the suffix of the output obtained so far. Variable i (resp. j) is used to represent the index such that $X[i, |X|]$ (resp. $Y[j, |Y|]$) is the shortest suffix of X (resp. Y) that contains Z, so that the algorithm can regard $X[1, i - 1]$ (resp. $Y[1, j - 1]$) as X (resp. Y) to apply Lemma 1. Furthermore, stack S is maintained so that element (g, h, s) popped from S consists of indices g_W and h_W and the number s of elements each in $X[g + 1, i - 1]$ or $Y[h + 1, j - 1]$ that are already checked. Note that W is represented by S implicitly as $X[g_2] \circ X[g_3] \circ \cdots \circ X[g_{|S|}]$ $(= Y[h_2] \circ Y[h_3] \circ \cdots \circ Y[h_{|S|}])$, where the sequence of all elements in S from the bottom one to the top to be popped is $(0, 0, s_1) \circ (g_2, h_2, s_2) \circ (g_3, h_3, s_3) \circ \cdots \circ (g_{|S|}, h_{|S|}, s_{|S|})$. For example, such sequences of elements in S just before Stages (a) through (g) in Fig. 1 are $(0, 0, 0)$, $(0, 0, 4) \circ (4, 2, 0)$, $(0, 0, 4) \circ (4, 2, 5) \circ (7, 12, 0)$, $(0, 0, 4) \circ (4, 2, 5)$, $(0, 0, 4) \circ (4, 2, 9) \circ (9, 9, 0)$, $(0, 0, 4) \circ (4, 2, 9)$, and $(0, 0, 4)$, respectively.

Algorithm findMCS works as follows. After initialization of variables Z, i, and j and stack S by line 1, the algorithm executes lines 3 through 12 iteratively until S becomes empty. In each iteration, line 3 pops (g, h, s) from S to let the algorithm know that the possible element that should be checked at the present moment is $X[g']$, if s is even, or $Y[h']$, otherwise, where $g' = g + s/2 + 1$ and $h' = h + (s + 1)/2$. Let us focus only on the case where s is even, because the algorithm works analogously for the other case. If $g' < i$, then since $X[g']$ is an element of $X[g + 1, i - 1]$, the algorithm checks if character $X[g']$ appears in $Y[h + 1, j - 1]$ to maintain S correctly by executing lines 7 through 10. Line 7 pushes $(g, h, s + 1)$ into S because the last element of W has not yet been adopted as an element of the output. Lines 8 and 9

```
1: Z ← ε; i ← |X|; j ← |Y|; push (0, 0, 0) into the empty stack S;
2: while S is non-empty,
3:   pop (g, h, s) from S;
4:   if s is even (resp. odd), then
5:     g′ ← g + s/2 + 1 (resp. h′ ← h + (s + 1)/2);
6:     if g′ < i (resp. h′ < j), then
7:       push (g, h, s + 1) into S;
8:       h′ ← next_Y(h, X[g′]) (resp. g′ ← next_X(g, Y[h′]));
9:       if h′ ≤ j − 1 (resp. g′ ≤ i − 1), then
10:        push (g′, h′, 0) into S,
11:    otherwise, if g ≥ 1, then
12:      Z ← X[g] ∘ Z; i ← prev_X(i, X[g]); j ← prev_Y(j, X[g]);
13: output Z.
```

Fig. 2. Algorithm findMCS.

```
1: Execute line 1 of Algorithm findMCS;
2: g ← 0; h ← 0;
3: for each index k from 1 to |P|,
4:   g ← next_X(g, P[k]); h ← next_Y(h, P[k]);
5:   push (g, h, 0) into S;
6: execute lines 2 through 15 of Algorithm findMCS.
```

Fig. 3. Algorithm findCMCS.

check if character $X[g']$ appears in $Y[h+1, j-1]$. This is done by checking if $\mathsf{next}_Y(h, X[g']) \le j-1$. If the result of this check is positive, then line 10 pushes $(g', \mathsf{next}_Y(h, X[g']), 0)$ into S because character $X[g']$ is appended to W, and also because $\mathsf{next}_X(g, X[g']) = g'$. On the other hand, if $g' = i$ and $g \ge 1$, then since no character in $X[g+1, i-1]$ appears in $Y[h+1, j-1]$, implying that $X[g+1, i-1]$ and $Y[h+1, j-1]$ are disjoint, and also since W is non-empty, the algorithm adopts the last element of W as an element of the output. Since W has $X[g]$ as the last element, line 12 executes the above by prepending $X[g']$ to Z and updating i and j to $\mathsf{prev}_X(i, X[g])$ and $\mathsf{prev}_Y(j, X[g])$, respectively. Here, $\mathsf{prev}_X(i, X[g])$ and $\mathsf{prev}_Y(j, X[g])$ are determined by linear search. Note that line 12 pushes nothing into S because $W[1, |W|-1]$ is regarded as W in the succeeding iteration. If $g' = i$ and $g = 0$, then since $X[1, i-1]$ and $Y[1, j-1]$ are disjoint, implying that Z is an MCS of X and Y, the algorithm terminates the loop of lines 3 through 12 by doing nothing to leave S empty and outputs Z by line 13.

Now we have the following theorem.

Theorem 1. *Algorithm* findMCS *outputs an MCS of X and Y in $O(n\sqrt{\log n / \log\log n})$ time and $O(n)$ space.*

Proof. The algorithm correctly outputs an MCS of X and Y due to Lemma 1. Since the data structure supporting queries of any of $\mathsf{next}_X(g, c)$ or $\mathsf{next}_Y(h, c)$ uses $O(n)$ space and both variable Z and stack S have at most $O(n)$ elements, the algorithm requires $O(n)$ space. The execution time of the algorithm can be estimated as $O(n + t\sqrt{\log n / \log\log n})$, where t is the number of times that the algorithm executes line 8. We can show that $t = O(n)$ as follows. It is easy to verify that t is equal to the sum of s over all elements (g, h, s) popped from S by line 3 such that $g' = i$, if s is even, or $h' = j$, otherwise. Whenever any such element (g, h, s) is popped from S by line 3, $s \le 2\min\{i-g-1, j-h-1\}$. Since $X[g+1, i-1]$ and $Y[h+1, j-1]$ are disjoint for this (g, h, s), it follows from Lemma 2 that $\min\{i-g, j-h\} \le \max\{i - \mathsf{prev}_X(i, X[g]), j - \mathsf{prev}_Y(j, X[g])\}$ unless $g = 0$. Furthermore, the sum of $i - \mathsf{prev}_X(i, X[g])$ (resp. $j - \mathsf{prev}Y(j, X[g])$) over all such elements (g, h, s) with $g \ge 1$ plus $i - g - 1$ (resp. $j - h - 1$) for the only such element (g, h, s) with $g = 0$ is equal to $|X|$ (resp. $|Y|$). Thus we have that $t \le 2(|X| + |Y|) = O(n)$. □

4. Algorithm for finding a constrained MCS

This section modifies Algorithm findMCS so as to output an MCS of X and Y that contains P. Here, P is an arbitrary common subsequence of X and Y that is given as an additional input. This modification does not increase the asymptotic time and space.

The only difference between the modified algorithm and the original is that the modified one initializes W to P, instead of ε. This simple modification works because the output of the original algorithm contains W at any moment. A pseudocode of this modified algorithm is presented as Algorithm findCMCS in Fig. 3.

Theorem 2. *For any common subsequence P of X and Y, Algorithm* findCMCS *outputs an MCS of X and Y that contains P in $O(n\sqrt{\log n / \log\log n})$ time and $O(n)$ space.*

Proof. By induction, $X[1, g]$ (resp. $Y[1, h]$) is the shortest prefix of X (resp. Y) that contains $P[1, k]$ at any execution of line 4 in Algorithm findCMCS. Using linear search to determine any of $\mathsf{next}_X(g, P[k])$ or $\mathsf{next}_Y(h, P[k])$ in line 4, the algorithm executes lines 2 through 5 in $O(n)$ time. Therefore, we can prove the theorem in a way similar to the proof of Theorem 1. □

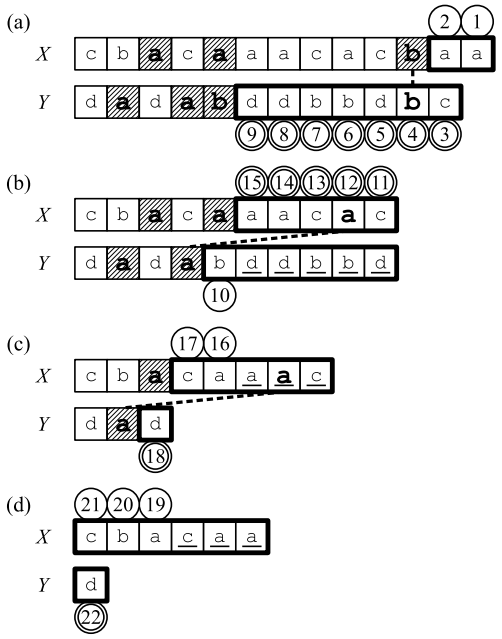

Fig. 4. How the proposed algorithm determines if a common string W of strings X and Y is maximal is presented, where we consider $X =$ cbacaaacacbaa, $Y =$ dadabddbbdbc, and $W =$ aab as a concrete example. The algorithm executes Stages (a) through (d) in this order. The regions of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ in each stage are indicated by thick boxes. The shaded elements in X (resp. Y) compose W, i.e., strings W in Stages (a) through (d) are aab, aa, a, and ε, respectively. In order to maintain indices $\mathsf{next}_X(g, c)$ and indices $\mathsf{next}_Y(h, c)$ for all characters c in Σ, the algorithm scans $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ according to the encircled numbers attached to their elements, where the suffixes already scanned by previous stages are indicated by underlined elements. The algorithm also checks whether each element with the number encircled doubly is a common character of $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ using the indices maintained. Each dashed line connects $X[i_W]$ and $Y[j_W]$.

5. Algorithm for determining if a common subsequence is maximal

This section proposes an $O(n)$-time algorithm that takes a common subsequence W of X and Y as an additional input and determines whether W is maximal or not. If W is an MCS of X and Y, the algorithm outputs TRUE; otherwise, it outputs FALSE.

The proposed algorithm is based on the following recurrence, which is obtained as an immediate corollary of Lemma 1.

Corollary 1. *For any non-empty common subsequence W of X and Y, if $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ are disjoint and $W[1, |W| - 1]$ is an MCS of $X[1, i_W - 1]$ and $Y[1, j_W - 1]$, then W is an MCS of X and Y.*

An outline of the algorithm is as follows. The algorithm determines if W is an MCS of X and Y by applying Corollary 1 recursively. That is, the algorithm obtains FALSE as the output and halts if $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ are not disjoint, or otherwise regards $X[1, i_W - 1]$, $Y[1, j_W - 1]$, $W[1, |W| - 1]$ as X, Y, and W, respectively, to do the same iteratively until X, Y, and W become no longer defined. In each iteration, in order to determine if $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ are disjoint, the algorithm maintains indices $\mathsf{next}_X(g, c)$ and indices $\mathsf{next}_Y(h, c)$ for all characters c in Σ. This is done in a straightforward way by scanning $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ both from the last element, where any part of this scan already done in previous iterations is skipped to guarantee that the algorithm runs in linear time. It follows from Lemma 2 that at least one of $X[g_W + 1, |X|]$ or $Y[h_W + 1, |Y|]$ is scanned from end to end with no skip. If $X[g_W + 1, |X|]$ is such one, then in order to determine if $X[g_W + 1, |X|]$ and $Y[h_W + 1, |Y|]$ are disjoint, the algorithm checks whether c appears in $Y[h_W + 1, |Y|]$ using $\mathsf{next}_Y(h, c)$ for each element c in $X[g_X + 1, |X|]$; otherwise, it checks whether c appears in $X[g_W + 1, |X|]$ using $\mathsf{next}_X(g, c)$ for each element c in $Y[h_W + 1, |Y|]$. This does not increase the asymptotic execution time of the algorithm.

Fig. 4 shows, as a concrete example, how the algorithm above determines if W is an MCS of X and Y for $X =$ cbacaaacacbaa, $Y =$ dadabddbbdbc, and $W =$ aab with $\Sigma = \{$a, b, c, d$\}$. Execution of the algorithm consists of Stages (a) through (d) in this order. Stage (a) with $|X| = 13$, $|Y| = 12$, $|W| = 3$, $g_W = 11$, and $h_W = 5$ scans $X[12, 13]$ to determine $\mathsf{next}_X(11,$ a$) = 12$, $\mathsf{next}_X(11,$ b$) > |X|$, $\mathsf{next}_X(11,$ c$) > |X|$, and $\mathsf{next}_X(11,$ d$) > |X|$. After that, it scans $Y[6, 12]$ to determine $\mathsf{next}_Y(5,$ a$) > |Y|$, $\mathsf{next}_Y(5,$ b$) = 8$, $\mathsf{next}_Y(5,$ c$) = 12$, and $\mathsf{next}_Y(5,$ d$) = 6$ and also to confirm that no character c in $Y[6, 12]$ appears in $X[12, 13]$ by checking if $\mathsf{next}_X(11, c) > |X|$. Finally, it passes $|X| = 11$, $|Y| = 11$, $|W| = 2$, $g_W = 5$, and $h_W = 4$ to Stage (b). Since suffix $Y[6, 10]$ of $Y[5, 10]$ has already been scanned by Stage (a), Stage (b) scans the remaining prefix $Y[5, 5]$ to determine $\mathsf{next}_Y(4,$ a$) > |Y|$, $\mathsf{next}_Y(4,$ b$) = 5$, $\mathsf{next}_Y(4,$ c$) > |Y|$, and $\mathsf{next}_Y(4,$ d$) = 6$. Then, it scans $X[6, 10]$ to determine

```
 1: g ← 0; h ← 0; push (0, 0) into the empty stack S;
 2: for each index k from 1 to |W|,
 3:   g ← next_X(g, W[k]); h ← next_Y(h, W[k]);
 4:   push (g, h) into S;
 5: let NEXT_X be the array of variables NEXT_X(c), each initialized to |X| + 1, for all characters c in Σ;
 6: let NEXT_Y be the array of variables NEXT_Y(c), each initialized to |Y| + 1, for all characters c in Σ;
 7: i ← |X| + 1; j ← |Y| + 1; g' ← i − 1, h' ← j − 1;
 8: while g' ≥ 1,
 9:   pop (g, h) from S;
10:   if g' = i − 1 (resp. g' < i − 1), then
11:     while h' > h (resp. g' > g),
12:       NEXT_Y(Y[h']) ← h' (resp. NEXT_X(X[g']) ← g');
13:       h' ← h' − 1 (resp. g' ← g' − 1);
14:     while g' > g (resp. h' > h),
15:       NEXT_X(X[g']) ← g' (resp. NEXT_Y(Y[h']) ← h');
16:       if NEXT_Y(X[g']) < j (resp. NEXT_X(Y[h']) < i), then
17:         output FALSE and halt,
18:       otherwise,
19:         g' ← g' − 1 (resp. h' ← h' − 1);
20:   i ← prev_Y(i, X[g]); j ← prev_Y(j, X[g]);
21:   g' ← min{g, i − 1}; h' ← min{h, j − 1};
22: output TRUE.
```

Fig. 5. Algorithm ifMCS.

$\mathsf{next}_X(5, \mathtt{a}) = 6$, $\mathsf{next}_X(5, \mathtt{b}) > |X|$, $\mathsf{next}_X(5, \mathtt{c}) = 8$, and $\mathsf{next}_X(5, \mathtt{d}) > |X|$ and also to confirm that no character c in $X[6, 10]$ appears in $Y[5, 10]$ by checking if $\mathsf{next}_Y(4, c) > |Y|$. Finally, it passes $|X| = 8$, $|Y| = 3$, $|W| = 1$, $g_W = 3$, and $h_W = 2$ to Stage (c). Note that, for example, $\mathsf{next}_Y(4, \mathtt{d}) = 6$ comes from $\mathsf{next}_Y(5, \mathtt{d}) = 6$, which is determined by Stage (a), and absence of character $\mathtt{d}$ in $Y[5, 5]$ scanned by Stage (b). Stage (c) passes $|X| = 6$, $|Y| = 1$, $|W| = 0$, $g_W = h_W = 0$ to Stage (d) in an analogous manner. Stage (d) scans prefix $X[1, 3]$ of $X[1, 6]$ to determine $\mathsf{next}_X(0, \mathtt{a}) = 3$, $\mathsf{next}_X(0, \mathtt{b}) = 2$, $\mathsf{next}_X(0, \mathtt{c}) = 1$, and $\mathsf{next}_X(0, \mathtt{d}) > |X|$. After that, it confirms that no character c in $Y[1, 1]$ appears in $X[1, 6]$ by checking if $\mathsf{next}_X(0, c) > |X|$ to eventually output TRUE.

A pseudocode of the algorithm is presented as Algorithm ifMCS in Fig. 5 works as the proposed algorithm. In this implementation, stack S is maintained so that element (g, h) popped from S consists of indices g_W and h_W. The algorithm initializes S by successively pushing the index pair (g, h) such that $X[1, g]$ (resp. $Y[1, h]$) is the shortest prefix of X (resp. Y) that contains $W[1, k]$ for each index k from 0 to $|W|$. This is done by lines 1 through 4 in $O(n)$ time, where line 3 obtains $\mathsf{next}_X(g, W[k])$ and $\mathsf{next}_Y(h, W[k])$ by linear search. For example, if X, Y, and W are in Fig. 4, then S is initialized by pushing $(0, 0)$, $(3, 2)$, $(5, 4)$, and $(11, 5)$ in this order. Variable i (resp. j) is maintained so that the algorithm can regard $X[1, i-1]$ (resp. $Y[1, j-1]$) as X (resp. Y) to apply Corollary 1. In order to maintain indices $\mathsf{next}_X(g, c)$ (resp. $\mathsf{next}_Y(h, c)$) for all characters c in Σ, the algorithm uses array NEXT_X (resp. NEXT_Y) consisting of variables $\mathrm{NEXT}_X(c)$ (resp. $\mathrm{NEXT}_Y(c)$) each representing an index for all characters c in Σ. For any index g' (resp. h') with $0 \le g' \le |X|$ (resp. $0 \le h' \le |Y|$), let $\mathrm{NEXT}_X^{(g')}$ (resp. $\mathrm{NEXT}_Y^{(h')}$) denote NEXT_X (resp. NEXT_Y) satisfying that $\mathsf{next}_X(g', c) = \min\{\mathrm{NEXT}_X(c), i\}$ (resp. $\mathsf{next}_Y(h', c) = \min\{\mathrm{NEXT}_Y(c), j\}$) for any character c in Σ. The algorithm uses variable g' (resp. h') to indicate that $\mathrm{NEXT}_X^{(g')}$ (resp. $\mathrm{NEXT}_Y^{(h')}$) is available. Note that $\mathrm{NEXT}_X^{(g')}$ (resp. $\mathrm{NEXT}_Y^{(h')}$) with $g' \ge 1$ (resp. $h' \ge 1$) can be updated to $\mathrm{NEXT}_X^{(g'-1)}$ (resp. $\mathrm{NEXT}_Y^{(h'-1)}$) only by setting $\mathrm{NEXT}_X(X[g'])$ (resp. $\mathrm{NEXT}_Y(Y[h'])$) to g' (resp. h'). The algorithm maintains NEXT_X (resp. NEXT_Y) based on this simple recurrence.

Algorithm ifMCS works as follows. After initializing stack S, arrays NEXT_X and NEXT_Y of variables, and variables i, j, g', and h' by lines 1 through 6, the algorithm repeats execution of lines 9 through 21 until either line 17 is executed or g' becomes zero. Due to lines 5 through 7, the algorithm has $\mathrm{NEXT}_X^{(g')}$ (resp. $\mathrm{NEXT}_Y^{(h')}$) as NEXT_X (resp. NEXT_Y) just before the first execution of lines 9 through 21. Each execution of lines 9 through 21 checks if $X[g+1, j-1]$ and $Y[h+1, j-1]$ are disjoint and outputs FALSE and halt, if the result of this check is negative, or pass $\mathrm{NEXT}_X^{(g)}$ and $\mathrm{NEXT}_Y^{(h)}$ to the next execution of lines 9 through 21, otherwise. More precisely, if $g' = i - 1$ (resp. $g' < i - 1$ and hence $h' = j - 1$ due to Lemma 2), then lines 11 through 13 scan $Y[h+1, h']$ (resp. $X[g+1, g']$) from the last element to update NEXT_Y (resp. NEXT_X) to $\mathrm{NEXT}_Y^{(h)}$ (resp. $\mathrm{NEXT}_X^{(g)}$). After that, lines 14 through 19 scan $X[g+1, i-1]$ (resp. $Y[h+1, j-1]$) from the last element to update NEXT_X (resp. NEXT_Y) to $\mathrm{NEXT}_X^{(g)}$ (resp. $\mathrm{NEXT}_Y^{(h)}$) and also to check if $X[g+1, i-1]$ and $Y[h+1, j-1]$ are disjoint using $\mathrm{NEXT}_Y^{(h)}$ (resp. $\mathrm{NEXT}_X^{(g)}$) obtained in the preceding execution of lines 11 through 13. If $X[g+1, i-1]$ and $Y[h+1, j-1]$ are disjoint, then g' (resp. h') becomes g (resp. h) eventually and lines 20 and 21 update i, j, g', and h' appropriately, where line 20 obtains $\mathsf{prev}_X(i, X[g])$ and $\mathsf{prev}_Y(j, X[g])$ by linear search. Finally, if $(0, 0)$ is popped from S as (g, h) by line 9 and lines 14 through 19 confirm that $X[g+1, i-1]$ and $Y[h+1, j-1]$ are disjoint, then g' becomes 0 and hence line 22 outputs TRUE.

Theorem 3. *For any common subsequence W of X and Y, Algorithm* ifMCS *outputs TRUE, if W is an MCS of X and Y, or outputs FALSE, otherwise, in $O(n)$ time.*

Proof. The algorithm correctly determines if W is an MCS of X and Y due to Corollary 1. Since values of g and h in lines 1 through 4 never decrease and also values of i, h, g', and h' in lines 8 through 21 never increase, the algorithm runs in $O(n)$ time. □

6. Conclusion

The present article proposed an $O(n\sqrt{\log n/\log\log n})$-time, $O(n)$-space algorithm that finds a maximal common subsequence (an MCS) of two strings over an alphabet set of at most n characters, which are totally ordered. Here, n is the sum of the length of the two strings and a common subsequence is maximal if inserting any character into it no longer yields a common subsequence. It was also shown that, without increasing asymptotic time and space complexities, this algorithm can be used to find a constrained MCS, which contains a common subsequence given arbitrarily, 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. These 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 the SETH assumption) for the LCS-related problems corresponding to ours. One reason for this difference is that any common subsequence is certainly a subsequence of some MCS but is not necessarily a subsequence of any LCS. This fact naturally poses a question whether we can find a restricted MCS, which does not contain a common subsequence given arbitrarily, in $O(n\sqrt{\log n/\log\log n})$ time and $O(n)$ space. This is because some restricted non-maximal common subsequences are not necessarily subsequences of any restricted MCS. The gap between asymptotic execution time of the proposed algorithms for finding an MCS and for determining if a common subsequence given is maximal immediately poses another natural question whether we can find an MCS in $O(n)$ time.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The author would like to thank the anonymous referees for their useful comments, especially for pointing out a mistake in the preliminary version [13] by drawing his attention to Ref. [2].

References

[1] A. Abboud, A. Backurs, V.V. Williams, Tight hardness results for lcs and other sequence similarity measures, in: Proc. of the 56th Annual Symposium on Foundations of Computer Science, 2015, pp. 59–78.
[2] P. Beame, F.E. Fich, Optimal bounds for the predecessor problem and related problems, J. Comput. Syst. Sci. 65 (2002) 38–72.
[3] K. Bringmann, M. Künnemann, Quadratic conditional lower bounds for string problems and dynamic time warping, in: Proc. of the 56th Annual Symposium on Foundations of Computer Science, 2015, pp. 79–97.
[4] Y.-C. Chen, K.-M. Chao, On the generalized constrained longest common subsequence problems, J. Comb. Optim. 21 (2011) 383–392.
[5] F.Y.L. Chin, A. De Santis, A. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Inf. Process. Lett. 90 (2004) 175–179.
[6] S. Deorowicz, Quadratic-time algorithm for a string constrained LCS problem, Inf. Process. Lett. 112 (2012) 423–426.
[7] C.B. Fraser, R.W. Irving, M. Middendorf, Maximal common subsequences and minimal common supersequences, Inf. Comput. 124 (1996) 145–153.
[8] Z. Gotthilf, D. Hermelin, G.M. Landau, M. Lewenstein, Restricted LCS, in: Proc. International Symposium on String Processing and Information Retrieval, 2010, pp. 250–257.
[9] Y. Han, Deterministic sorting in $O(n\log\log n)$ time and linear space, in: Proc. the 34th Annual ACM Symposium on Theory of Computing, 2002, pp. 602–608.
[10] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (1977) 664–675.
[11] R. Impagliazzo, R. Paturi, On the complexity of k-SAT, J. Comput. Syst. Sci. 62 (2001) 367–375.
[12] R. Impagliazzo, R. Paturi, F. Zane, Which problems have strongly exponential complexity?, J. Comput. Syst. Sci. 63 (2001) 512–530.
[13] Y. Sakai, Maximal common subsequence algorithms, in: Proc. the 29th Annual Symposium on Combinatorial Pattern Matching, 2018, pp. 1:1–1:10.
[14] Y.-T. Tsai, The constrained longest common subsequence problem, Inf. Process. Lett. 88 (2003) 173–176.
[15] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1974) 168–173.