LETTER *Special Section on Discrete Mathematics and Its Applications*

# A Fast Algorithm for Finding a Maximal Common Subsequence of Multiple Strings

**Miyuji HIROTA**[†a)], ***Nonmember and*** **Yoshifumi SAKAI**[†b)], ***Member***

**SUMMARY** For any $m$ strings of total length $n$, we propose an $O(mn \log n)$-time, $O(n)$-space algorithm that finds a maximal common subsequence of all the strings, in the sense that inserting any character in it no longer yields a common subsequence of them. Such a common subsequence could be treated as indicating a nontrivial common structure we could find in the strings since it is NP-hard to find any longest common subsequence of the strings.

***key words:*** *algorithms, longest common subsequence, multiple strings*

## 1. Introduction

Comparing strings and searching for the common structure they share is an essential task to unravel and exploit the regularity behind them. For example, in molecular biology, comparing strings that encode homologous proteins in various model organisms may be useful in understanding what constitutes the function of the protein. As for the structure that a string has, this article will consider a subsequence, which is obtained from the string by deleting any number of characters not necessarily contiguous. Thus, what we intend to find as the common structure of multiple strings is a certain nontrivial subsequence common to all of them that can be found in polynomial time.

The longest common subsequence (LCS) is one of the most studied nontrivial common subsequences as the most likely to make sense. However, finding an LCS of multiple strings was shown to be NP-hard [4], and the well-known algorithm based on dynamic programming [7] finds an LCS of multiple strings in time exponential in the number of strings. The maximal common subsequence (MCS), which is a generalization of LCS, is defined as a common subsequence that is no longer a common subsequence if any more characters are inserted. For two strings, a cubic-time algorithm to find the shortest MCS [3], a linearithmic-time algorithm to find an arbitrary MCS [5], and a polynomial-time delay algorithm to enumerate all MCSs [2] have been proposed. Although MCS does not have the same strict requirements for length as LCS, it is nontrivial in the sense that there is no other common subsequence that contains itself as a subsequence, and hence, for multiple strings, MCS would be a good alternative candidate to LCS.

In this article, we consider the problem of finding an arbitrary MCS of $m$ strings of total length $n$ that contains a given "pattern" string as a subsequence and propose an algorithm that solves this problem in $O(mn \log n)$ time and $O(n)$ space. The advantages of considering the problem of finding the conditional MCS for the pattern string include the following features.

- One can search for an MCS with no condition by setting the empty string as the pattern string.
- As in the case of the constrained LCS problem [6], if one knows a crucial pattern that should be included as a subsequence, it is possible to search for the desired MCS by setting this as the pattern string.
- The condition does not increase the time complexity of the proposed algorithm, unlike in the case of the constrained LCS problem [1].
- As a standard heuristic in practical use, one can attempt to refine the resulting MCS by applying an iterative method of repeatedly updating itself by deleting some characters chosen arbitrarily or randomly and setting it as the pattern string.

The algorithm we propose is a generalization of the existing algorithm for two strings [5], but with a certain modification. This modification makes the algorithm $O(m)$ times faster than the unmodified algorithm.

## 2. Preliminaries

For any sequences $S$ and $S'$, let $S \circ S'$ denote the concatenation of $S$ followed by $S'$. For any sequence $S$, let $|S|$ denote the number of elements in $S$, and for any index $i$ with $1 \le i \le |S|$, let $S[i]$ denote the $i$th 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 any number of elements at any position not necessarily contiguous, which is hence $S[i_1] \circ S[i_2] \circ \cdots \circ S[i_\ell]$ for some length $\ell$ with $0 \le \ell \le |S|$ and any $\ell$ indices $i_1, i_2, \ldots, i_\ell$ with $1 \le i_1 < i_2 < \cdots < i_\ell \le |S|$. We say that sequence $S$ *contains* sequence $S'$, if $S'$ is a subsequence of $S$. For any indices $i$ and $i'$ with $1 \le i \le i' \le |S|$, let $S[i : i']$ denote the contiguous subsequence $S[i] \circ S[i+1] \circ \cdots \circ S[i']$ of $S$. For convenience, we treat $S[i : i-1]$ with $1 \le i \le |S|+1$ as the empty sequence. For any index $i$ with $0 \le i \le |S|+1$, $S[1 : i]$ (resp. $S[i : |S|]$) is called a *prefix* (resp. *suffix*) of $S$ and is denoted by $S\langle i]$ (resp. $S[i\rangle$).

A *string* is a sequence whose elements are characters in

Manuscript received September 12, 2022.
Manuscript revised December 20, 2022.
Manuscript publicized March 6, 2023.

[†]The authors are with the Graduate School of Agricultural Science, Tohoku University, Sendai-shi, 980-0845 Japan.
a) E-mail: miyuji.hirota.p8@dc.tohoku.ac.jp
b) E-mail: yoshifumi.sakai.c7@tohoku.ac.jp (Corresponding author)

DOI: 10.1587/transfun.2022DML0002

an alphabet set $\Sigma$. A string $Z$ is called a *common subsequence* of a set $\mathcal{X}$ of strings, if all strings in $\mathcal{X}$ contain $Z$. We say that $\mathcal{X}$ *shares* $Z$, if $Z$ is a common subsequence of $\mathcal{X}$. A *longest common subsequence* (an *LCS*) of $\mathcal{X}$ is one of the longest strings that $\mathcal{X}$ shares. In contrast, a *maximal common subsequence* (an *MCS*) of $\mathcal{X}$ is defined as a string that $\mathcal{X}$ shares in which inserting any character no longer yields a string that $\mathcal{X}$ shares. Note that any LCS is an MCS, but an MCS is not necessarily an LCS.

**Example 1:** If $\mathcal{X} = \{\mathtt{CATCGCAT}, \mathtt{CGGAGTCC}, \mathtt{ATTCGAAT}\}$, then $\mathtt{ATC}$ is an MCS of $\mathcal{X}$, but $\mathtt{CAT}$ is not an MCS since inserting $\mathtt{G}$ in it yields $\mathtt{CGAT}$, whch is shared by $\mathcal{X}$.

Let $\mathcal{X} = \{X_1, X_2, \ldots, X_m\}$ be an arbitrary set of $m$ strings of total length $n$, and let $Y$ be an arbitrary string that $\mathcal{X}$ shares. We consider the problem of finding an arbitrary MCS of $\mathcal{X}$ that contains $Y$.

In what follows, for simplicity, we assume that $\Sigma$ consists of consecutive positive integers starting from 1 and $\mathcal{X}$ shares all characters in $\Sigma$. For any index $s$ with $1 \le s \le m$, any index $l$ with $0 \le l \le |X_s|$, and any character $c$ in $\Sigma$, let $next(s, l, c)$ denote the least index $i$ with $l < i \le |X_s|$ such that $X_s[i] = c$, if any, or $|X_s| + 1$, otherwise. Similarly, for any index $r$ with $1 \le r \le |X_s| + 1$, let $prev(s, r, c)$ denote the greatest index $i$ with $1 \le i < r$ such that $X_s[i] = c$, if any, or 0, otherwise. Any of $next(s, l, c)$ and $prev(s, r, c)$ is determined in $O(\log n)$ time by binary search on the array consisting of all indices $i$ with $1 \le i \le |X_s|$ such that $X_s[i] = c$ in ascending order. Since the number of characters in $\Sigma$ is at most $n/m$ due to the assumption, all such arrays are prepared in $O(n)$ time as preprocessing.

For any string $Z$ that $\mathcal{X}$ shares and any index $s$ with $1 \le s \le m$, let $l_{s,Z}$ (resp. $r_{s,Z}$) denote the index $i$ such that $X_s\langle i]$ (resp. $X_s[i\rangle$) is the shortest prefix (resp. suffix) of $X_s$ that contains $Z$. For any string $Z$ that $\mathcal{X}$ shares and any index $k$ with $0 \le k \le |Z|$, let $\mathcal{X}_{(Z,k)}$ denote the set of the strings $X_s[l_{s,Z\langle k]} + 1 : r_{s,Z[k+1\rangle} - 1]$, obtained from $X_s$ by deleting both the shortest prefix that contains $Z\langle k]$ and the shortest suffix that contains $Z[k+1\rangle$, for all indices $s$ with $1 \le s \le m$.

**Example 2:** Consider the same $\mathcal{X}$ as Example 1 and let $X_1 = \mathtt{CATCGCAT}$, $X_2 = \mathtt{CGGAGTCC}$, and $X_3 = \mathtt{ATTCGAAT}$, where $\mathtt{A}, \mathtt{T}, \mathtt{G}$, and $\mathtt{C}$ represent characters $1, 2, 3$, and $4$ in $\Sigma$, respectively. If $Z = \mathtt{CAT}$, then for any index $s$ with $1 \le s \le 3$, $l_{s,Z\langle 0]} = 0$, $r_{s,Z[4\rangle} = |X_s| + 1$ $(= 9)$, and for any index $k$ with $1 \le k \le |Z|$ $(= 3)$, $l_{s,Z\langle k]}$ (resp. $r_{s,Z[k\rangle}$) is indicated by the position of the $k$th character in $X_s$ that is overlined (resp. underlined) in the figure below.

$$X_1 = \overline{\mathtt{C}}\,\overline{\mathtt{A}}\,\overline{\mathtt{T}}\,\mathtt{C}\,\mathtt{G}\,\underline{\mathtt{C}}\,\underline{\mathtt{A}}\,\underline{\mathtt{T}}$$
$$X_2 = \underline{\overline{\mathtt{C}}}\,\mathtt{G}\,\mathtt{G}\,\underline{\overline{\mathtt{A}}}\,\mathtt{G}\,\underline{\overline{\mathtt{T}}}\,\mathtt{C}\,\mathtt{C}$$
$$X_3 = \mathtt{A}\,\mathtt{T}\,\mathtt{T}\,\underline{\overline{\mathtt{C}}}\,\mathtt{G}\,\underline{\overline{\mathtt{A}}}\,\underline{\mathtt{A}}\,\underline{\overline{\mathtt{T}}}$$

Hence, $\mathcal{X}_{(Z,0)} = \{\mathtt{CATCG}, \varepsilon, \mathtt{ATT}\}$, $\mathcal{X}_{(Z,1)} = \{\mathtt{ATCGC}, \mathtt{GG}, \mathtt{GA}\}$, $\mathcal{X}_{(Z,2)} = \{\mathtt{TCGCA}, \mathtt{G}, \mathtt{A}\}$, and $\mathcal{X}_{(Z,3)} = \{\mathtt{CGCAT}, \mathtt{CC}, \varepsilon\}$, where $\varepsilon$ denotes the empty string.

The following lemmas are straightforward generalizations of what appeared in [5] for two strings. The first lemma redefines the MCS of $\mathcal{X}$ in a convenient form while the second states a useful property about $\mathcal{X}_{(Z,k)}$.

**Lemma 1:** A string $Z$ that $\mathcal{X}$ shares is an MCS of $\mathcal{X}$ if and only if $\mathcal{X}_{(Z,k)}$ shares no character for any index $k$ with $0 \le k \le |Z|$.

**Proof.** Let $k$ be an arbitrary index with $0 \le k \le |Z|$ and let $c$ be an arbitrary character. If $\mathcal{X}_{(Z,k)}$ shares $c$, then $\mathcal{X}$ shares $Z\langle k] \circ c \circ Z[k+1\rangle$, implying that $Z$ is not an MCS of $\mathcal{X}$. On the other hand, if $\mathcal{X}$ shares $Z\langle k] \circ c \circ Z[k+1\rangle$, then $\mathcal{X}_{(Z,k)}$ shares $c$. □

**Lemma 2:** For any string $Z$ that $\mathcal{X}$ shares and any index $k$ with $1 \le k \le |Z|$, if $\mathcal{X}_{(Z,k)}$ shares no character, then there exists at least an index $t$ with $1 \le t \le m$ such that $l_{t,Z\langle k]} = r_{t,Z[k\rangle}$.

**Proof.** If no such index $t$ exists, then $\mathcal{X}_{(Z,k)}$ shares $Z[k]$, a contradiction. □

## 3. Algorithm

As mentioned in Sect. 1, the proposed algorithm is a generalization of the existing algorithm [5] for two strings, but with a certain modification, and this modification makes the proposed algorithm $O(m)$ times faster than the unmodified algorithm.

To begin with, we introduce the unmodified algorithm. This algorithm adopts the following framework to find an MCS of $\mathcal{X}$ that contains $Y$.

**Definition 1:** Define the *update procedure* as performing the following:

1. Initialize variable $Z$ to $Y$ and variable $k$ to $|Y|$,
2. repeatedly search for a character $c$ that $\mathcal{X}_{(Z,k)}$ shares to update $(Z, k)$ to $(Z\langle k] \circ c \circ Z[k+1\rangle, k+1)$, if any, or update $(Z, k)$ to $(Z, k-1)$, otherwise, until $k = -1$, and
3. output the resulting $Z$.

The update from $(Z, k)$ to $(Z\langle k] \circ c \circ Z[k+1\rangle, k+1)$ is called a *forward update* (with insertion of $c$). The update from $(Z, k)$ to $(Z, k-1)$ is called a *backward update*.

**Lemma 3:** The update procedure outputs an MCS of $\mathcal{X}$ that contains $Y$.

**Proof.** For any $(Z, k)$ in step 2 of the update procedure, let $C_{(Z,k)}$ denote the condition that $\mathcal{X}_{(Z,k')}$ shares no character for any index $k'$ with $k < k' \le |Z|$. Obviously $C_{(Y,|Y|)}$ holds for $(Y, |Y|)$, the initial $(Z, k)$. Consider any forward update from $(Z, k)$ with insertion of $c$ and let $Z_c = Z\langle k] \circ c \circ Z[k+1\rangle$. For any index $k'$ with $k < k' \le |Z|$ (i.e., $k+1 < k'+1 \le |Z_c|$) and any index $s$ with $1 \le s \le m$, $l_{s,Z_c\langle k'+1]} = l_{s,Z\langle k] \circ c \circ Z[k+1:k']} \ge l_{s,Z\langle k']}$ and $r_{s,Z_c[k'+2\rangle} = r_{s,Z[k'+1\rangle}$. Therefore, if $C_{(Z,k)}$ holds, then $C_{(Z_c,k+1)}$ also holds. On the other hand, for any backward update of $(Z, k)$, $\mathcal{X}_{(Z,k)}$ shares no character. Hence, if $C_{(Z,k)}$ holds, then $C_{(Z,k-1)}$ also holds. Consequently by induction,

$C_{(Z,-1)}$ holds for the resulting $Z$ output by step 3 of the procedure. Thus, the lemma follows from Lemma 1. □

To show how the unmodified algorithm searches for a character that $\mathcal{X}_{(Z,k)}$ shares, the following definition and lemma need to be introduced.

**Definition 2:** Let $Z' \prec_k Z$ mean that the update procedure performs a chain of updates starting with a forward update from $(Z', k)$ and ending with a backward update to $(Z, k)$ such that $k'' > k$ for any $(Z'', k'')$ appearing in the chain other than $(Z', k)$ and $(Z, k)$. If there exist more than one strings $Z_1, Z_2, \ldots, Z_\ell$ such that $Z' = Z_1 \prec_k Z_2 \prec_k \cdots \prec_k Z_\ell = Z$, then $Z'$ is called a *k-precursor* of $Z$.

**Lemma 4:** For any $k$-precursor $Z'$ of $Z$, $\mathcal{X}_{(Z,k)}$ consists of prefixes of distinct strings in $\mathcal{X}_{(Z',k)}$. Furthermore, if the update procedure performs a forward update with insertion of $c$ from $(Z', k)$, then $\mathcal{X}_{(Z,k)}$ does not share $c$.

**Proof.** Since $Z\langle k]$ has no change before and after any execution of step 2 of the update procedure, $l_{s,Z\langle k]} = l_{s,Z'\langle k]}$ holds for any index $s$ with $1 \le s \le m$. On the other hand, since $Z'[k+1\rangle$ is a suffix of $Z[k+1\rangle$, $r_{s,Z[k+1\rangle} \le r_{s,Z'[k+1\rangle}$. Thus, $\mathcal{X}_{(Z,k)}$ consists of prefixes of distinct strings in $\mathcal{X}_{(Z',k)}$. To prove the second half of the lemma, let $Z' \prec_k Z''$. Note that $Z''$ is either $Z$ or a $k$-precursor of $Z$. From definition of $Z' \prec_k Z''$, $Z''[k+1] = c$ and $\mathcal{X}_{(Z'',k+1)}$ shares no character. This implies that $\mathcal{X}_{(Z'',k)}$ does not share $c$. Furthermore, if $Z''$ is a $k$-precursor of $Z$, then $\mathcal{X}_{(Z,k)}$ does not share $c$ due to the first half of the lemma. □

The unmodified algorithm searches for a character that $\mathcal{X}_{(Z,k)}$ shares by repeatedly choosing an element $X_t[i]$ of some string $X_t[l_{t,Z\langle k]}+1 : r_{t,Z[k+1\rangle}-1]$ in $\mathcal{X}_{(Z,k)}$ and testing whether the chosen element is a character that $\mathcal{X}_{(Z,k)}$ shares. This process is performed until an element that passes the test is found or it is confirmed that some string in $\mathcal{X}_{(Z,k)}$ has no such element. Note that each test is done in $O(m \log n)$ time by checking if inequalities $next(s, l_{s,Z\langle k]}, X_t[i]) < r_{s,Z[k+1\rangle}$ hold for all indices $s$ with $1 \le s \le m$. The element $X_t[i]$ to be tested is always chosen so that any element of almost the same length prefix of any string in $\mathcal{X}_{(Z,k)}$ has already been tested whether $\mathcal{X}_{(Z',k)}$ shares it, where $Z'$ is either $Z$ itself or some $k$-precursor of $Z$. This is because Lemma 4 guarantees that $\mathcal{X}_{(Z,k)}$ never shares any of such elements, including even the element found as shared by $\mathcal{X}_{(Z',k)}$ for any $k$-precursor $Z'$ of $Z$. Therefore, for any $(Z, k)$ appearing in the execution of the update procedure such that $\mathcal{X}_{(Z,k)}$ shares no character, the number of tested elements is at most $m$ times the length of the shortest string in $\mathcal{X}_{(Z,k)}$. The reason why the algorithm does not test only elements of the shortest string in $\mathcal{X}_{(Z,k)}$ is that it is unclear which string the elements belong to in $\mathcal{X}_{(Z',k)}$ for the $k$-precursors $Z'$ of $Z$. Once it is confirmed that $\mathcal{X}_{(Z,k)}$ shares no character, no element of any string $X_t[l_{t,Z\langle k]}+1 : r_{t,Z[k+1\rangle}-1]$ with $l_{t,Z\langle k]} = r_{t,Z[k\rangle}$ in $\mathcal{X}_{(Z,k)}$ will be chosen to be tested thereafter. Lemma 2 guarantees that there exists such a string, which is not shorter than the shortest string in $\mathcal{X}_{(Z,k)}$. Thus, the total number of elements chosen to be tested throughout the execution of the algorithm is at most $mn$, and hence the algorithm runs in $O(m^2 n \log n)$ time.

By utilizing the following lemma, instead of Lemma 2, the proposed algorithm improves the execution time of the unmodified algorithm to $O(mn \log n)$.

**Lemma 5:** For any string $Z$ that $\mathcal{X}$ shares, any index $k$ with $1 \le k \le |Z|$, and any index $t$ with $1 \le t \le m$, if character $Z[k]$ does not appear in $X_t[l_{t,Z\langle k]}+1 : r_{t,Z[k+1\rangle}-1]$, then $l_{t,Z\langle k]} = r_{t,Z[k\rangle}$.

**Proof.** The lemma follows from definition of $l_{t,Z\langle k]}$ and $r_{t,Z[k\rangle}$. □

The proposed algorithm differs from the unmodified algorithm as follows.

1. The first character to be tested whether $\mathcal{X}_{(Z,k)}$ shares it is $Z[k]$. (Consequently, the maximum possible number of copies of $Z[k]$ are inserted between $Z\langle k]$ and $Z[k+1\rangle$, but they do not always appear as consecutive repeats in the resulting MCS.)
2. If $\mathcal{X}_{(Z,k)}$ does not share $Z[k]$ and no index is specified as $t_{(Z,k)}$, then an arbitrary index $t$ such that $Z[k]$ does not appear in $X_t[l_{t,Z\langle k]}+1 : r_{t,Z[k+1\rangle}-1]$ is specified as $t_{(Z,k)}$, which is also inherited as $t_{(Z',k)}$s for all $Z'$s having $Z$ as their $k$-precursor.
3. If some index $t$ is specified as $t_{(Z,k)}$, then the element to be tested whether $\mathcal{X}_{(Z,k)}$ shares it is $X_t[i]$ chosen from $X_t[i_{(Z,k)} : r_{t,Z[k+1\rangle}-1]$ in ascending order of $i$. Here, if $Z$ has no $k$-precursor, then $i_{(Z,k)} = l_{t,Z\langle k]}+1$; otherwise, $i_{(Z,k)}$ is the index such that the update procedure performs a forward update from $(Z', k)$ with insertion $X_t[i_{(Z,k)}-1]$, where $Z' \prec_k Z$.

We implement the proposed algorithm as Algorithm findMCS$(\mathcal{X}, Y)$ presented in Fig. 1. This algorithm consists of three phases, the initialization phase (lines 1 through 6), the iterative phase (lines 7 through 23), and the output phase (line 24), each corresponding to a distinct step of the update procedure listed in Definition 1 in the same order. As variables, stack $Z^{\langle]}$, string $Z^{[\rangle}$, and indices $r_s$ with $1 \le s \le m$ are used to maintain $Z$ and $k$. At the end of each iteration in the iterative phase, $Z^{\langle]}$ consists of $k+1$ tuples $\tau_0, \tau_1, \ldots, \tau_k$ in order from bottom to top. For any index $k'$ with $0 \le k' \le k$, $\tau_{k'} = (c, t, i, (l_1, l_2, \ldots, l_m))$, where $c$ represents $Z[k']$, if $k' \ge 1$, and $l_s$ represents $l_{s,Z\langle k']}$ for any index $s$ with $1 \le s \le m$. In contrast, $Z^{[\rangle}$ represents $Z[k+1\rangle$ and $r_s$ represents $r_{s,Z[k+1\rangle}$. This asymmetry is due to the fact that the update procedure updates $(Z, k)$ to either $(Z_c, k+1)$ or $(Z, k-1)$, where $Z_c = Z\langle k] \circ c \circ Z[k+1\rangle$ for some character $c$ that $\mathcal{X}_{(Z,k)}$ shares. While updating $l_{s,Z\langle k]}$ to $l_{s,Z_c\langle k+1]}$ $(= next(s, l_{s,Z\langle k]}, c))$, $r_{s,Z[k+1\rangle}$ to $r_{s,Z_c[k+2\rangle}$ $(= r_{s,Z[k+1\rangle})$, and $r_{s,Z[k+1\rangle}$ to $r_{s,Z[k\rangle}$ $(= prev(s, r_{s,Z[k+1\rangle}, Z[k]))$ are easy, updating $l_{s,Z\langle k]}$ to $l_{s,Z\langle k-1]}$ is difficult, so keeping $l_s$ in $Z^{\langle]}$ is adopted. Elements $t$ and $i$ in tuple $\tau_k$ represent $t_{(Z,k)}$ and $i_{(Z,k)}$, respectively. We use $t = 0$ to indicate that no index is specified as $t_{(Z,k)}$.

1: Let $Z^{\langle]}$ and $Z^{[\rangle}$ be the empty stack and string, respectively;
2: $l_s \leftarrow 0$ and $r_s \leftarrow |X_s| + 1$ for $s = 1, 2, \ldots, m$;
3: push $(0, 1, 0, (l_1, l_2, \ldots, l_m))$ to $Z^{\langle]}$;
4: for each character $c$ in $Y$ from $Y[1]$ to $Y[|Y|]$,
5: $l_s \leftarrow next(s, l_s, c)$ for $s = 1, 2, \ldots, m$;
6: push $(c, 0, 0, (l_1, l_2, \ldots, l_m))$ to $Z^{\langle]}$;
7: while $Z^{\langle]}$ is nonempty,
8: pop $(c, t, i, (l_1, l_2, \ldots, l_m))$ from $Z^{\langle]}$;
9: if $t = 0$, then
10: while $\forall s, next(s, l_s, c) < r_s$,
11: push $(c, 0, 0, (l_1, l_2, \ldots, l_m))$ to $Z^{\langle]}$;
12: $l_s \leftarrow next(s, l_s, c)$ for $s = 1, 2, \ldots, m$;
13: let $t$ be any index such that $next(t, l_t, c) \geq r_t$;
14: $i \leftarrow l_t + 1$;
15: while $i < r_t$ and $\exists s, next(s, l_s, X_t[i]) \geq r_s$,
16: $i \leftarrow i + 1$;
17: if $i < r_t$, then
18: push $(c, t, i + 1, (l_1, l_2, \ldots, l_m))$ to $Z^{\langle]}$;
19: $l_s \leftarrow next(s, l_s, X_t[i])$ for $s = 1, 2, \ldots, m$;
20: push $(X_t[i], 0, 0, (l_1, l_2, \ldots, l_m))$ to $Z^{\langle]}$;
21: otherwise, if $Z^{\langle]}$ is nonempty, then
22: $Z^{[\rangle} \leftarrow c \circ Z^{[\rangle}$;
23: $r_s \leftarrow prev(s, r_s, c)$ for $s = 1, 2, \ldots, m$;
24: output $Z^{[\rangle}$.

**Fig. 1** Algorithm findMCS($\{X_1, X_2, \ldots, X_m\}, Y$).

By using variables $Z^{\langle]}$, $Z^{[\rangle}$, and $r_1, r_2, \ldots, r_m$ as described above, Algorithm findMCS simulates the update procedure as follows. The initialization phase initializes $(Z, k)$ to $(Y, |Y|)$ in lines 1 through 6. The iterative phase repeats execution of lines 8 through 23 until $k = -1$. In each iteration, lines 8 through 14 check whether some index is specified as $t_{(Z,k)}$, and if not, insert the maximum possible number of copies of $Z[k]$ between $Z\langle k]$ and $Z[k+1\rangle$, increase $k$ by the number of the inserted copies of $Z[k]$, and specify an arbitrary index $t$ such that $Z[k]$ does not appear in $X_t[l_{t,Z\langle k]} + 1 : r_{t,Z[k+1\rangle} - 1]$ as $t_{(Z,k)}$. Lines 15 through 16 search in ascending order of $i$ for the first element $X_{t_{(Z,k)}}[i]$ shared by $\mathcal{X}_{(Z,k)}$ in $X_{t_{(Z,k)}}[i_{(Z,k)} : r_{t_{(Z,k)},Z[k+1\rangle} - 1]$. Finally, lines 18 through 20 performs a forward update from $(Z, k)$ to $(Z\langle k] \circ X_{t_{(Z,k)}}[i] \circ Z[k+1\rangle, k+1)$, if any, or lines 22 through 23 performs a backward update from $(Z, k)$ to $(Z, k-1)$, otherwise. The output phase outputs the resulting $Z$ in line 24.

The only concern about the correctness of Algorithm findMCS is that if $t \neq 0$ and $i \geq l_{t,Z\langle k]}$ in line 8, then testing whether $\mathcal{X}_{(Z,k)}$ shares any element in $X_t[l_{t,Z\langle k]} + 1 : i - 1]$ is skipped. Since this skip is essential to make the algorithm's execution time $O(mn \log n)$, we show below that $\mathcal{X}_{(Z,k)}$ shares no such element. Any tuple popped from $Z^{\langle]}$ by line 8 with $t \neq 0$ and $i \geq l_{t,Z\langle k]}$ is the one that was pushed by line 18 to perform a forward update from $(Z', k)$ with insertion of $X_t[i_{(Z,k)} - 1]$, where $Z' \prec_k Z$. This implies that $i = i_{(Z,k)}$ and $\mathcal{X}_{(Z',k)}$ shares no element in $X_t[i_{(Z',k)} : i_{(Z,k)} - 2]$. Hence, from Lemma 4, $\mathcal{X}_{(Z,k)}$ shares no element in $X_t[i_{(Z',k)} : i_{(Z,k)} - 1]$. By repeatedly applying the same argument to $Z'$ and so on, we also have a chain $Z_1 \prec_k Z_2 \prec_k \cdots \prec_k Z_\ell = Z'$ such that $i_{(Z_1,k)} = l_{t,Z\langle k]} + 1$ and for any index $\ell'$ with $2 \leq \ell' \leq \ell$, $\mathcal{X}_{(Z,k)}$ shares no element in $X_t[i_{(Z_{\ell'-1},k)} : i_{(Z_{\ell'},k)} - 1]$. Therefore, $\mathcal{X}_{(Z,k)}$ shares no element in $X_t[l_{t,Z\langle k]} + 1 : i - 1]$, and thus we obtain the following theorem.

**Theorem 1:** Algorithm findMCS($\mathcal{X}, Y$) outputs an MCS that contains $Y$.

The execution time and required space of Algorithm findMCS($\mathcal{X}, Y$) are estimated as follows. The initialization phase (lines 1 through 6) and the output phase (line 24) are respectively executed in $O(n \log n)$ time and $O(n/m)$ time because the length of any string that is shared by $\mathcal{X}$ is $O(n/m)$. It is easy to verify that the iterative phase (lines 7 through 23) is executed in time linear in the product of $O(m \log n)$ and the number of times line 15 of the algorithm is executed. The number of times line 15 is executed is equal to the sum of the number of elements in $X_{t_{(Z,k)}}[l_{t_{(Z,k)},Z\langle k]}+1 : r_{t_{(Z,k)},Z[k+1\rangle}-1]$ over all backward updates from $(Z, k)$ to $(Z, k-1)$, which are executed by lines 22 through 23. Let $t$ be an arbitrary index with $1 \leq t \leq m$. Consider any backward update from $(Z, k)$ to $(Z, k-1)$ with $t_{(Z,k)} = t$ that is performed before another backward update from $(Z', k')$ to $(Z', k')$ with $t_{(Z',k')} = t$ is performed. Since $Z[k\rangle$ is a suffix of $Z'[k'+1\rangle$, $r_{t,Z'[k'+1\rangle} \leq r_{t,Z[k\rangle}$. On the other hand, since $Z[k]$ does not appear in $X_t[l_{t,Z\langle k]} + 1 : r_{t,Z[k+1\rangle} - 1]$, $l_{t,Z\langle k]} = r_{t,Z[k\rangle}$ due to Lemma 5. Therefore, $r_{t,Z'[k'+1\rangle} \leq l_{t,Z\langle k]}$, implying that $X_t[l_{t,Z\langle k]}+1 : r_{t,Z[k+1\rangle}-1]$ and $X_t[l_{t,Z'\langle k']}+1 : r_{t,Z'[k'+1\rangle}-1]$ never overlap. Hence, line 15 is executed at most $n$ times, and thus the algorithm runs in $O(mn \log n)$ time. Since $Z^{\langle]}$ stores $O(n/m)$ tuples, each consisting of a character and $m+2$ indices, the algorithm uses $O(n)$ space.

**Theorem 2:** Algorithm findMCS($\mathcal{X}, Y$) runs in $O(mn \log n)$ time and $O(n)$ space.

### References

[1] 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., vol.90, pp.175–179, 2004.
[2] A. Conte, R. Grossi, G. Punzi, T. Uno, "Enumeration of maximal common subsequence between two strings," Algorithmica, vol.84, no.3, pp.757–783, 2022.
[3] C.B. Fraser, R.W. Irving, M. Middendorf, "Maximal common subsequences and minimal common supersubsequences," Inf. Comput., vol.124, no.2, pp.145–153, 1996.
[4] D. Maier, "The complexity of some problems on subsequences and supersequences," J. ACM, vol.25, no.2, pp.322–336, 1978.
[5] Y. Sakai, "Maximal common subsequence algorithms," Theor. Comput. Sci., vol.793, pp.132–139, 2019.
[6] Y.-T. Tsai, "The constrained longest common subsequence problem," Inf. Process. Lett., vol.88, no.4, pp.173–176, 2003.
[7] R.A. Wagner, M.J. FIscher, "The string-to-string correction problem," J. ACM, vol.21, no.1, pp.168–173, 1974.