A Polynomial Time Algorithm for a Generalized Longest Common Subsequence Problem

Xiaodong Wang[2], Yingjie Wu[3], and Daxin Zhu[1]([⊠])

[1] Quanzhou Normal University, Quanzhou 362000, China
dex@qztc.edu.cn
[2] Fujian University of Technology, Fuzhou 350108, China
[3] Fuzhou University, Fuzhou 350002, China

Abstract. In this paper, we consider a generalized longest common subsequence problem with multiple substring exclusive constraints. For the two input sequences X and Y of lengths n and m, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r, the problem is to find a common subsequence Z of X and Y excluding each of constraint string in P as a substring and the length of Z is maximized. A very simple dynamic programming algorithm to this problem is presented in this paper. The correctness of the new algorithm is demonstrated. The time and space complexities of the new algorithm are both $O(nmr)$.

1 Introduction

The longest common subsequence (LCS) problem is a classic computer science problem, and has applications in bioinformatics. It is also widely applied in diverse areas, such as file comparison, pattern matching and computational biology [3,4,8,9]. Given two sequences X and Y, the longest common subsequence problem is to find a subsequence of X and Y whose length is the longest among all common subsequences of the two given sequences. It differs from the problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The most referred algorithm, proposed by Wagner and Fischer [29], solves the LCS problem by using a dynamic programming algorithm in quadratic time. Other advanced algorithms were proposed in the past decades [2–4,16,17,19,21]. If the number of input sequences is not fixed, the problem to find the LCS of multiple sequences has been proved to be NP-hard [23]. Some approximate and heuristic algorithms were proposed for these problems [6,25].

For some biological applications some constraints must be applied to the LCS problem. These kinds of variants of the LCS problem are called the constrained LCS (CLCS) problem. One of the recent variants of the LCS problem, the constrained longest common subsequence (CLCS) which was first addressed by Tsai [27], has received much attention. It generalizes the LCS measure by introducing of a third sequence, which allows to extort that the obtained CLCS has some special properties [26]. For two given input sequences X and Y of lengths m and

n, respectively, and a constrained sequence P of length r, the CLCS problem is to find the common subsequences Z of X and Y such that P is a subsequence of Z and the length of Z is the maximum. The most referred algorithms were proposed independently [5,8], which solve the CLCS problem in $O(mnr)$ time and space by using dynamic programming algorithms. Some improved algorithms have also been proposed [11,18]. The LCS and CLCS problems on the indeterminate strings were discussed in [20]. Moreover, the problem was extended to the one with weighted constraints, a more generalized problem [24].

Recently, a new variant of the CLCS problem, the restricted LCS problem, was proposed [14], which excludes the given constraint as a subsequence of the answer. The restricted LCS problem becomes NP-hard when the number of constraints is not fixed. Some more generalized forms of the CLCS problem, the generalized constrained longest common subsequence (GC-LCS) problems, were addressed independently by Chen and Chao [7]. For the two input sequences X and Y of lengths n and m, respectively, and a constraint string P of length r, the GC-LCS problem is a set of four problems which are to find the LCS of X and Y including/excluding P as a subsequence/substring, respectively. The four generalized constrained LCS [7] can be summarized in Table 1.

Table 1. The GC-LCS problems

Problem	Input	Output
SEQ-IC-LCS	X, Y, and P	The longest common subsequence of X and Y including P as a subsequence
STR-IC-LCS	X, Y, and P	The longest common subsequence of X and Y including P as a substring
SEQ-EC-LCS	X, Y, and P	The longest common subsequence of X and Y excluding P as a subsequence
STR-EC-LCS	X, Y, and P	The longest common subsequence of X and Y excluding P as a substring

For the four problems in Table 1, $O(mnr)$ time algorithms were proposed [7]. For all four variants in Table 1, $O(r(m+n)+(m+n)\log(m+n))$ time algorithms were proposed by using the finite automata [12]. Recently, a quadratic algorithm to the STR-IC-LCS problem was proposed [10], and the time complexity of [12] was pointed out not correct.

The four GC-LCS problems can be generalized further to the cases of multiple constraints. In these generalized cases, the single constrained pattern P will be generalized to a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r, as shown in Table 2.

The problem M-SEQ-IC-LCS has been proved to be NP-hard in [13]. The problem M-SEQ-EC-LCS has also been proved to be NP-hard in [14,28]. In addition, the problems M-STR-IC-LCS and M-STR-EC-LCS were also declared

Table 2. The Multiple-GC-LCS problems

Problem	Input	Output
M-SEQ-IC-LCS	X, Y, and a set of constraints $P = \{P_1, \cdots, P_d\}$	LCS of X and Y including each of constraint $P_i \in P$ as a subsequence
M-STR-IC-LCS	X, Y, and a set of constraints $P = \{P_1, \cdots, P_d\}$	LCS of X and Y including each of constraint $P_i \in P$ as a substring
M-SEQ-EC-LCS	X, Y, and a set of constraints $P = \{P_1, \cdots, P_d\}$	LCS of X and Y excluding each of constraint $P_i \in P$ as a subsequence
M-STR-EC-LCS	X, Y, and a set of constraints $P = \{P_1, \cdots, P_d\}$	LCS of X and Y excluding each of constraint $P_i \in P$ as a substring

to be NP-hard in [7], but without a proof. The exponential-time algorithms for solving these two problems were also presented in [7].

We will discuss the problem M-STR-EC-LCS in this paper. The failure functions in the Knuth-Morris-Pratt algorithm [22] for solving the string matching problem have been proved very helpful for solving the STR-EC-LCS problem. It has been found by Aho and Corasick [1] that the failure functions can be generalized to the case of keyword tree to speedup the exact string matching of multiple patterns. This idea can be very supportive in our dynamic programming algorithm. This is the principle idea of our new algorithm.

The organization of the paper is as follows.

In the following 3 sections, we describe our presented dynamic programming algorithm for the M-STR-EC-LCS problem.

In Sect. 2 the preliminary knowledge for presenting our algorithm for the M-STR-EC-LCS problem is discussed. In Sect. 3 we give a new dynamic programming solution for the M-STR-EC-LCS problem with time complexity $O(nmr)$, where n and m are the lengths of the two given input strings, and r is the total length of d constraint strings. In Sect. 4, we consider the issues to implement the algorithm efficiently.

2   Preliminaries

A sequence is a string of characters over an alphabet $\sum$. A subsequence of a sequence X is obtained by deleting zero or more characters from X (not necessarily contiguous). A substring of a sequence X is a subsequence of successive characters within X.

For a given sequence $X = x_1 x_2 \cdots x_n$ of length n, the ith character of X is denoted as $x_i \in \sum$ for any $i = 1, \cdots, n$. A substring of X from position i to j can be denoted as $X[i : j] = x_i x_{i+1} \cdots x_j$. If $i \neq 1$ or $j \neq n$, then the

substring $X[i : j] = x_i x_{i+1} \cdots x_j$ is called a proper substring of X. A substring $X[i : j] = x_i x_{i+1} \cdots x_j$ is called a prefix or a suffix of X if $i = 1$ or $j = n$, respectively.

For the two input sequences $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_m$ of lengths n and m, respectively, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r, the problem M-STR-EC-LCS is to find an LCS of X and Y excluding each of constraint $P_i \in P$ as a substring.

Keyword tree (Aho-Corasick Automaton) [1,9,15] is a main data structure in our dynamic programming algorithm to process the constraint set P of the M-STR-EC-LCS problem.

Definiton 1. *The keyword tree for set P is a rooted directed tree T satisfying 3 conditions: 1. each edge is labeled with exactly one character; 2. any two edges out of the same node have distinct labels; and 3. every string P_i in P maps to some node v of T such that the characters on the path from the root of T to v exactly spell out P_i, and every leaf of T is mapped to some string in P.*

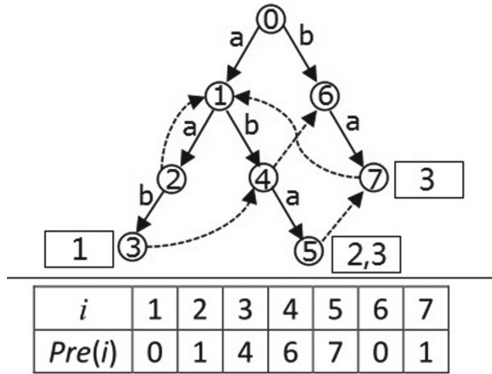| i	1	2	3	4	5	6	7
$Pre(i)$	0	1	4	6	7	0	1

Fig. 1. Keyword trees

In order to identify the nodes of T, we assign numbers $0, 1, \cdots, t - 1$ to all t nodes of T in their preorder numbering. Then, each node will be assigned an integer $i, 0 \leq i < t$, as shown in Fig. 1. For each node numbered i of a keyword tree T, the concatenation of characters on the path from the root to the node i spells out a string denoted as $L(i)$. The string $L(i)$ is also called the label of the node i in the keyword tree T. For example, Fig. 1 shows the keyword tree T for the constraint set $P = \{aab, aba, ba\}$, where $P_1 = aab, P_2 = aba, P_3 = ba$, and $d = 3, r = 8$. Clearly, every node in the keyword tree corresponds to a prefix of one of the strings in set P, and every prefix of a string P_i in P maps to a distinct node in the keyword tree T. The keyword tree for set P of total length r of all strings can be easily constructed in $O(r)$ time for a constant alphabet size.

The keyword tree can be extended into an automaton, Aho-Corasick automaton, which is composed of three functions, a goto function, an output function and a failure function. The goto function is presented as the solid edges of the keyword tree and the output function indicates when the matches occur and which strings are output. For each node i, its output function is denoted as O_i, a set of indices which indicate when the node i is reached then for each index $j \in O_i$, the string P_j is matched. For example, the output sets of nodes 3, 5 and 7 are $O_3 = \{1\}, O_5 = \{2, 3\}$ and $O_7 = \{3\}$, which means that the outputs of node 3, 5 and 7 are $\{P_1 = aab\}, \{P_2 = aba, P_3 = ba\}$ and $\{P_3 = ba\}$, respectively.

The failure function indicates which node to go if there is no character to be further matched. It is a generalization of the failure functions in the Knuth-Morris-Pratt algorithm for solving the string matching problem. It is represented by the dashed edges in Fig. 1.

For any node i of T, define $lp(i)$ to be the length of the longest proper suffix of string $L(i)$ that is a prefix of some string in T. It can be verified readily that for each node i of T, if A is an $lp(i)$-length suffix of string $L(i)$, then there must be a unique node $pre(i)$ in T such that $L(pre(i)) = A$. If $lp(i) = 0$ then $pre(i) = 0$ is the root of T.

The ordered pair $(i, pre(i))$ is called a failure link. The failure link is a direct generalization of the failure functions in the KMP algorithm. For example, in Fig. 1, failure links are shown as pointers from every node i to node $pre(i)$ where $lp(i) > 0$. The other failure links point to the root and are not shown. The failure links of T define actually a failure function pre for the constraint set P. As stated in [1,9], for a constant alphabet size, in the worst case, the failure function pre can be computed in $O(r)$ time.

The failure list of a given node is the ordered list of the nodes which locate on the path to the root via dashed edges. For example, for the nodes $i = 1, 2, 3, 4, 5, 6, 7$, the corresponding values of failure function are $pre(i) = 0, 1, 4, 6, 7, 0, 1$. The failure list of node 5 is $\{7 \to 1 \to 0\}$, and the failure list of node 6 is $\{0\}$, as shown in Fig. 1.

The failure function pre is used to speedup the search for all occurrences in a text Z of strings from P. For each node i of T, and a character $c \in \sum$, if no edges out of the node i is labeled c, then the failure link of node i direct the search to the node $pre(i)$. It is equivalent to add the edge $(i, pre(i))$ labeled c to the node i. This set matching method generalized the next function in KMP algorithm to the Aho-Corasick-next function as follows.

Definiton 2. *Given a keyword tree T and its failure function, for each node i of T and each character $c \in \sum$, Aho-Corasick-next function $\delta(i, c)$ denotes the destination of the first node in i's failure list which has an edge labeled c. If there exists no such node in the failure list, the function returns the root.*

Table 3 shows the Aho-Corasick-next function δ corresponding to the example in Fig. 1.

We take node 4 as an example. It can be seen from Fig. 1 that $\delta(4, a) = 5$ and $\delta(4, b) = 0$. It is easy to understand that each element of Aho-Corasick-next function can be computed in constant time.

Table 3. Aho-Corasick-next function

δ	0	1	2	3	4	5	6	7
a	1	2	1	4	5	1	7	1
b	6	4	3	0	0	1	0	1

The symbol $\oplus$ is also used to denote the string concatenation. For example, if $S_1 = aaa$ and $S_2 = bbb$, then it is readily seen that $S_1 \oplus S_2 = aaabbb$.

3   Our Main Result: A Dynamic Programming Algorithm

Let T be a keyword tree for the given constraint set P, and $Z[1 : l] = z_1, z_2, \cdots, z_l$ be any common subsequence of X and Y. If we search the set matching of Z from the root of T in the direction of the Aho-Corasick-next function δ of T, then the search will stop in a node i of T. All such common subsequence of X and Y can be classified into a group k, $0 \le k < t$. These t groups can be used to distinguish the different states in our dynamic programming algorithm. For each integer k, $0 \le k < t$, the state k represents the set of common subsequence of X and Y in group k.

Definiton 3. *Let $Z(i,j,k)$ denote the set of all LCSs of $X[1 : i]$ and $Y[1 : j]$ with state k, where $1 \le i \le n, 1 \le j \le m$, and $0 \le k < t$. The length of an LCS in $Z(i,j,k)$ is denoted as $f(i,j,k)$.*

If we can compute $f(i,j,k)$ for any $1 \le i \le n, 1 \le j \le m$, and $0 \le k < t$ efficiently, then the length of an LCS of X and Y excluding P must be $\max\limits_{0 \le k < t} \{f(n,m,k)|O_k = \emptyset\}$.

By using the keyword tree data structure described in the last section, we can give a recursive formula for computing $f(i,j,k)$ by the following theorem.

Theorem 1. *For the two input sequences $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_m$ of lengths n and m, respectively, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r, let $Z(i,j,k)$ and $f(i,j,k)$ be defined as in Definition 3. Suppose a keyword tree T for the constraint set P has been built, and the t nodes of T are numbered in their preorder numbering. Then, for any $1 \le i \le n, 1 \le j \le m$, and $0 \le k < t$, $f(i,j,k)$ can be computed by the following recursive formula.*

$$f(i,j,k) = \begin{cases} \max\left\{f(i-1,j,k), f(i,j-1,k)\right\} & if\ x_i \ne y_j, \\ \max\left\{f(i-1,j-1,k), 1 + \max\limits_{\bar{k} \in S(k,x_i)} \left\{f(i-1,j-1,\bar{k})\right\}\right\} & if\ x_i = y_j. \end{cases} \quad (1)$$

where,

$$S(k, x_i) = \{\bar{k}|0 \le \bar{k} < t, \delta(\bar{k}, x_i) = k\} \quad (2)$$

The boundary conditions of this recursive formula are $f(i,0,0) = f(0,j,0) = 0$ for any $0 \le i \le n, 0 \le j \le m$.

Proof. For any $0 \le i \le n, 0 \le j \le m$, and $0 \le k < t$, suppose $f(i,j,k) = l$ and $z = z_1 \cdots z_l \in Z(i,j,k)$.

First of all, we notice that for each pair $(i',j'), 1 \le i' \le n, 1 \le j' \le m$, such that $i' \le i$ and $j' \le j$, we have $f(i',j',k) \le f(i,j,k)$, since a common subsequence z of $X[1:i']$ and $Y[1:j']$ with state k is also a common subsequence of $X[1:i]$ and $Y[1:j]$ with state k.

(1) In the case of $x_i \ne y_j$, we have $x_i \ne z_l$ or $y_j \ne z_l$.

    (1.1) If $x_i \ne z_l$, then $z = z_1 \cdots z_l$ is a common subsequence of $X[1:i-1]$ and $Y[1:j]$ with state k, and so $f(i-1,j,k) \ge l$. On the other hand, $f(i-1,j,k) \le f(i,j,k) = l$. Therefore, in this case we have $f(i,j,k) = f(i-1,j,k)$.

    (1.2) If $y_j \ne z_l$, then we can prove similarly that in this case, $f(i,j,k) = f(i,j-1,k)$.

    Combining the two subcases we conclude that in the case of $x_i \ne y_j$, we have

$$f(i,j,k) = \max \{f(i-1,j,k), f(i,j-1,k)\}.$$

(2) In the case of $x_i = y_j$, there are also two cases to be distinguished.

    (2.1) If $x_i = y_j \ne z_l$, then $z = z_1 \cdots z_l$ is also a common subsequence of $X[1:i-1]$ and $Y[1:j-1]$ with state k, and so $f(i-1,j-1,k) \ge l$. On the other hand, $f(i-1,j-1,k) \le f(i,j,k) = l$. Therefore, in this case we have $f(i,j,k) = f(i-1,j-1,k)$.

    (2.2) If $x_i = y_j = z_l$, then $f(i,j,k) = l > 0$ and $z = z_1 \cdots z_l$ is an LCS of $X[1:i]$ and $Y[1:j]$ with state k.

Let the state of $(z_1, \cdots, z_{l-1})$ be $\bar{k}$, then we have $\bar{k} \in S(k,x_i)$, since $z_l = x_i$. It follows that $z_1 \cdots z_{l-1}$ is a common subsequence of $X[1:i-1]$ and $Y[1:j-1]$ with state $\bar{k}$. Therefore, we have

$$f(i-1,j-1,\bar{k}) \ge l-1$$

Furthermore, we have

$$\max_{\bar{k} \in S(k,x_i)} \{f(i-1,j-1,\bar{k})\} \ge l-1$$

In other words,

$$f(i,j,k) \le 1 + \max_{\bar{k} \in S(k,x_i)} \{f(i-1,j-1,\bar{k})\} \tag{3}$$

On the other hand, for any $\bar{k} \in S(k,x_i)$, and $v = v_1 \cdots v_h \in Z(i-1,j-1,\bar{k})$, $v \oplus x_i$ is a common subsequence of $X[1:i]$ and $Y[1:j]$ with state k. Therefore, $f(i,j,k) = l \ge 1 + h = 1 + f(i-1,j-1,\bar{k})$, and so we conclude that,

$$f(i,j,k) \ge 1 + \max_{\bar{k} \in S(k,x_i)} \{f(i-1,j-1,\bar{k})\} \tag{4}$$

Combining (3) and (4) we have, in this case,

$$f(i, j, k) = 1 + \max_{\bar{k} \in S(k, x_i)} \left\{ f(i - 1, j - 1, \bar{k}) \right\} \tag{5}$$

Combining the two subcases in the case of $x_i = y_j$, we conclude that the recursive formula (1) is correct for the case $x_i = y_j$.

The proof is complete.   ∎

4   The Implementation of the Algorithm

According to Theorem 1, our algorithm for computing $f(i, j, k)$ is a standard 3-dimensional dynamic programming algorithm. By the recursive formula (1), the dynamic programming algorithm for computing $f(i, j, k)$ can be implemented as the following Algorithm 1.

Algorithm 1. M-STR-EC-LCS

Input: Strings $X = x_1 \cdots x_n$, $Y = y_1 \cdots y_m$ of lengths n and m, respectively, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r
Output: The length of an LCS of X and Y excluding P
 1: Build a keyword tree T for P
 2: **for all** i, j, $0 \leq i \leq n, 0 \leq j \leq m$ **do**
 3:     $f(i, 0, 0) \leftarrow 0, f(0, j, 0) \leftarrow 0$ {boundary condition}
 4: **end for**
 5: $S \leftarrow \{0\}$ {current set of states}
 6: **for** $i = 1$ to n **do**
 7:     **for** $j = 1$ to m **do**
 8:         **for each** $k \in S$ **do**
 9:             **if** $x_i \neq y_j$ **then**
10:                 $f(i, j, k) \leftarrow \max\{f(i - 1, j, k), f(i, j - 1, k)\}$
11:             **else**
12:                 $\bar{k} \leftarrow \delta(k, x_i)$
13:                 **if** $|O_{\bar{k}}| = 0$ **then**
14:                     $f(i, j, \bar{k}) \leftarrow \max\{f(i - 1, j - 1, \bar{k}), 1 + f(i - 1, j - 1, k)\}$
15:                     $S \leftarrow S \bigcup \{\bar{k}\}$
16:                 **end if**
17:             **end if**
18:         **end for**
19:     **end for**
20: **end for**
21: **return** $\max_{0 \leq k < t} \{f(n, m, k)\}$

In Algorithm 1, T is the keyword tree for set P. The root of the keyword tree is numbered 0, and the other nodes are numbered $1, 2, \cdots, t - 1$ in their preorder numbering. $\delta(\alpha, c)$ is the Aho-Corasick-next function defined in

Definition 2, which can be computed in $O(1)$ time. O_k is the output set of node k in T. The variable S is used to record the current states created. When a node is visited first time, a new state may be created. Therefore, in Algorithm 1, the current state set S is extended gradually while the for loop processed. In the worst case, the set S will have a size of r, the total lengths of the constrained strings. The body of the triple for loops can be computed in $O(1)$ time in the worst case. Therefor, the total time of Algorithm 1 is $O(nmr)$. The space used by Algorithm 1 is also $O(nmr)$.

The number of constraints is an influent factor in the time and space complexities of our new algorithm. If a string P_i in the constraint set P is a proper substring of another string P_j in P, then an LCS of X and Y excluding P_i must also exclude P_j. For this reason, the constraint string P_j can be removed from constraint set P without changing the solution of the problem. Without loss of generality, we can put forward the following two assumptions on the constraint set P.

Assumption 1. *There are not any duplicated strings in the constraint set P.*

Assumption 2. *No string in the constraint set P is a proper substring of any other string in P.*

If Assumption 1 is violated, then there must be some duplicated strings in the constraint set P. In this case, we can first sort the strings in the constraint set P, then duplicated strings can be removed from P easily and then Assumption 1 on the constraint set P is satisfied. It is clear that removed strings will not change the solution of the problem.

For Assumption 2, we first notice that a string A in the constraint set P is a proper substring of string B in P, if and only if in the keyword tree T of P, there is a directed path of failure links from a node v on the path from the root to the leaf node corresponding to string B to the leaf node corresponding to string A [1,9]. For instance, in Fig. 1, there is a directed path of failure links from node 5 to node 7 and thus we know the string ba corresponding to node 7 is a proper substring of string aba corresponding to node 5.

With this fact, if Assumption 2 is violated, we can remove all proper super strings from the constraint set P as follows. We first build a keyword tree T for the constraint set P, then mark all the leaf nodes pointed by a failure link in T by using a depth first traversal of T. All the strings corresponding to the marked leaf node can then be removed from P. Assumption 2 is now satisfied on the new constraint set and the keyword tree T for the new constraint set is then rebuilt. It is not difficult to do this preprocessing in $O(r)$ time. It is clear that the removed proper substrings will not change the solution of the problem.

If we want to compute the longest common subsequence of X and Y excluding P, but not just its length, we can also present a simple recursive backtracking algorithm for this purpose as the following Algorithm 2.

In the end of our new algorithm, we will find an index k such that $f(n, m, k)$ gives the length of an LCS of X and Y excluding P. Then, a function call $back(n, m, k)$ will produce the answer LCS accordingly.

Algorithm 2. $back(i, j, k)$

Comments: A recursive back tracing algorithm to construct the answer LCS

1: **if** $i = 0$ **or** $j = 0$ **then**
2:     **return**
3: **end if**
4: **if** $x_i = y_j$ **then**
5:     **if** $f(i, j, k) = f(i - 1, j - 1, k)$ **then**
6:         $back(i - 1, j - 1, k)$
7:     **else**
8:         **for each** $\bar{k} \in S$ **do**
9:             **if** $k = \delta(\bar{k}, x_i)$ **and** $f(i, j, k) = 1 + f(i - 1, j - 1, \bar{k})$ **then**
10:                 $back(i - 1, j - 1, \bar{k})$
11:                 **print** x_i
12:             **end if**
13:         **end for**
14:     **end if**
15: **else if** $f(i - 1, j, k) > f(i, j - 1, k)$ **then**
16:     $back(i - 1, j, k)$
17: **else**
18:     $back(i, j - 1, k)$
19: **end if**

Since the cost of $\delta(k, x_i)$ is $O(1)$ in the worst case, the time complexity of the algorithm $back(i, j, k)$ is $O(n + m)$.

Finally we summarize our results in the following theorem.

Theorem 2. *For the two input sequences $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_m$ of lengths n and m, respectively, and a set of d constraints $P = \{P_1, \cdots, P_d\}$ of total length r, the Algorithms 1 and 2 solve the M-STR-EC-LCS problem correctly in $O(nmr)$ time and $O(nmr)$ space, with preprocessing time $O(r|\Sigma|)$.*

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Ann, H.Y., Yang, C.B., Tseng, C.T., Hor, C.Y.: A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. Inform. Process Lett. **108**(11), 360–364 (2008)
3. Ann, H.Y., Yang, C.B., Peng, Y.H., Liaw, B.C.: Efficient algorithms for the block edit problems. Inf. Comput. **208**(3), 221–229 (2010)
4. Apostolico, A., Guerra, C.: The longest common subsequences problem revisited. Algorithmica **2**(1), 315–336 (1987)
5. Arslan, A.N., Egecioglu, O.: Algorithms for the constrained longest common subsequence problems. Int. J. Found. Comput. Sci. **16**(6), 1099–1109 (2005)
6. Blum, C., Blesa, M.J., Lpez-Ibnez, M.: Beam search for the longest common subsequence problem. Comput. Oper. Res. **36**(12), 3178–3186 (2009)

7. Chen, Y.C., Chao, K.M.: On the generalized constrained longest common subsequence problems. J. Comb. Optim. **21**(3), 383–392 (2011)
8. Chin, F.Y.L., Santis, A.D., Ferrara, A.L., Ho, N.L., Kim, S.K.: A simple algorithm for the constrained sequence problems. Inform. Process. Lett. **90**(4), 175–179 (2004)
9. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge, UK (2007)
10. Deorowicz, S.: Quadratic-time algorithm for a string constrained LCS problem. Inform. Process. Lett. **112**(11), 423–426 (2012)
11. Deorowicz, S., Obstoj, J.: Constrained longest common subsequence computing algorithms in practice. Comput. Inform. **29**(3), 427–445 (2010)
12. Farhana, E., Ferdous, J., Moosa, T., Rahman, M.S.: Finite automata based algorithms for the generalized constrained longest common subsequence problems. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 243–249. Springer, Heidelberg (2010)
13. Gotthilf, Z., Hermelin, D., Lewenstein, M.: Constrained LCS: hardness and approximation. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 255–262. Springer, Heidelberg (2008)
14. Gotthilf, Z., Hermelin, D., Landau, G.M., Lewenstein, M.: Restricted LCS. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 250–257. Springer, Heidelberg (2010)
15. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
16. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 664–675 (1977)
17. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977)
18. Iliopoulos, C.S., Rahman, M.S.: New efficient algorithms for the LCS and constrained LCS problems. Inform. Process. Lett. **106**(1), 13–18 (2008)
19. Iliopoulos, C.S., Rahman, M.S.: A new efficient algorithm for computing the longest common subsequence. Theor. Comput. Sci. **45**(2), 355–371 (2009)
20. Iliopoulos, C.S., Rahman, M.S., Rytter, W.: Algorithms for two versions of LCS problem for indeterminate strings. J. Comb. Math. Comb. Comput. **71**, 155–172 (2009)
21. Iliopoulos, C.S., Rahman, M.S., Vorcek, M., Vagner, L.: Finite automata based algorithms on subsequences and supersequences of degenerate strings. J. Discrete Algorithm **8**(2), 117–130 (2010)
22. Knuth, D.E., Morris, J.H., Pratt, V.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
23. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**, 322–336 (1978)
24. Peng, Y.H., Yang, C.B., Huang, K.S., Tseng, K.T.: An algorithm and applications to sequence alignment with weighted constraints. Int. J. Found. Comput. Sci. **21**(1), 51–59 (2010)
25. Shyu, S.J., Tsai, C.Y.: Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Comput. Oper. Res. **36**(1), 73–91 (2009)
26. Tang, C.Y., Lu, C.L.: Constrained multiple sequence alignment tool development and its application to RNase family alignment. J. Bioinform. Comput. Biol. **1**, 267–287 (2003)

27. Tsai, Y.T.: The constrained longest common subsequence problem. Inform. Process. Lett. **88**(4), 173–176 (2003)
28. Tseng, C.T., Yang, C.B., Ann, H.Y.: Efficient algorithms for the longest common subsequence problem with sequential substring constraints. J. Complex. **29**, 44–52 (2013)
29. Wagner, R., Fischer, M.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)
30. Wang, L., Wang, X., Wu, Y., Zhu, D.: A dynamic programming solution to a generalized LCS problem. Inform. Process. Lett. **113**(1), 723–728 (2013)