JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 26, Number 9, 2019

Pp. 938–947
DOI: 10.1089/cmb.2019.0008

Efficient Computation of Longest Common Subsequences with Multiple Substring Inclusive Constraints

XIAODONG WANG,[1] LEI WANG,[2] and DAXIN ZHU[3]

ABSTRACT

In this article, we consider a generalized longest common subsequence (LCS) problem with multiple substring inclusive 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 including each of constraint string in P as a substring and the length of Z is maximized. A new dynamic programming solution to this problem is presented in this article. The correctness of the new algorithm is proved. The time complexity of our algorithm is $O(d2^dnmr)$. In the case of the number of constraint strings is fixed, our new algorithm for the generalized LCS problem with multiple substring inclusive constraints requires $O(nmr)$ time and space.

Keywords: dynamic programming, generalized longest common subsequence problem, longest common subsequence, multiple substring inclusive constraints.

1. INTRODUCTION

The longest common subsequence (LCS) problem is a classic computer science problem, and has applications in bioinformatics. It is further widely applied in diverse areas, such as file comparison, pattern matching, and computational biology (Apostolico and Guerra, 1987; Chin et al., 2004; Ann et al., 2010). Given two sequences X and Y, the LCS 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 (1974), solves the LCS problem by using a dynamic programming algorithm in quadratic time. Other advanced algorithms were proposed in the past decades (Hirschberg, 1977; Apostolico and Guerra, 1987; Ann et al., 2008, 2010; Iliopoulos and Rahman, 2009). If the number of input sequences is not fixed, the problem to find the LCS of multiple sequences has been proved to be nondeterministic polynomial (NP)-hard. Some approximate and heuristic algorithms were proposed for these problems (Blum et al., 2009).

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 longest common subsequence (CLCS) problem. One of the recent

[1]School of Information Science and Engineering, University of Technology, Fuzhou, China.
[2]Facebook, Inc., Menlo Park, California.
[3]School of Mathematics and Computer Science, Quanzhou Normal University, Quanzhou, China.

variants of the LCS problem, the CLCS that was first addressed by Tsai (2003), 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. 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 (Chin et al., 2004; Arslan and Egecioglu, 2005), which solve the CLCS problem in $O(mnr)$ time and space by using dynamic programming algorithms. Some improved algorithms have also been proposed (Iliopoulos and Rahman, 2008; Deorowicz and Obstoj, 2010; Wang et al., 2013). The LCS and CLCS problems on the indeterminate strings were extended to that with weighted constraints, a more generalized problem (Peng et al., 2010).

Recently, a new variant of the CLCS problem, the restricted LCS problem, was proposed, 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 (2011).

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 that are to find the LCS of X and Y including/excluding P as a subsequence/substring, respectively. The four GC-LCS problems (Chen and Chao, 2011) can be summarized in Table 1.

For the four problems given in Table 1, $O(mnr)$ time algorithms were proposed (Chen and Chao, 2011). 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. Recently, a quadratic algorithm to the STR-IC-LCS problem was proposed (Deorowicz, 2012).

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-EC-LCS has also been proved to be NP-hard in Tseng and Yang (2013). In addition, the problems M-STR-IC-LCS and M-STR-EC-LCS were also declared to be NP-hard in Chen and Chao (2011), but without a proof.

We discuss the problem M-STR-IC-LCS in this article. The failure functions in the Knuth–Morris–Pratt (KMP) algorithm (Knuth et al., 1977) for solving the string matching problem have been proved very helpful for solving the STR-IC-LCS problem. It has been found by Aho and Corasick (1975) that the failure functions can be generalized to the case of keyword tree to speed up the exact string matching of multiple patterns. This idea can be very helpful in our dynamic programming algorithm. This is the principle idea of our new algorithm, and it enables us to design a very efficient algorithm for the M-STR-IC-LCS problem with time complexity $O(d2^d nmr)$, where n and m are the lengths of the two given input strings and r is the total length of d constraint strings.

In the special case of $d=1$, our solution is cubic. Some solutions are quadratic in the literature, for example, the quadratic algorithms for the STR-IC-LCS problem in Alam and Rahman (2012) and Deorowicz (2012). In the algorithm of Deorowicz (2012), the LCS-computation procedure is used as a component of the algorithm. Two dynamic programming (DP) matrices are computed in the algorithm: the forward matrix and the reverse matrix. The recurrence is exactly as for the LCS computation. In the final stage, the result is established according to the two DP matrices. In the section of improvements and extensions in Deorowicz (2012), the author claimed that the generalization of the LCS problem for many sequences is direct, but the time complexity of the exact algorithm computing the multidimensional DP matrix is $O(2^d n^d)$, where d is the number of sequences of length $O(n)$ each. The STR-IC-LCS problem generalizes in the same way and the worst-case time complexity is also $O(2^d n^d)$. It is not clear how the quadratic algorithm of Deorowicz (2012) can be generalized to solve the M-STR-IC-LCS problem.

Alam and Rahman (2012) claimed to provide a slightly better quadratic algorithm for the STR-IC-LCS problem independently and also a general solution wherein the set of constraint patterns can be handled. The

Table 1. The Generalized Constrained Longest Common Subsequence Problems

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

LCS, longest common subsequence.

Table 2. The Multiple-Generalized Constrained Longest Common Subsequence Problems

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

basic idea of their first quadratic algorithm for the STR-IC-LCS problem is the same as the algorithm of Deorowicz (2012). The forward and the reverse DP matrices are computed first. Then result is established according to the two DP matrices. The only difference between the two algorithms is that while computing each of the occurrences of the string, the unique position is kept in Alam and Rahman (2012) while multiple positions are kept in Deorowicz (2012). Based on the first algorithm, the authors present a dynamic programming formula to directly compute the STR-IC-LCS problem. Moreover, the algorithm can be generalized to solve a restrict M-STR-IC-LCS problem, where an ordered list of d constrained strings is given and the goal is to find an LCS containing each of them as a substring in the order they appear in the list.

A similar dynamic programming formula is given to solve this restrict M-STR-IC-LCS problem, with a computing time $O(nmd)$, where n and m are the lengths of the two given input strings and d is the number of ordered constrained strings. The order of the d constrained strings plays an important role in the generalized algorithm. The quadratic algorithm for the STR-IC-LCS problem can be applied successively to the d constrained strings with their order. For the more general M-STR-IC-LCS problem, if we know the order of the p constrained strings in the solution in advance, we can use the algorithm in Tseng and Yang (2013) to find a solution in $O(nmd)$. But, it is difficult to see how the dynamic programming formula for solving the restrict M-STR-IC-LCS problem can be generalized to the M-STR-IC-LCS problem without an order restriction.

The exponential-time algorithms for solving these two problems were also presented in Chen and Chao (2011). In their article, Chen and Chao (2011) present a property of a solution for the STR-IC-LCS problem with two constrained patterns. Based on this property, they can solve the STR-IC-LCS problem with two constrained patterns in $O(mn\rho_1\rho_2)$ time and $O(mn \cdot \max\{\rho_1\rho_2\})$ space, where ρ_1 and ρ_2 are the lengths of the two constrained patterns, respectively. It is difficult to see how this property can be extended to the cases of more than two constrained patterns. Even though the property can be extended to the cases of more than two constrained patterns, the time cost of the extended algorithm would be $O(mn\prod_{k=1}^{d}\rho_k)$, where ρ_k, $1 \le k \le d$, are the lengths of the d constrained patterns, respectively. Compared with the time complexity of the algorithm presented in this article, the factor $\prod_{k=1}^{d}\rho_k$ will be reduced to $d2^d\sum_{k=1}^{d}\rho_k$. The big difference between the product and the sum of d positive integers is evident, especially in the case of d, the number of constrained patterns, being a constant.

The organization of the article is as follows.

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

In Section 2, preliminary knowledge for presenting our algorithm for the M-STR-IC-LCS problem is discussed. In Section 3, we give a new dynamic programming solution for the M-STR-IC-LCS problem with time complexity $O(d2^dnmr)$, where n and m are the lengths of the two given input strings, and r is the total length of d constraint strings. In Section 4, we discuss the issues to implement the algorithm efficiently. Some concluding remarks are provided in Section 5.

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 $X = x_1x_2 \cdots x_n$ 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 positions i to j can be denoted as $X[i : j] = x_ix_{i+1} \cdots x_j$. If $i \neq 1$ or $j \neq n$, then the substring $X[i : j] = x_ix_{i+1} \cdots x_j$ is called a proper substring of X. A substring $X[i : j] = x_ix_{i+1} \cdots x_j$ is called a prefix or a suffix of X if $i = 1$ or $j = n$, respectively.

For the two input sequences $X = x_1x_2 \cdots x_n$ and $Y = y_1y_2 \cdots y_m$ 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-IC-LCS is to find an LCS of X and Y including each of constraint $P_i \in P$ as a substring.

Keyword tree (Aho–Corasick automaton; Aho and Corasick, 1975) is a main data structure in our dynamic programming algorithm to process the constraint set P of the M-STR-IC-LCS problem.

Definition 1. *The keyword tree for set* P *is a rooted directed tree* T *satisfying three 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.

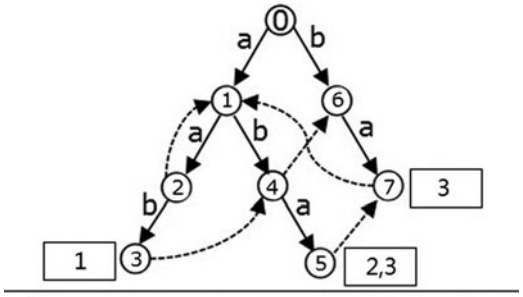
To identify the nodes of T, we assign numbers $0, 1, \cdots, t-1$ to all t nodes of T in their preorder numbering. Then, each node will be assigned an integer i, $0 \leq i < t$, as shown in Figure 1. For each node numbered i of a keyword tree T, the concatenation of characters on the path from the root 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, Figure 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 consists of three functions, a goto function, an output function, and a failure function. The goto function is represented 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 that indicates 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 nodes 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 KMP algorithm for solving the string matching problem. It is represented by the dashed edges in Figure 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. For each node i of T, if A is a suffix of string $L(i)$ in length $lp(i)$, then there must be a 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 Figure 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 Aho and Corasick (1975), for a constant alphabet size, in the worst case, the failure function pre can be computed in $O(r)$ time.



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

FIG. 1. Keyword trees.

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 failure list of a given node is the ordered list of the nodes that locate on the path to the root through 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 $\{7 \rightarrow 1 \rightarrow 0\}$, and the failure list of node 6 is $\{0\}$, as shown in Figure 1.

The failure function *pre* is used to speed up 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 directs 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.

Definition 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 that 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 Figure 1.

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

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 $\leftrightarrow \delta$ of T, then the search will stop in a node i of T. All such common subsequences of X and Y can be classified into a group i, $0 \le i < t$. These t groups are still not sufficient to distinguish the different states in our dynamic programming algorithm, since the common subsequence of X and Y in the same group may contain different subsets of P. Therefore, we must divide each group into 2^d new states by attaching d flags to denote the combinations that constraints have been kept. The d flags can be recorded by a d bits vector s. If the string $P_j \in P$ is kept, then the bit j of s is set to 1, otherwise 0. There are total 2^d different such bit vectors, denoted as $s_0, s_1, \cdots, s_{2^d-1}$ as follows.

Definition 3.

- *Let* $0 \le j < 2^d$ *and* $j=\sum_{i=1}^{d} b_i 2^{i-1}$. *Then the set* s_j *is defined as* $s_j=\{i | b_i=1, 1 \le i \le d\}$. *Inversely, let* $s=\{k_1, k_2, \cdots, k_p\}$, *where* $1 \le k_1 < k_2 < \cdots < k_p \le d$. *Then, the set* s *can be mapped into an integer* $j=g(s)=\sum_{i=1}^{p} 2^{k_i-1}$, *and* $s=s_j$.
- *If a subset of strings* $q=\{P_{k_1}, P_{k_2}, \cdots, P_{k_h}\} \subseteq P$ *must be added to the set* s_j, *then the set* s_j *becomes* s_k, *where* $k=j \vee \sum_{i=1}^{h} 2^{k_i-1}$, *and the operation* $\vee$ *is a bitwise or operation of two integers. In this case we denote* $s_k = s_j \bigcup_q$.
- *For a sequence* z *with state* (α, β) *in a given keyword tree* T, *and a character* $c \in \sum$, *we now consider the state of the sequence* $\bar{z}=z \oplus c$ in T. *From the node* α, *the search for* $\bar{z}$ *will go to node* $\bar{\alpha}=\delta(\alpha, c)$. *If* $O_{\bar{\alpha}}$, *the output set of the node* $\bar{\alpha}$ *is not empty, then the strings of* $O_{\bar{\alpha}}$ *must be included in the sequence* $\bar{z}$, *and thus the set* s_β *will be changed to* $s_\beta \bigcup O_{\bar{\alpha}}$. *In this case, the set* $s_\beta \bigcup O_{\bar{\alpha}}$ *can be mapped into an integer* $\bar{\beta}=g(s_\beta \bigcup O_{\bar{\alpha}})$. *We denote this integer as* $\bar{\beta}=\gamma(\alpha, \beta, c)$. *In other words, the state of the* $\bar{z}=z \oplus c$ *in* T *becomes* $(\delta(\alpha, c), \gamma(\alpha, \beta, c))$.

For example, in the example of Figure 1, we have $d=3$, and $s_1=\{1\}$, $s_6=\{2, 3\}$, $s_7=\{1, 2, 3\}$, and $s_7 = s_1 \bigcup_{s_6}$.

Finally we have $t2^d$ different states in our dynamic programming algorithm. For each pair (α, β), $0 \le \alpha < t, 0 \le \beta < 2^d$, the state (α, β) represents the set of common subsequence of X and Y in group i and the subset of P contained in the subsequence is recorded by bit vector s_β.

Definition 4. *Let $Z(i, j, (\alpha, \beta))$ denote the set of all LCSs of $X[1 : i]$ and $Y[1 : j]$ with state (α, β), where $1 \le i \le n, 1 \le j \le m$, and $0 \le \alpha < t, 0 \le \beta < 2^d$. The length of an LCS in $Z(i, j, (\alpha, \beta))$ is denoted as $f(i, j, (\alpha, \beta))$.*

If we can compute $f(i, j, (\alpha, \beta))$ for any $1 \le i \le n$, $1 \le j \le m$, and $0 \le \alpha < t, 0 \le \beta < 2^d$ efficiently, then the length of an LCS of X and Y including P must be $\max_{0 \le i < t} \{f(n, m, (i, 2^d - 1))\}$.

By using the keyword tree data structure described in the last section, we can give a recursive formula for computing $f(i, j, (\alpha, \beta))$ 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 $Z(i, j, (\alpha, \beta))$ and $f(i, j, (\alpha, \beta))$ be defined as in Definition 4. 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. The label of the node numbered $k (0 \le k < t)$ is denoted as $L(k)$. Then, for any $1 \le i \le n, 1 \le j \le m$, and $0 \le \alpha < t, 0 \le \beta < 2^d$, $f(i, j, (\alpha, \beta))$ can be computed by the following recursive Equation (1).*

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

where

$$S(\alpha, \beta, x_i) = \{(\bar{\alpha}, \bar{\beta}) | 0 \le \bar{\alpha} < t, 0 \le \bar{\beta} < 2^d, \delta(\bar{\alpha}, x_i) = \alpha, \gamma(\bar{\alpha}, \bar{\beta}, x_i) = \beta\} \quad (2)$$

The boundary conditions of this recursive formula are $f(i, 0, (0, 0)) = f(0, j, (0, 0)) = 0$ 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 $0 \le \alpha < t, 0 \le \beta < 2^d$, suppose $f(i, j, (\alpha, \beta)) = l$ and $z = z_1 \cdots z_l \in Z(i, j, (\alpha, \beta))$.

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 $f(i', j', (\alpha, \beta)) \le f(i, j, (\alpha, \beta))$, since a common subsequence z of $X[1 : i']$ and $Y[1 : j']$ with state (α, β) is also a common subsequence of $X[1 : i]$ and $Y[1 : j]$ with state (α, β).

(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 (α, β), and so $f(i-1, j, (\alpha, \beta)) \ge l$. In contrast, $f(i-1, j, (\alpha, \beta)) \le f(i, j, (\alpha, \beta)) = l$. Therefore, in this case we have $f(i, j, (\alpha, \beta)) = f(i-1, j, (\alpha, \beta))$.
(1.2) If $y_j \ne z_l$, then we can prove similarly that in this case, $f(i, j, (\alpha, \beta)) = f(i, j-1, (\alpha, \beta))$.
Combining the two subcases we conclude that in the case of $x_i \ne y_j$, we have

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

(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 (α, β), and so $f(i-1, j-1, (\alpha, \beta)) \ge l$. In contrast, $f(i-1, j-1, (\alpha, \beta)) \le f(i, j, (\alpha, \beta)) = l$. Therefore, in this case we have $f(i, j, (\alpha, \beta)) = f(i-1, j-1, (\alpha, \beta))$.
(2.2) If $x_i = y_j = z_l$, then $f(i, j, (\alpha, \beta)) = l > 0$ and $z = z_1 \cdots z_l$ is an LCS of $X[1 : i]$ and $Y[1 : j]$ with state (α, β).

Let the state of $(z_1, \cdots, z_{l-1})$ be $(\bar{\alpha}, \bar{\beta})$, then we have $(\bar{\alpha}, \bar{\beta}) \in S(\alpha, \beta, x_i)$, 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 $(\bar{\alpha}, \bar{\beta})$. Therefore, we have

$$f(i-1, j-1, (\bar{\alpha}, \bar{\beta})) \ge l - 1.$$

Furthermore, we have

$$\max_{(\bar{\alpha},\bar{\beta})\in S(\alpha,\beta,x_i)}\{f(i-1,j-1,(\bar{\alpha},\bar{\beta}))\}\geq l-1.$$

In other words,

$$f(i,j,(\alpha,\beta))\leq 1+\max_{(\bar{\alpha},\bar{\beta})\in S(\alpha,\beta,x_i)}\{f(i-1,j-1,(\bar{\alpha},\bar{\beta}))\}. \tag{3}$$

In contrast, for any $(\bar{\alpha},\bar{\beta})\in S(\alpha,\beta,x_i)$, and $v=v_1\cdots v_h\in Z(i-1,j-1,(\bar{\alpha},\bar{\beta}))$, $v\oplus x_i$ is a common subsequence of $X[1:i]$ and $Y[1:j]$ with state (α,β). Therefore, $f(i,j,(\alpha,\beta))=l\geq 1+h=1+f(i-1,j-1,(\bar{\alpha},\bar{\beta}))$, and so we conclude that

$$f(i,j,(\alpha,\beta))\geq 1+\max_{(\bar{\alpha},\bar{\beta})\in S(\alpha,\beta,x_i)}\{f(i-1,j-1,(\bar{\alpha},\bar{\beta}))\}. \tag{4}$$

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

$$f(i,j,(\alpha,\beta))=1+\max_{(\bar{\alpha},\bar{\beta})\in S(\alpha,\beta,x_i)}\{f(i-1,j-1,(\bar{\alpha},\bar{\beta}))\}. \tag{5}$$

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

The proof is complete. ■

4. IMPLEMENTATION OF THE ALGORITHM

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

Algorithm 1: M-STR-IC-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 including P

Build a keyword tree T for P
for all i,j, $0\leq i\leq n$, $0\leq j\leq m$ **do**
  $f(i,0,(0,0))\leftarrow 0, f(0,j,(0,0))\leftarrow 0$ {boundary condition}
end for
$S\leftarrow\{(0,0)\}$ {current set of states}
for $i=1$ to n **do**
  for $j=1$ to m **do**
    for each $(\alpha,\beta)\in S$ **do**
      if $x_i\neq y_j$ **then**
        $f(i,j,(\alpha,\beta))\leftarrow\max\{f(i-1,j,(\alpha,\beta)),f(i,j-1,(\alpha,\beta))\}$
      else
        $\bar{\alpha}\leftarrow\delta(\alpha,x_i)$, $\bar{\beta}\leftarrow\gamma(\alpha,\beta,c)$, $s_{\bar{\beta}}\leftarrow s_\beta\bigcup O_{\bar{\alpha}}$
        $f(i,j,(\bar{\alpha},\bar{\beta}))\leftarrow\max\{f(i-1,j-1,(\bar{\alpha},\bar{\beta})),1+f(i-1,j-1,(\alpha,\beta))\}$
        $S\leftarrow S\bigcup\{(\bar{\alpha},\bar{\beta})\}$
      end if
    end for
  end for
end for
return $\max_{0\leq i<t}\{f(n,m,(i,2^d-1)\}$

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. The function $\gamma(\alpha,\beta,c)$ is defined in Definition 3, which can be computed in $O(d)$ time. The variable S is used to record the current states created. If a node

is reached and its output set is not empty, then a new state may be created. The current state set S is extended gradually in the ''for'' loop of Algorithm 1. In the worst case, the set S will have a size of $t2^d = O(2^d r)$, where r is the total lengths of the constrained strings. The body of the triple for loops can be computed in $O(d)$ time in the worst case. Therefore, the total time of Algorithm 1 is $O(d2^d nmr)$. The space used by Algorithm 1 is $O(2^d nmr)$. In case the number of constraint strings is fixed, that is, d is a constant, our new algorithm for the M-STR-IC-LCS problem requires $O(nmr)$ time and space.

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 including P_j must also include P_i. For this reason, the constraint string P_i can be removed from constraint set P without changing the solution of the problem. Without loss of generality, we can make the following two assumptions on the constraint set P.

Assumption 1. *There are no 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 (Aho and Corasick, 1975). For example, in Figure 1, there is a directed path of failure links from nodes 5 to 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 substrings 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 LCS of X and Y including 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 α such that $f(n, m, (\alpha, 2^d - 1))$ gives the length of an LCS of X and Y including P. Then, a function call $back(n, m, (\alpha, 2^d - 1))$ will produce the answer LCS accordingly.

Algorithm 2: $back(i, j, (\alpha, \beta))$

```
1 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,(α,β)) = f(i-1,j-1,(α,β)) then
 6:         back(i-1,j-1,(α,β))
 7:     else
 8:         for each (ᾱ,β̄) ∈ S do
 9:             if α = δ(ᾱ,x_i) and β = γ(ᾱ,β̄,x_i) and f(i,j,(α,β)) = 1+f(i-1,j-1,(ᾱ,β̄)) then
10:                 back(i-1,j-1,(ᾱ,β̄))
11:                 print x_i
12:             end if
13:         end for
14:     end if
15: else if f(i-1,j,(α,β)) > f(i,j-1,(α,β)) then
16:     back(i-1,j,(α,β))
17: else
18:     back(i,j-1,(α,β))
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* $X = x_1x_2 \cdots x_n$ *and* $Y = y_1y_2 \cdots y_m$ *of lengths* n *and* m*, respectively, and a set of* d *constraints* $P = \{P_1, \cdots, P_d\}$ *of total length* r*, Algorithms 1 and 2 solve the M-STR-IC-LCS problem correctly in* $O(d2^dnmr)$ *time and* $O(2^dnmr)$ *space, with preprocessing time* $O(r|\Sigma|)$*. In case the number of constraint strings is fixed, Algorithms 1 and 2 for the M-STR-IC-LCS problem require* $O(nmr)$ *time and space.*

5. CONCLUDING REMARKS

We have suggested a new dynamic programming solution for the new GC-LCS problem M-STR-IC-LCS. The new dynamic programming algorithm requires $O(d2^dnmr)$ time in the worst case. In case the number of constraint strings d is fixed, our new algorithm for the M-STR-IC-LCS problem requires $O(nmr)$ time and space, and thus this is a polynomial time algorithm. If d is not fixed, the time complexity $O(d2^dnmr)$ is still exponential in its expression. It is not clear whether there is an efficient algorithm in this case. We conjecture that our new algorithm is still polynomial even though d is not fixed. We will investigate this issue further.

ACKNOWLEDGMENTS

This work was supported by Fujian Provincial Key Laboratory of Data-Intensive Computing and Fujian University Laboratory of Intelligent Computing and Information Processing.

AUTHOR DISCLOSURE STATEMENT

The authors declare there are no competing financial interests.

REFERENCES

Aho, A., and Corasick, M. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM*. 18, 333–340.

Alam, M.R., and Rahman, M.S. 2012. The substring inclusion constraint longest common subsequence problem can be solved in quadratic time. *J. Discret. Algorithm.* 17, 67–73.

Ann, H., Yang, C., Peng, Y., et al. 2010. Efficient algorithms for the block edit problems. *Inf. Comput*. 208, 221–229.

Ann, H., Yang, C., Tseng, C., et al. 2008. A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. *Inform. Process. Lett*. 108, 360–364.

Apostolico, A., and Guerra, C. 1987. The longest common subsequences problem revisited. *Algorithmica*. 2, 315–336.

Arslan, A., and Egecioglu, O. 2005. Algorithms for the constrained longest common subsequence problems. *Int. J. Found. Comput. Sci*. 16, 1099–1109.

Blum, C., Blesa, M., and Lpez-Ibnez, M. 2009. Beam search for the longest common subsequence problem. *Comput. Oper. Res*. 36, 3178–3186.

Chen, Y., and Chao, K. 2011. On the generalized constrained longest common subsequence problems. *J. Comb. Optim*. 21, 383–392.

Chin, F., Santis, A., Ferrara, A., et al. 2004. A simple algorithm for the constrained sequence problems. *Inform. Process. Lett*. 90, 175–179.

Deorowicz, S. 2012. Quadratic-time algorithm for a string constrained LCS problem. *Inform. Process. Lett*. 112, 423–426.

Deorowicz, S., and Obstoj, J. 2010. Constrained longest common subsequence computing algorithms in practice. *Comput. Inform*. 29, 427–445.

Hirschberg, D. 1977. Algorithms for the longest common subsequence problem. *J. ACM*. 24, 664–675.

Iliopoulos, C., and Rahman, M. 2008. New efficient algorithms for the LCS and constrained LCS problems. *Inform. Process. Lett*. 106, 13–18.

Iliopoulos, C., and Rahman, M. 2009. A new efficient algorithm for computing the longest common subsequence. *Theor. Comput. Sci*. 45, 355–371.

Knuth, D., Morris, J., and Pratt, V. 1977. Fast pattern matching in strings. *SIAM J. Comput*. 6, 323–350.

Peng, Y., Yang, C., and Huang, K. 2010. An algorithm and applications to sequence alignment with weighted constraints. *Int. J. Found. Comput. Sci*. 21, 51–59.

Tsai, Y. 2003. The constrained longest common subsequence problem. *Inform. Process. Lett*. 88, 173–176.

Tseng, C., and Yang, C. 2013. Efficient algorithms for the longest common subsequence problem with sequential substring constraints. *J. Complexity*. 29, 44–52.

Wagner, R., and Fischer, M. 1974. The string-to-string correction problem. *J. ACM*. 21, 168–173.

Wang, L., Wang, X., Wu, Y., et al. 2013. A dynamic programming solution to a generalized LCS problem. *Inform. Process. Lett*. 113, 723–728.

Address correspondence to:
Prof. Daxin Zhu
School of Mathematics and Computer Science
Quanzhou Normal University
Quanzhou 362000, China

E-mail: dex@qztc.edu.cn