Faster STR-EC-LCS Computation

Kohei Yamada[1]($\boxtimes$), Yuto Nakashima[1], Shunsuke Inenaga[1,2], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Fukuoka, Japan
`{kohei.yamada,yuto.nakashima,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp`
[2] PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Abstract. The longest common subsequence (LCS) problem is a central problem in stringology that finds the longest common subsequence of given two strings A and B. More recently, a set of four constrained LCS problems (called generalized constrained LCS problem) were proposed by Chen and Chao [J. Comb. Optim, 2011]. In this paper, we consider the substring-excluding constrained LCS (STR-EC-LCS) problem. A string Z is said to be *an STR-EC-LCS of two given strings A and B excluding P* if, Z is one of the longest common subsequences of A and B that does not contain P as a substring. Wang et al. proposed a dynamic programming solution which computes an STR-EC-LCS in $O(mnr)$ time and space where $m = |A|, n = |B|, r = |P|$ [Inf. Process. Lett., 2013]. In this paper, we show a new solution for the STR-EC-LCS problem. Our algorithm computes an STR-EC-LCS in $O(n|\Sigma| + (L+1)(m-L+1)r)$ time where $|\Sigma| \leq \min\{m, n\}$ denotes the set of distinct characters occurring in both A and B, and L is the length of the STR-EC-LCS. This algorithm is faster than the $O(mnr)$-time algorithm for short/long STR-EC-LCS (namely, $L \in O(1)$ or $m - L \in O(1)$), and is at least as efficient as the $O(mnr)$-time algorithm for all cases.

1 Introduction

The *longest common subsequence* (*LCS*) problem of finding an LCS of given two strings, is a classical and important problem in Theoretical Computer Science. Given two strings A and B of respective lengths m and n, it is well known that the LCS of A and B can be computed by a standard dynamic programming technique [13]. Since LCS is one of the most fundamental similarity measures for string comparison, there are a number of studies on faster computation of LCS and its applications [2,3,11,14]. It is also known that there is a conditional lower bound which states that the LCS of two strings of length n each cannot be computed in $O(n^{2-\epsilon})$ time for any constant $\epsilon > 0$, unless the famous popular Strong Exponential Time Hypothesis (SETH) fails [1]. Thus, it is highly likely that one needs to use almost quadratic time for computing LCS in the worst case. Still, it is possible to design algorithms for computing LCS whose running time depends on other parameters. One of such algorithms was proposed by Nakatsu et al. [10], which finds an LCS of given two strings A and B in $O(n(m-l))$

time and space, where l is the length of the LCS of the two given strings. This algorithm is efficient when l is large, namely, A and B are very similar.

Of a variety of extensions to LCS that have been extensively studied, this paper focuses on a class of problems called the *constrained LCS* problems, first considered by Tsai [12]. We are given strings A, B and constraint string P of length r, and the CLCS problem is to find a longest subsequence common to A and B, such that the subsequence has P as a subsequence. He also presented a dynamic programming algorithm which solves the problem in $O(m^2n^2r)$ time and space. The motivation for introducing constraints is to reflect some a-priori knowledge (e.g., biological knowledge) to the solutions. Later, the *generalized constrained LCS* (*GC-LCS*) problems were introduced by Chen et al. [4]. GC-LCS consists of four variants of the constrained LCS problem, which are respectively called *SEQ-IC-LCS*, *SEQ-EC-LCS*, *STR-IC-LCS*, and *STR-EC-LCS*. For given strings A, B and P, the problem is to find a longest subsequence common to A and B such that the subsequence includes/excludes/includes/excludes P as a subsequence/subsequence/substring/substring, respectively for SEQ-IC-LCS/SEQ-EC-LCS/STR-IC-LCS/STR-EC-LCS. We remark that CLCS is the same as SEQ-IC-LCS. The best known results for these problems were proposed in [4–6,15].

The quadratic bound for STR-IC-LCS seems to be very difficult to improve, since STR-IC-LCS is a special case of LCS (recall the afore-mentioned conditional lower bound for LCS). Since the other three variants require cubic time, it is important to discover more efficient solutions for these problems. There exist faster dynamic programming solutions for SEQ-IC-LCS and STR-IC-LCS which are based on run-length encodings [8,9]. However, no faster solutions to STR-EC-LCS than the one with $O(mnr)$ running time [15] are known to date.

In this paper, we revisit the STR-EC-LCS problem. More formally, we say that a string Z is *an STR-EC-LCS of two given strings A and B excluding* P if, Z is one of the longest common subsequences of A and B that does not contain P as a substring. We show a new dynamic programming solution for the STR-EC-LCS problem which runs in $O(n|\Sigma| + (L + 1)(m - L + 1)r)$ time and space, where Σ is the set of distinct characters occurring in both A and B, and L is the length of the solution. Note that $|\Sigma| \leq \min\{m, n\}$ always holds. Our algorithm is built on Nakatsu et al.s' method for the (original) LCS problem [10]. Assume w.l.o.g. that $m \leq n$. When the length of STR-EC-LCS is quite short or long (namely, $L \in O(1)$ or $m - L \in O(1)$), our algorithm runs only in $O(n|\Sigma| + mr) = O((n + r)m) = O(nm)$ time and space, since $r \leq n$. Even in the worst case where $L \in \Theta(m)$ and $m - L \in \Theta(m)$, which happens when $L = cm$ for any constant $0 < c < 1$, our algorithm is still as efficient as $O(mnr)$ since $|\Sigma| \leq \min\{m, n\}$.

This paper is organized as follows; we will give notations which we use in this paper in Sect. 2, we will propose our dynamic programming solution for the STR-EC-LCS problem in Sect. 3, finally, we will explain our algorithm for the STR-EC-LCS in Sect. 4.

2   Preliminaries

2.1   Strings

Let Σ be an integer *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$.

A string Z is a *subsequence* of A if Z can be obtained from A by removing zero or more characters. In this paper, we consider common subsequences of two strings A and B of respective lengths m and n. For this sake, we can perform a standard preprocessing on A and B that removes every character that occurs only in either A or B, because such a character is never contained in any common subsequences of A and B. Assuming $n \geq m$, this preprocessing can be done in $O(n \log n)$ time with $O(n)$ space for general ordered alphabets, and in $O(n)$ time and space for integer alphabets of polynomial size in n (c.f. [7]). In what follows, we consider the latter case of integer alphabets, and assume that A and B have been preprocessed as above. In the sequel, let Σ denote the set of distinct characters that occur in both A and B. Note that $|\Sigma| \leq \min\{m, n\} = m$ holds.

2.2   STR-EC-LCS

Let A, B and P be strings. A string Z is said to be *an STR-EC-LCS of two given strings A and B excluding* P if, Z is one of the longest common subsequences of A and B that does not contain P as a substring. For instance, `bcaac`, `bcaba`, `acaac`, `acaba`, `abaac` and `ababa` are STR-EC-LCS of $A = $ `abcabac` and $B = $ `acbcaacbaa` excluding $P = $ `abc`. Although `abcaba` and `abcaac` are longest common subsequences of A and B, they are not STR-EC-LCS of the same strings (since they have P as a substring).

In Sect. 3, we revisit the STR-EC-LCS problem defined as follows.

Problem 1 (STR-EC-LCS problem [4]*).* Given strings A, B, and P, compute an STR-EC-LCS (and/or its length) of given strings.

In the rest of the paper, m, n, and r respectively denote the length of A, B and P. It is easy to see that STR-EC-LCS problem is the same as LCS problem when $r > \min\{m, n\}$. We assume that $r \leq m \leq n$ without loss of generality.

3   Dynamic Programming Solution for the STR-EC-LCS Problem

Our aim of this section is to show our dynamic programming solution for the STR-EC-LCS problem. We first give short descriptions of a dynamic programming solution for the LCS problem proposed by Nakatsu et al. [10], and a

s\i	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	*	2	2	1	1	1	1	1
2	*	*	3	3	2	2	2	2
3	*	*	*	4	4	4	3	3
4	*	*	*	*	7	6	6	4
5	*	*	*	*	*	*	7	7
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

Fig. 1. This is an example for table e of given strings $A = \texttt{aabacab}$ and $B = \texttt{baabbcaa}$. For the sake of visibility, the value $n + 1 = 9$ is replaced by asterisk ($*$). The last row in the table which has a value smaller than $n + 1$ is 5; that is, the length of an LCS of A and B is 5.

dynamic programming solution for the STR-EC-LCS problem proposed by Wang et al. [15].

3.1 Solution for LCS by Nakatsu et al.

Nakatsu et al. proposed a dynamic programming solution for computing an LCS of given strings A and B. Here, we give a slightly modified description of their solution in order to describe our algorithm. For any $0 \le i, s \le m$, let $e(i, s)$ be the length of the shortest prefix $B[1..e(i, s)]$ of B such that the length of the longest common subsequence of $A[1..i]$ and $B[1..e(i, s)]$ is s. For convenience, $e(i, s) = n + 1$ if no such prefix exists or if $s > i$ holds. The values $e(i, s)$ will be computed using dynamic programming, where i represents the column number, and s represents the row number. Let $\tilde{s}$ be the largest value such that $e(i, s) < n + 1$ for some i, i.e, $\tilde{s}$ is the last row in the table of e, which has a value smaller than $n + 1$. We can see that the length of the longest common subsequence of A and B is $\tilde{s}$. We give an example in Fig. 1.

Now we explain how to compute e efficiently. Assume that $e(i - 1, s)$ and $e(i - 1, s - 1)$ have already been computed. We consider $e(i, s)$. It is easy to see that $e(i, s) \le e(i - 1, s)$. If $e(i, s) < e(i - 1, s)$, an LCS of $A[1..i]$ and $B[1..e(i, s)]$ must use the character $A[i]$ as the last character. Then, we can see that $e(i, s)$ is the index of the leftmost occurrence of $A[i]$ in $B[e(i - 1, s - 1) + 1..n]$. Let $j_{i,s}$ be the the index of the leftmost occurrence of $A[i]$ in $B[e(i - 1, s - 1) + 1..n]$. From these facts, the following recurrence formula holds for e:

$$e(i, s) = \min\{e(i - 1, s), j_{i,s}\}.$$

If we add more information, we can backtrack on the table in order to compute an LCS (as a string), and not just its length.

3.2   Solution for STR-EC-LCS by Wang et al.

Wang et al. proposed a dynamic programming solution for STR-EC-LCS problem of given strings A, B and P. Here, we describe a key idea of their solution.

Definition 1. *For any string S, $\sigma(S)$ is the length of the longest prefix of P which is a suffix of S.*

By using this notation, they considered a table f defined as follows: let $f(i, j, k)$ be the length of the longest common subsequence Z of $A[1..i]$ and $B[1..j]$ such that Z does not have P as a substring and $\sigma(Z) = k$. They also showed a recurrence formula for f. By the definition of f, the length of an STR-EC-LCS is $\max\{f(m, n, t) \mid 0 \le t < r\}$.

3.3   Our Solution for STR-EC-LCS

Our solution is based on the idea of Sect. 3.1. We maintain occurrences of a prefix of P as a suffix of a common subsequence by using the idea of Sect. 3.2.

For convenience, we introduce the following notation.

Definition 2. *A string Z is said to satisfy* Property(i, s, k) *if*

– *Z is a subsequence of $A[1..i]$,*
– *Z does not have P as a substring,*
– *$|Z| = s$, and*
– *$\sigma(Z) = k$.*

Thanks to the above notation, we can simply introduce our table d for computing STR-EC-LCS as follows. Let d be a 3-dimensional table where $d(i, s, k)$ is the length of the shortest prefix $B[1..d(i, s, k)]$ of B such that there exists a subsequence which satisfies Property(i, s, k) (if no such subsequence exists, then $d(i, s, k) = n + 1$ for convenience).

We can obtain the following observation about the length of an STR-EC-LCS by the definition of d.

Observation 1. *Let $\tilde{s}$ be the largest $1 \le s \le m$ such that $d(i, s, k) < n + 1$ for some i and k. $\tilde{s}$ is the length of an STR-EC-LCS by the definition of d.*

We give an example of a table in Fig. 2.

The next lemma shows a recurrence formula for d. We use this lemma for computing the length of a STR-EC-LCS.

Lemma 1.

$$d(i, s, k) = \min(\{d(i - 1, s, k)\} \cup \{j_t \mid 0 \le t < r\})$$

holds, where j_t is the smallest position j in $B[d(i - 1, s - 1, t) + 1..n]$ such that $A[i] = B[j]$, and there exists a string Z which satisfies Property$(i - 1, s - 1, t)$ *and $\sigma(ZA[i]) = k$ (if no such Z exists for t, then $j_t = n + 1$).*

$k = 0$

$s\backslash i$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	*	*	*	1	1	1	1	1
2	*	*	*	4	4	4	4	4
3	*	*	*	*	*	6	6	4
4	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

$k = 1$

$s\backslash i$	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	2	2	2	2	2	2	2
2	*	*	*	*	2	2	2	2
3	*	*	*	*	7	7	7	7
4	*	*	*	*	*	*	7	7
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

$k = 2$

$s\backslash i$	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	3	3	3	3	3	3
3	*	*	*	*	*	*	3	3
4	*	*	*	*	*	*	8	8
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

Fig. 2. This is our table d for given strings $A = \texttt{aabacab}$, $B = \texttt{baabbcaa}$, and $P = \texttt{aab}$. In this figure, the value $n + 1 = 9$ is replaced by asterisk $(*)$ for convenience. The lowest row which has a value smaller than $n + 1 = 9$ is $\tilde{s} = 4$. Thus, the length of a STR-EC-LCS is 4.

*Proof.* We show the following inequations to prove this lemma;

1. $d(i, s, k) \leq \min(\{d(i - 1, s, k)\} \cup \{j_t \mid 0 \leq t < r\})$,
2. $d(i, s, k) \geq \min(\{d(i - 1, s, k)\} \cup \{j_t \mid 0 \leq t < r\})$.

We start from proving the first inequation. By the definition of d, $d(i, s, k) \leq d(i - 1, s, k)$ always holds. If $\{j_t \mid 0 \leq t < r\} = \emptyset$, then the first inequation holds. We assume that $\{j_t \mid 0 \leq t < r\} \neq \emptyset$, and j_{t_1} is in the set $(0 \leq t_1 < r)$. Then, there exists a subsequence Z_1 of $B[1..d(i - 1, s - 1, t_1)]$ which satisfies Property$(i - 1, s - 1, t_1)$. Since $A[i] = B[j_{t_1}]$ and $j_{t_1} > d(i - 1, s - 1, t_1)$, $Z_1 A[i]$ is a subsequence of $B[1..j_{t_1}]$ that satisfies Property(i, s, k) and $\sigma(Z_1 A[i]) = k$. This implies that $d(i, s, k) \leq j_{t_1}$. Thus, the first inequation holds.

Suppose that the second inequation does not hold, namely,

$$d(i, s, k) < \min(\{d(i - 1, s, k)\} \cup \{j_t \mid 0 \leq t < r\}) \tag{1}$$

holds. If $d(i, s, k) = n + 1$, then the above inequation does not hold. Now we consider the case $d(i, s, k) < n + 1$. By the definition of d, there exists a subsequence Z_2 of $B[1..d(i, s, k)]$ that satisfies Property(i, s, k). Let $Z_2' = Z_2[1..|Z_2| - 1]$. Then, Z_2' is a length $s - 1$ subsequence of $A[1..i - 1]$ which does not have P as a substring. Since Z_2' satisfies Property$(i - 1, s - 1, \sigma(Z_2'))$, $d(i - 1, s - 1, \sigma(Z_2')) < d(i, s, k)$ holds. Moreover, $\sigma(Z_2' B[d(i, s, k)]) = k$ holds. If $A[i] = B[d(i, s, k)]$, then, $j_{\sigma(Z_2')} \leq d(i, s, k)$ holds. This fact contradicts Inequation (1). Now we can assume that $A[i] \neq B[d(i, s, k)]$. This implies that Z_2 is a common subsequence of $A[1..i]$ and $B[1..d(i, s, k) - 1]$, or a common subsequence of $A[1..i - 1]$ and $B[1..d(i, s, k)]$. The first case implies a contradiction by the definition of d. The second case implies that $d(i, s, k) = d(i - 1, s, k)$, a contradiction. Thus, $d(i, s, k) \geq \min(\{d(i - 1, s, k)\} \cup \{j_t \mid 0 \leq t < r\})$ holds. $\square$
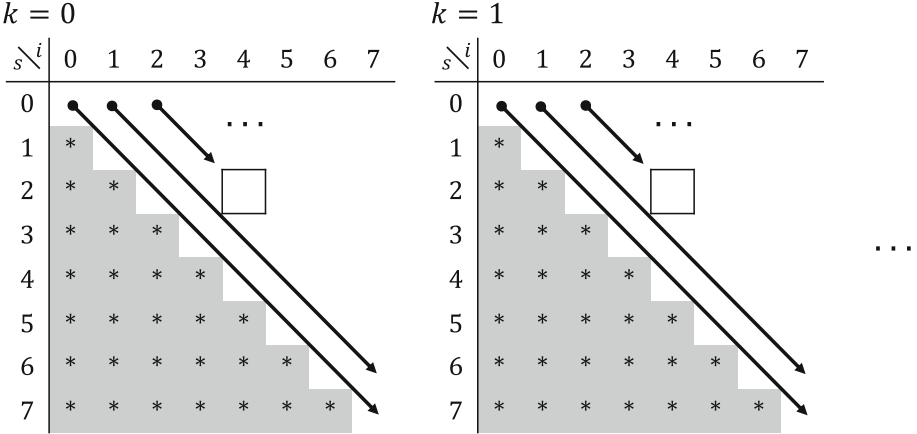
Fig. 3. This figure shows the order of computation for table d. For each table (i.e., for each k), we do not need to compute the lower left part (satisfying $s > i$). We start from computing values on the leftmost arrow for each table. In each step (i, s), we compute $d(i, s, k)$ for all tables (for instance, squared values in the figure will be computed in the same step).

4 Algorithm

In this section, we show how to compute STR-EC-LCS by using Lemma 1. We mainly explain our algorithm to compute the length of an STR-EC-LCS (we will explain how to compute an STR-EC-LCS at the end of this section).

To use Lemma 1, we need $d(i-1, s, k)$ and $d(i-1, s-1, t)$ for all $0 \le t < r$ for computing $d(i, s, k)$. We compute our table for every diagonal line from upper left to lower right in left-to-right order. In each step of our algorithm, we will fix $0 \le i, s \le m$ (we use (i, s) to denote the step for fixed i and s). Then we compute $d(i, s, k)$ for any $0 \le k < r$ in the step. We can see from a simple observation that $d(i, s, k) = n + 1$ holds for any input strings if $i < s$ (since no STR-EC-LCS of length s exists). Thus, we do not compute $d(i, s, k)$ explicitly such that $i < s$. We also describe this strategy in Fig. 3.

Now we consider how to compute $d(i, s, k)$ for any $0 \le k < r$. Let $Z(i, s, k)$ be a subsequence of $B[1..d(i-1, s-1, k)]$ satisfying Property$(i-1, s-1, k)$. Due to Lemma 1, string $Z(i, s, k)A[i]$ is a witness for value $d(i, s, \sigma(Z(i, s, k)A[i]))$ if a (leftmost) position j in $B[d(i-1, s-1, k)+1..n]$ such that $A[i] = B[j]$ exists. For any i, s, k, let $J(i, s, k)$ denote the position j described above. Thus, we can compute $d(i, s, k)$ for any k in step (i, s) as follows.

1. Set $d(i-1, s, k)$ as the initial value for $d(i, s, k)$ for each k.
2. Compute $J(i, s, k)$ and $\sigma(Z(i, s, k)A[i])$ for each k.
3. If $J(i, s, k) < d(i, s, \sigma(Z(i, s, k)A[i]))$, then update $d(i, s, \sigma(Z(i, s, k)A[i]))$ to $J(i, s, k)$.

Lemma 1 and the above discussion ensure the correctness of this algorithm. Next we show how to do these operations efficiently. We use the following two data structures.

Definition 3. *For any position j in B (i.e., $j \in [1, n]$) and any character $\alpha \in \Sigma$,*

$$\mathsf{next}_B(j, \alpha) = \min\{q \mid B[q] = \alpha, q \geq j\}.$$

Definition 4. *For any position t in P (i.e., $t \in [0, r-1]$) and any character $\alpha \in \Sigma$,*

$$\mathsf{next}_\sigma(t, \alpha) = \sigma(P[1..t]\alpha).$$

At the second operation, we need to compute $J(i, s, k)$. $J(i, s, k)$ is the index of the leftmost occurrence of $A[i]$ in $B[d(i-1, s-1, k)+1..n]$. We can compute the occurrence by using next_B, namely, $J(i, s, k) = \mathsf{next}_B(d(i-1, s-1, k)+1, A[i])$.

Moreover, we need to compute $\sigma(Z(i, s, k)A[i])$. We know that $\sigma(Z(i, s, k)) = k$, namely, $Z(i, s, k)$ has $P[1..k]$ as a suffix. By the definition of $\sigma(\cdot)$, $\sigma(S) + 1 \geq \sigma(S\alpha)$ holds for any string S and $\alpha \in \Sigma$. This implies that $\sigma(Z(i, s, k)A[i]) = \sigma(P[1..t]A[i])$. Thus, we can compute $\sigma(Z(i, s, k)A[i])$ by using $\mathsf{next}_\sigma(\cdot)$, namely, $\sigma(Z(i, s, k)A[i]) = \sigma(P[1..t]A[i]) = \mathsf{next}_\sigma(t, A[i])$.

We can easily compute next_B in linear time and space (we give a pseudo-code in Algorithm 1). next_σ was introduced in [15] (as table λ). They also showed that this table can be computed in linear time and space (we give a pseudo-code in Algorithm 2).

Algorithm 1: Construction for next_B

Input: String B of length n, Alphabet Σ
Output: next_B

1 **foreach** *character* $\alpha \in \Sigma$ **do** $\mathsf{next}_B(n, \alpha) = n + 1$;
2 **for** $j = n - 1$ **to** 0 **do**
3 $\quad$ **foreach** $\alpha \in \Sigma$ **do**
4 $\quad\quad$ **if** $\alpha = B[j+1]$ **then** $\mathsf{next}_B(j, \alpha) = j + 1$;
5 $\quad\quad$ **else** $\mathsf{next}_B(j, \alpha) = \mathsf{next}_B(j+1, \alpha)$;

6 **return** next_B

We have finished describing how to compute d. This algorithm computes $O(m^2 r)$ values (i.e., the size of the table d). We can see that every operation can be done in constant time. Thus, this algorithm takes $O(n|\Sigma| + m^2 r)$ time and space. This complexity is similar to Wang et al.s' result (algorithm described in Sect. 3.2). We can modify our algorithm to compute d more efficiently by using the following two observations.

Observation 2. *Assume that we have already computed table d until the i-th diagonal line (i.e., the diagonal line which has $d(i, 0, \cdot)$). Let s' be the lowest row*

Algorithm 2: Construction for next_σ

Input: String P of length r, Alphabet Σ
Output: next_σ

1 $kmp(0) \leftarrow -1$;
2 $kmp(1) \leftarrow 0$;
3 $k \leftarrow 0$;
4 for $i = 2$ **to** r **do**
5   **while** $k \geq 0$ *and* $P[k+1] \neq P[i]$ **do** $k \leftarrow kmp(k)$;
6 $k \leftarrow k + 1$;
7 $kmp(i) \leftarrow k$;

8 $\mathsf{next}_\sigma(0, P[1]) \leftarrow 1$;
9 foreach $\alpha \in \Sigma - \{P[1]\}$ **do**
10 $\mathsf{next}_\sigma(0, \alpha) \leftarrow 0$;

11 for $k = 1$ **to** $r - 1$ **do**
12   **foreach** $\alpha \in \Sigma$ **do if** $\alpha = P[k+1]$ **then** $\mathsf{next}_\sigma(k, \alpha) \leftarrow k + 1$;
13   **else** $\mathsf{next}_\sigma(k, \alpha) \leftarrow \mathsf{next}_\sigma(\mathrm{kmp}(k), \alpha)$;

14 return next_σ

---

which has a value smaller than $n + 1$. Then, we do not need to compute the last $s' + 1$ diagonal lines since these diagonal lines do not make better candidates for STR-EC-LCS.

Observation 3. *If $d(i, s, k) = n + 1$ for all k, then $d(i + 1, s + 1, k) = \ldots = d(i + (m - i), s + (m - i), k) = n + 1$ holds for any k.*

Thanks to the above observations, the number of values which we need to compute is $O((L + 1)(m - L + 1)r)$ where L is the length of STR-EC-LCS (see also Fig. 4).

Finally, we discuss how to store d. We consider computing the i-th diagonal line (i.e., $d(i, 0, k), \ldots, d(i + (m - i), m - i, k))$. Suppose that $d(i, 0, k), \ldots, d(i + t - 1, t - 1, k)$ have already been computed. Then, we store these values by using an array of size $2^{\lceil \log t \rceil}$. If the array filled with values for the line (i.e., $d(i + 2^{\lceil \log t \rceil} - 1, 2^{\lceil \log t \rceil} - 1, k) < n + 1$ for some k), we make new array of size $2^{\lceil \log t \rceil + 1}$ for values $d(i, 0, k), \ldots, d(i + 2^{\lceil \log t \rceil + 1} - 1, 2^{\lceil \log t \rceil + 1} - 1, k)$ on the line. By Observation 3, we will compute at most $L + 2$ values for each line, the total length of arrays for each line is $O(L)$, where L is the length of an STR-EC-LCS. Therefore, we can compute the length of an STR-EC-LCS in $O(n|\Sigma| + (L + 1)(m - L + 1)r)$ time and space.
Computing an STR-EC-LCS. If we want to compute an STR-EC-LCS, we store a pair (s', k') for every $d(i, s, k)$. The pair (s', k') represents that $d(i, s, k)$ was given by $d(i - 1, s', k')$. By using these information, we can compute an STR-EC-LCS from right to left. We show an example in Fig. 5.

Since we can store (s', k') in constant time and space for each $d(i, s, k)$, and compute an STR-EC-LCS in $O(m)$ time, we can get the following main result.

$$O(m - L') \qquad O(L')$$
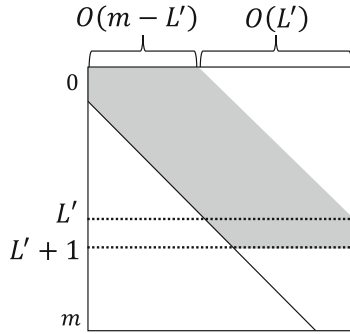
Fig. 4. This is a table for some k. Due to Observations 2 and 3, we do not need to compute values in white part (there might exist positions which do not need their values). The maximum number of values which we need to compute (namely, the total area of the r gray parts) is $O((L+1)(m-L+1)r)$.

$k = 0$

$s\backslash i$	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	*	*	*	1				
2	*	*	*	4	4			
3	*	*	*	*	*	6		
4	*	*	*	*			*	*
5	*	*	*	*	*			*
6	*	*	*	*	*	*		
7	*	*	*	*	*	*	*	

$k = 1$

$s\backslash i$	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	*	2	2	2			
2	*	*	*	*	2			
3	*	*	*	*	7	7		
4	*	*	*	*			*	7
5	*	*	*	*	*			*
6	*	*	*	*	*	*		
7	*	*	*	*	*	*	*	

$k = 2$

$s\backslash i$	0	1	2	3	4	5	6	7
0	*	*	*	*	*	*	*	*
1	*	*	*	*				
2	*	*	3	3	3			
3	*	*	*	*	*	*		
4	*	*	*	*		*	8	
5	*	*	*	*	*			*
6	*	*	*	*	*	*		
7	*	*	*	*	*	*	*	

Fig. 5. In this figure, an arrow represents additional information for backtracking. For instance, $d(6, 4, 1) = 7$ was given by $d(5, 3, 0) = 6$ while computing d. We can get an STR-EC-LCS abca of $A = $ aabacab, $B = $ baabbcaa, and $P = $ aab.

Theorem 1. *For given strings A, B and P, we can compute an STR-EC-LCS in $O(n|\Sigma| + (L+1)(m-L+1)r)$ time and space where m, n, r and L are the length of A, B, P and the STR-EC-LCS, respectively.*

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. FOCS **2015**, 59–78 (2015)
2. Ahsan, S.B., Aziz, S.P., Rahman, M.S.: Longest common subsequence problem for run-length-encoded strings. In: 2012 15th International Conference on Computer and Information Technology (ICCIT), pp. 36–41, December 2012

3. Bunke, H., Csirik, J.: An improved algorithm for computing the edit distance of run-length coded strings. Inf. Process. Lett. **54**(2), 93–96 (1995). http://www.sciencedirect.com/science/article/pii/002001909500005W
4. Chen, Y.C., Chao, K.M.: On the generalized constrained longest common subsequence problems. J. Comb. Optim. **21**(3), 383–392 (2011). https://doi.org/10.1007/s10878-009-9262-5
5. Chin, F.Y., Santis, A.D., Ferrara, A.L., Ho, N., Kim, S.: A simple algorithm for the constrained sequence problems. Inf. Process. Lett. **90**(4), 175–179 (2004). http://www.sciencedirect.com/science/article/pii/S0020019004000614
6. Deorowicz, S.: Quadratic-time algorithm for a string constrained lcs problem. Inf. Process. Lett. **112**(11), 423–426 (2012). http://www.sciencedirect.com/science/article/pii/S0020019012000567
7. Inenaga, S., Hyyrö, H.: A hardness result and new algorithm for the longest common palindromic subsequence problem. Inf. Process. Lett. **129**, 11–15 (2018)
8. Kuboi, K., Fujishige, Y., Inenaga, S., Bannai, H., Takeda, M.: Faster STR-IC-LCS computation via RLE. In: Kärkkäinen, J., Radoszewski, J., Rytter, W. (eds.) 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, 4–6 July 2017, Warsaw, Poland. LIPIcs, vol. 78, pp. 20:1–20:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.CPM.2017.20
9. Liu, J.J., Wang, Y.L., Chiu, Y.S.: Constrained longest common subsequences with run-length-encoded strings. Comput. J. **58**(5), 1074–1084 (2014). https://doi.org/10.1093/comjnl/bxu012
10. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Inf. **18**, 171–179 (1982). https://doi.org/10.1007/BF00264437
11. Stern, H., Shmueli, M., Berman, S.: Most discriminating segment - longest common subsequence (MDSLCS) algorithm for dynamic hand gesture classification. Pattern Recogn. Lett. **34**(15), 1980–1989 (2013). http://www.sciencedirect.com/science/article/pii/S0167865513000512, smart Approaches for Human Action Recognition
12. Tsai, Y.T.: The constrained longest common subsequence problem. Inf. Process. Lett. **88**(4), 173–176 (2003). http://www.sciencedirect.com/science/article/pii/S002001900300406X
13. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974). https://doi.org/10.1145/321796.321811
14. Wang, C., Zhang, D.: A novel compression tool for efficient storage of genome resequencing data. Nucleic Acids Res. **39**(7), e45 (2011). https://doi.org/10.1093/nar/gkr009
15. Wang, L., Wang, X., Wu, Y., Zhu, D.: A dynamic programming solution to a generalized LCS problem. Inf. Process. Lett. **113**(19–21), 723–728 (2013). https://doi.org/10.1016/j.ipl.2013.07.005