Theoretical Computer Science 710 (2018) 29–34

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On finding a longest common palindromic subsequence

Sang Won Bae [a], Inbok Lee [b,*]

[a] Department of Computer Science, Kyonggi University, Suwon, 443-760, Republic of Korea
[b] Department of Software and Computer Engineering, Korea Aerospace University, Goyang, 412-791, Republic of Korea

ARTICLE INFO

Article history:
Received 1 June 2016
Received in revised form 12 January 2017
Accepted 14 February 2017

Keywords:
String algorithm
Longest common palindromic subsequence

ABSTRACT

Recently, Chowdhury et al. [5] proposed *the longest common palindromic subsequence problem*. It is a variant of the well-known LCS problem, which refers to finding a palindromic LCS between two strings T_1 and T_2. In this paper, we present a new $O(n + \mathcal{R}^2)$-time algorithm where $n = |T_1| = |T_2|$ and $\mathcal{R}$ is the number of matches between T_1 and T_2. We also show that the average running time of our algorithm is $O(n^4/|\Sigma|^2)$, where Σ is the alphabet of T_1 and T_2. This improves the previously best algorithms whose running times are $O(n^4)$ and $O(\mathcal{R}^2 \log^2 n \log\log n)$.

1. Introduction

The *longest common subsequence (LCS) problem* is a classical problem in string algorithms, where two strings T_1 and T_2 are given and it is required to find a longest subsequence that appears simultaneously in both strings T_1 and T_2. Previous works on this problem include [2,3,6–9,11]. [3] is a good survey. Chvátal and Sankoff [7] studied the expected length of an LCS of two random strings. A special case for large alphabets was covered in [11]. In [2], it was shown that the worst time complexity of the LCS problem on the comparison-based model must be quadratic unless the alphabet size is fixed. Hirshberg showed that the LCS problem can be solved in linear space [9]. Crochemore, Iliopoulos and Pinzon showed a practically faster algorithm based on bit-vector operations in [6]. Another bit-vector operation approach for a variant of the LCS problem is shown in [8].

Many variants of the LCS problem have been studied. Among them is the *longest common palindromic subsequence* problem which asks to find a longest common subsequence of T_1 and T_2 that is a palindrome. For example, if $T_1 = \texttt{cabbba}$ and $T_2 = \texttt{aabcbab}$, then $\texttt{abba}$ is a longest common palindromic subsequence of T_1 and T_2.

Problem 1. Given two strings T_1 and T_2, report the longest common subsequence of T_1 and T_2 which is a palindrome.

Manacher [13] presented a linear time algorithm reporting all the palindromes which are prefixes of a given string. Apostolico, Breslauer, and Galil [1] showed that the same algorithm could be used to report all the maximal palindromes in a given string, also in linear time. A suffix-tree based algorithm is shown in [10].

The problem has recently been proposed by Chowdhury et al. [5], in which two algorithms were proposed: an $O(n^4)$-time algorithm based on dynamic programming and the other of $O(\mathcal{R}^2 \log^2 n \log\log n)$ time based on an idea from computational geometry, where $n = |T_1| = |T_2|$ and $\mathcal{R}$ is the number of matches between T_1 and T_2. Note that a match is

* Corresponding author.
E-mail address: inboklee@kau.ac.kr (I. Lee).

http://dx.doi.org/10.1016/j.tcs.2017.02.018
0304-3975/© 2017 Elsevier B.V. All rights reserved.

a pair of positions in T_1 and T_2 with the same character. This means that the second algorithm may be slower than the first one in the worst case where $\mathcal{R} = \Theta(n^2)$. In particular, in the second algorithm, Chowdhury et al. reduces the problem to a geometric problem in four-dimensional space, and adopts geometric data structures such as range trees to efficiently solve it.

In this paper, we present an $O(n + \mathcal{R}^2)$ time algorithm for the longest common palindromic subsequence problem. Note that our algorithm strictly outperforms the second algorithm of Chowdhury et al. [5] and its running time is bounded by $O(n^4)$ even in the worst case. In order to achieve this improvement, we start with a recurrence relation being essentially the same as the one of Chowdhury et al. yet in a different form, and apply our new ideas using staircase structures of matches, finally yielding a more efficient recurrence relation and algorithm. This approach borrows some geometric ideas while our algorithm does not need to handle any complicated geometric structure. Hence, compared to previous algorithms, our algorithm is much easier to implement while increasing its performance.

Related problems include the longest common repeat problem. Given a set of strings, we would like to find the longest substring which appears at least twice in each string. Linear time algorithms for the problem were introduced in [12].

2. Preliminaries

Let Σ be the alphabet. $T[i]$ denotes the i-th character of T and $T[i..j]$ is the substring $T[i]T[i+1]\cdots T[j]$. $|T|$ denotes the length of T. T^R, *reverse* of a string T is obtained by reading T from right to left: $T[|T|]T[|T|-1]\cdots T[1]$. A string T is *palindromic* if $T = T^R$.

A *subsequence* of a string is a string obtained by deleting zero or more symbols from the original string. Given two or more strings, a *common subsequence* is a subsequence which is common to each string. A *longest common subsequence (LCS)* is a longest common subsequence among all the common subsequences. Finally, a *longest common palindromic subsequence (LCPS)* is a longest subsequence among all common subsequences that are palindromic.

$LCS(T_1, T_2)$ is the length of an LCS of T_1 and T_2 and $LCPS(T_1, T_2)$ is that of an LCPS of T_1 and T_2.

For simplicity, we assume that T_1 and T_2 are of the same length: $n = |T_1| = |T_2|$. It is straightforward to modify our algorithm to handle the case when $|T_1| \neq |T_2|$. Again, the time complexity depends on the number of matches between T_1 and T_2.

3. Algorithm

Here we present an $O(\mathcal{R}^2)$-time algorithm for computing $LCPS(T_1, T_2)$ of two strings T_1 and T_2 of length n, where $\mathcal{R}$ denotes the number of matches between T_1 and T_2. A *match* between two strings T_1 and T_2 is a pair (a_1, a_2) of indices such that $T_1[a_1] = T_2[a_2]$.

Let $D((a_1, a_2), (b_1, b_2)) = LCPS(T_1[a_1..b_1], T_2[a_2..b_2])$. The $O(n^4)$-time dynamic programming in [5] is based on the following recurrence:

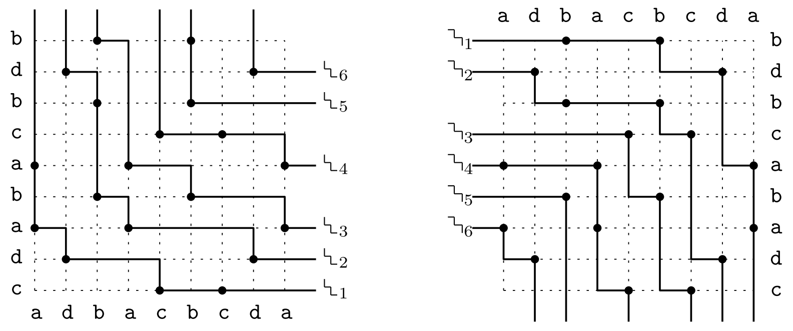
$$D((a_1,a_2),(b_1,b_2)) = \begin{cases} 0 & (a_1 > b_1 \text{ or } a_2 > b_2) \\ 1 + D((a_1+1, a_2+1), (b_1-1, b_2-1)) & \begin{pmatrix} a_1 \le b_1, a_2 \le b_2, \\ T_1[a_1] = T_1[b_1] \\ = T_2[a_2] = T_2[b_2] \end{pmatrix} \\ \max \begin{Bmatrix} D((a_1+1, a_2), (b_1, b_2)), \\ D((a_1, a_2+1), (b_1, b_2)), \\ D((a_1, a_2), (b_1-1, b_2)), \\ D((a_1, a_2), (b_1, b_2-1)) \end{Bmatrix} & (\text{otherwise}). \end{cases}$$

A straightforward dynamic programming based on the above recurrence solves n^4 subproblems. We observe that they may make a difference only when the corresponding pairs (a_1, a_2) and (b_1, b_2) are matches.

Lemma 1. *Let a_1, a_2, b_1, b_2 be indices from 1 to n. For any $1 \le a'_1, a'_2 \le n$ such that there is no match (a''_1, a''_2) with $a_1 \le a''_1 \le a'_1$ and $a_2 \le a''_2 \le a'_2$, we have $D((a_1, a_2), (b_1, b_2)) = D((a'_1, a'_2), (b_1, b_2))$. Symmetrically, for any $1 \le b'_1, b'_2 \le n$ such that there is no match (b''_1, b''_2) with $b_1 \ge b''_1 \ge b'_1$ and $b_2 \ge b''_2 \ge b'_2$, we have $D((a_1, a_2), (b_1, b_2)) = D((a_1, a_2), (b'_1, b'_2))$.*

Proof. This immediately follows from the recurrence relation for D. $\square$

Lemma 1 gives us a hint to reducing the number of subproblems to solve. We implement this using the following partial order on the set of matches. Consider for the moment any set $P \subset \mathbb{R}^2$ of points in the plane. For any $p \in P$, let p_1 and p_2 be the first and second coordinates of p in $\mathbb{R}^2$, namely, $p = (p_1, p_2) \in \mathbb{R}^2$. For any $p, q \in P$, we write $p \prec q$ if and only if $p_1 < q_1$ and $p_2 < q_2$. If $p \prec q$, then we say that p *precedes* q or q *succeeds* p. Also, we write $p \preceq q$ if either $p \prec q$ or $p = q$.



b d b c a b a d c
a d b a c b c d a
$\llcorner_6$ $\llcorner_5$ $\llcorner_4$ $\llcorner_3$ $\llcorner_2$ $\llcorner_1$
a d b a c b c d a
$\urcorner_1$ $\urcorner_2$ $\urcorner_3$ $\urcorner_4$ $\urcorner_5$ $\urcorner_6$
b d b c a b a d c

Fig. 1. Illustration of the two partitions of M when $T_1 = \mathtt{adbacbcda}$ and $T_2 = \mathtt{cdabacbdb}$. In this example, there are $|M| = 20$ matches between T_1 and T_2, depicted as dots in the figure.

We define $\llcorner(P) \subseteq P$ to be the maximal set of points $q \in P$ such that there is no $p \in P$ with $p \prec q$. Symmetrically, define $\urcorner(P) \subseteq P$ to be the set of points $p \in P$ such that there is no $q \in P$ with $p \prec q$.

We now apply the above partial ordering to the set M of all matches between T_1 and T_2, interpreting each match $(a_1, a_2) \in M$ as a point in the plane $\mathbb{R}^2$ with the standard (x, y)-coordinate system. Repeatedly applying $\llcorner(\cdot)$ and $\urcorner(\cdot)$ to M gives us two layered partitions of M: Let $\llcorner_1 := \llcorner(M)$ and $\llcorner_i := \llcorner(M \setminus (\llcorner_1 \cup \cdots \cup \llcorner_{i-1}))$, while let $\urcorner_1 := \urcorner(M)$ and $\urcorner_j := \urcorner(M \setminus (\urcorner_1 \cup \cdots \cup \urcorner_{j-1}))$. Let m_1 and m_2 be the largest integers such that $\llcorner_{m_1} \neq \emptyset$ and $\urcorner_{m_2} \neq \emptyset$, respectively. See Fig. 1 for an illustration.

It is more intuitive to look at these structures from a geometric viewpoint. If $p \prec q$, then q is located in the first quadrant with apex p. Thus, the union U of quadrants with apices $p \in \llcorner_1$ contains all the other matches $M \setminus \llcorner_1$. Observe that the boundary of the union U forms a staircase-like polygonal chain, which implies a natural ordering on $\llcorner_1$. For any integer $1 \leq l \leq |\llcorner_i|$, let $\llcorner_i[l]$ denote the l-th point in $\llcorner_i$ along the boundary of U in the x-increasing order. The above argument applies symmetrically to $\urcorner_j$, and thus let $\urcorner_j[r]$ denote the r-th point in $\urcorner_j$ for $1 \leq r \leq |\urcorner_j|$.

The two partitions $\{\llcorner_1, \ldots, \llcorner_{m_1}\}$ and $\{\urcorner_1, \ldots, \urcorner_{m_2}\}$ of M provide a way to jump to "next" matches in M from any current instance $((a_1, a_2), (b_1, b_2))$ of the subproblems, and it suffices to solve $D((a_1, a_2), (b_1, b_2))$ by Lemma 1. Specifically, we obtain the following recurrence relation based on the partitions: For convenience, let $D^{i,j}_{l,r} := D(\llcorner_i[l], \urcorner_j[r])$, and let $\chi((a_1, a_2)) := T_1[a_1] = T_2[a_2]$ for any match $(a_1, a_2) \in M$. Then, for any $i \in \{1, \ldots, m_1\}$, $j \in \{1, \ldots, m_2\}$, $l \in \{1, \ldots, |\llcorner_i|\}$, and $r \in \{1, \ldots, |\urcorner_j|\}$,

$$D^{i,j}_{l,r} = \begin{cases} 1 + A(i, l; j, r) & \begin{pmatrix} \llcorner_i[l] \preceq \urcorner_j[r] \\ \chi(\llcorner_i[l]) = \chi(\urcorner_j[r]) \end{pmatrix} \\ \max\{B_{\llcorner}(i, l; j, r), B_{\urcorner}(i, l; j, r)\} & \text{(otherwise)} \end{cases},$$

where

$$A(i, l; j, r) := \max\{D^{i+1,j+1}_{l',r'} \mid \llcorner_i[l] \prec \llcorner_{i+1}[l'], \urcorner_{j+1}[r'] \prec \urcorner_j[r]\},$$
$$B_{\llcorner}(i, l; j, r) := \max\{D^{i,j+1}_{l,r'} \mid \urcorner_{j+1}[r'] \prec \urcorner_j[r]\}, \text{ and}$$
$$B_{\urcorner}(i, l; j, r) := \max\{D^{i+1,j}_{l',r} \mid \llcorner_i[l] \prec \llcorner_{i+1}[l']\}.$$

Here, we assume $\max \emptyset = 0$. The base case of the above recurrence is when $i = m_1$ or $j = m_2$, and in this case $A(i, l; j, r)$, $B_{\llcorner}(i, l; j, r)$, and $B_{\urcorner}(i, l; j, r)$ are all evidently zero since $\llcorner_{m_1+1} = \urcorner_{m_2+1} = \emptyset$. Note that the solution to the original problem $D((1, 1), (n, n))$ is taken by the maximum of $D^{1,1}_{l,r}$ over all $1 \leq l \leq |\llcorner_1|$ and $1 \leq r \leq |\urcorner_1|$. Hence, the problem can be solved via dynamic programming by running four indices: i from m_1 down to 1, j from m_2 down to 1, l from 1 to $|\llcorner_i|$, and r from 1 to $|\urcorner_j|$. It solves exactly $|M|^2 = \mathcal{R}^2$ subproblems.

Now we describe how to solve each subproblem in $O(1)$ amortized time. The following is a key observation for the purpose.

Lemma 2. *Let $p \in \{1, \ldots, n\}^2$ be any point. For any positive integers i, l_1, and l_2 with $1 \leq l_1 \leq l_2 \leq |\llcorner_i|$, if $p \prec \llcorner_i[l_1]$ and $p \prec \llcorner_i[l_2]$, then it holds that $p \prec \llcorner_i[l']$ for any $l_1 \leq l' \leq l_2$. Symmetrically, for any positive integers j, r_1, and r_2 with $1 \leq r_1 \leq r_2 \leq |\urcorner_j|$, if $\urcorner_j[r_1] \prec p$ and $\urcorner_j[r_2] \prec p$, then it holds that $\urcorner_j[r'] \prec p$ for any $r_1 \leq r' \leq r_2$.*

Proof. Suppose that $p \prec \llcorner_i[l_1]$ and $p \prec \llcorner_i[l_2]$. By the structure of $\llcorner_i$, the point $\llcorner_i[l']$ for any $l_1 \leq l' \leq l_2$ is contained in the axis-parallel rectangle R whose upper-left corner is $\llcorner_i[l_1]$ and whose lower-right corner is $\llcorner_i[l_2]$. Since the first quadrant at p contains $\llcorner_i[l_1]$ and $\llcorner_i[l_2]$, so does the rectangle. See Fig. 2 for an illustration. This implies that the first quadrant at p contains all points $\llcorner_i[l']$ and it thus holds that $p \prec \llcorner_i[l']$.

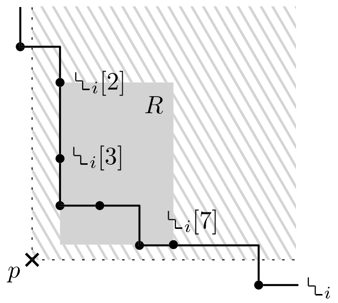


Fig. 2. Illustration to Lemma 2.

Next, suppose that $\urcorner_j[r_1] \prec p$ and $\urcorner_j[r_2] \prec p$. As above, the point $\urcorner_j[r']$ for any $r_1 \le r' \le r_2$ is contained in the axis-parallel rectangle R whose upper-left corner is $\urcorner_j[r_1]$ and whose lower-right corner is $\urcorner_j[r_2]$. On the other hand, the third quadrant at p contains both $\urcorner_j[r_1]$ and $\urcorner_j[r_2]$ since the first quadrant at $\urcorner_j[r_1]$ contains p and the first quadrant at $\urcorner_j[r_2]$ contains p. This implies that the rectangle R is completely contained in the third quadrant at p, and so is $\urcorner_j[r']$ for any $r_1 \le r' \le r_2$. Hence, the first quadrant at each $\urcorner_j[r']$ contains p, and it thus holds that $\urcorner_j[r'] \prec p$. □

Let $I^i_{\llcorner}(p)$ be the set of positive integers l' such that $p \prec \llcorner_i[l']$, and $I^j_{\urcorner}(p)$ be the set of those r' such that $\urcorner_j[r'] \prec p$. Then, Lemma 2 implies that for any point p both $I^i_{\llcorner}(p)$ and $I^j_{\urcorner}(p)$ form a set of consecutive integers in $\{1, \dots, |\llcorner_i|\}$ and $\{1, \dots, |\urcorner_j|\}$, respectively. For an illustrative example, see Fig. 2 in which $I^i_{\llcorner}(p) = \{2, 3, 4, 5, 6, 7\}$. Now, we regard $D^{i,j}$ as a $|\llcorner_i| \times |\urcorner_j|$ matrix with elements $D^{i,j}_{l,r}$ for any $l \in \{1, \dots, |\llcorner_i|\}$ and $r \in \{1, \dots, |\urcorner_j|\}$. We then observe that the three values $A(i, l; j, r)$, $B_{\llcorner}(i, l; j, r)$, and $B_{\urcorner}(i, l; j, r)$ are the maximum elements of a submatrix of $D^{i+1,j+1}$, $D^{i,j+1}$, and $D^{i+1,j}$, respectively. Hence, computing $D^{i,j}_{l,r}$ can be done by processing one or two range queries on the matrices. Moreover, our queries are not arbitrary but obey a strict order as follows:

Lemma 3. *If $I^{i+1}_{\llcorner}(\llcorner_i[l]) = \{l_1, \dots, l_2\}$ and $I^{i+1}_{\llcorner}(\llcorner_i[l+1]) = \{l'_1, \dots, l'_2\}$, then we have $l_1 \le l'_1$ and $l_2 \le l'_2$. Analogously, if $I^{j+1}_{\urcorner}(\urcorner_j[r]) = \{r_1, \dots, r_2\}$ and $I^{j+1}_{\urcorner}(\urcorner_j[r+1]) = \{r'_1, \dots, r'_2\}$, then we have $r_1 \le r'_1$ and $r_2 \le r'_2$.*

Proof. First, observe that if $\llcorner_i[l] = (x_l, y_l)$ and $\llcorner_i[l+1] = (x_{l+1}, y_{l+1})$, then it holds that $x_l \le x_{l+1}$ and $y_l \ge y_{l+1}$ by the structure of $\llcorner_i$.

Suppose to the contrary that $l_1 > l'_1$. This implies that the x-coordinate of $\llcorner_i[l+1]$ is strictly smaller than that of $\llcorner_i[l]$. This is a contradiction to our construction of $\llcorner_i$ and the way of indexing the points in $\llcorner_i$. Similarly, if $l_2 > l'_2$, then the y-coordinate of $\llcorner_i[l+1]$ must be strictly larger than that of $\llcorner_i[l]$, a contradiction.

The case of $\urcorner_j$ can also be verified in an analogous way. □

Our implementation of the dynamic programming computes the matrices $D^{i,j}$ by running i from m_1 down to 1 and j from m_2 down to 1. Assume that we are about to compute $D^{i,j}$ for any $i < m_1$ and $j < m_2$ and we have correctly computed three matrices $D^{i+1,j+1}$, $D^{i,j+1}$, and $D^{i+1,j}$. For $l = 1, \dots, |\llcorner_i|$ and $r = 1, \dots, |\urcorner_j|$ in order, we describe how to compute $A(i, l; j, r)$, $B_{\llcorner}(i, l; j, r)$, and $B_{\urcorner}(i, l; j, r)$.

First, we describe how to compute $B_{\llcorner}(i, l; j, r)$ for all $l = 1, \dots, |\llcorner_i|$ and $r = 1, \dots, |\urcorner_j|$ in $O(|\llcorner_i| \cdot (|\urcorner_j| + |\urcorner_{j+1}|))$ time. Fix any $l \in \{1, \dots, |\llcorner_i|\}$. For simplicity, let $I_r := I^{j+1}_{\urcorner}(\urcorner_j[r])$ for each $r \in \{1, \dots, |\urcorner_j|\}$. Recall that $B_{\llcorner}(i, l; j, r) = \max\{D^{i,j+1}_{l,r'} \mid r' \in I_r\}$. Thus, $B_{\llcorner}(i, l; j, r)$ is the maximum in the corresponding interval I_r in the l-th row of $D^{i,j+1}$.

For our purpose, we exploit a special queue Q that supports all standard queue operations and in addition the operation that reports the maximum element in Q in $O(1)$ amortized time. We show how to build such a structure Q, similar to the one in [4]. It consists of a standard queue Q_1 and a standard deque Q_2. When we insert v into Q, it is added at the rear of Q_1. Also we repeatedly compare the element w at the rear of Q_2 and remove w from Q_2 if $v > w$. We stop if $v \le w$. Finally, add v into the rear of Q_2. Since we can charge a removal from Q_2 to an addition to Q_1, this operation takes $O(1)$ amortized time. On a removal on Q, we remove the front element v from Q_1 and if the front element of Q_2 is equal to v, then remove it from Q_2, too. This can be done in $O(1)$ time. Note that the maximum element in Q is located at the front of Q_2: reporting the maximum is easily done in $O(1)$ time.

As r increases from 1 to $|\urcorner_j|$, Q will store the values of $D^{i,j+1}_{l,r'}$ for each $r' \in I_r$. Then, $B_{\llcorner}(i, l; j, r)$ can be obtained by the maximum operation on Q in $O(1)$ time.

Now, we describe how to update I_r and Q as r increases. First, we initialize Q to be an empty set. For $r = 1$, we specify I_1 and add the values of $D^{i,j+1}_{l,r'}$ to Q in order for all $r' \in I_1$. For any $r \ge 2$, we already know I_{r-1}. By Lemma 3, I_r can be found by exploring from each endpoint of I_{r-1} to the right. Add $D^{i,j+1}_{l,r'}$ for $r' \in I_r \setminus I_{r-1}$ to Q in order and remove $D^{i,j+1}_{l,r'}$ for $r' \in I_{r-1} \setminus I_r$ from Q.

Finding all the intervals I_r for all r takes $O(|\urcorner_j| + |\urcorner_{j+1}|)$ since the endpoints of I_r always move to the right. Also, it is easy to see that the above procedure calls $O(|\urcorner_j| + |\urcorner_{j+1}|)$ operations to Q since every $D^{i,j+1}_{l,r'}$ is once added to Q and removed, and one maximum operation is performed at each r to evaluate $B_{\llcorner}(i,l;j,r)$. Therefore, repeating this for all $l = 1, \ldots, |\llcorner_i|$ takes $O(|\llcorner_i| \cdot (|\urcorner_j| + |\urcorner_{j+1}|))$ time.

Computing $B_{\urcorner}(i,l;j,r)$ can be done in an analogous way, taking $O((|\llcorner_i| + |\llcorner_{i+1}|) \cdot |\urcorner_j|)$ time. For simplicity, we let $I_r := I^{j+1}_{\urcorner}(\urcorner_j[r])$ and $I'_l := I^{i+1}_{\llcorner}(\llcorner_i[l])$. As done above, one can specify all the intervals I_r and I'_l in $O(|\llcorner_i| + |\llcorner_{i+1}| + |\urcorner_j| + |\urcorner_{j+1}|)$ time. Note that $A(i,l;j,r)$ is the maximum in $D^{i+1,j+1}$ over range $I'_l \times I_r$.

We associate each column of $D^{i+1,j+1}$ with an instance of our special queue; for $r' \in \{1, \ldots, |\urcorner_{j+1}|$, let $Q_{r'}$ be the queue associated with the r'-th column of $D^{i+1,j+1}$. As l increases from 1 to $|\llcorner_i|$, each $Q_{r'}$ will store the values of $D^{i+1,j+1}_{l',r'}$ for all $l' \in I'_l$. This can be done similarly as above by Lemma 3: we add $D^{i+1,j+1}_{l',r'}$ into $Q_{r'}$ for all $l' \in I'_l \setminus I'_{l-1}$ and remove $D^{i+1,j+1}_{l',r'}$ from $Q_{r'}$ for all $l' \in I'_{l-1} \setminus I'_l$. Thus, for each l, we will have the queues $Q_1, \ldots, Q_{|\urcorner_{j+1}|}$ that all together store the rows of $D^{i+1,j+1}$ corresponding to the interval I'_l.

Now, consider any fixed l. We construct an array Z of $|\urcorner_{j+1}|$ elements such that $Z[r'] = \max\{D^{i+1,j+1}_{l',r'} | l' \in I'_l\}$. This array Z can be computed by performing the maximum operation on each queue $Q_{r'}$ in $O(|\urcorner_{j+1}|)$ time. Initialize another queue Q. As r increases from 1 to $|\urcorner_j|$, the new queue Q will store $Z[r']$ for all $z' \in I_r$. Again by Lemma 3, this can be done for each r by adding $Z[r']$ for all $z' \in I_r \setminus I_{r-1}$ and removing those for all $z' \in I_{r-1} \setminus I_r$. Then, at each r, the maximum among those stored in Q is exactly $\max\{D^{i+1,j+1}_{l',r'} \mid l' \in I'_l, r' \in I_r\}$, that is, $A(i,l;j,r)$.

Theorem 1. *$LCPS(T_1, T_2)$ can be computed in $O(n + \mathcal{R}^2)$ time.*

Proof. Our algorithm runs in three steps: (1) specify all the matches M between two given strings T_1 and T_2, (2) compute the partitions $\{\llcorner_1, \ldots, \llcorner_{m_1}\}$ and $\{\urcorner_1, \ldots, \urcorner_{m_2}\}$ of M, and (3) compute the values of $D^{i,j}_{l,r}$ as described above.

The first step can be done in $O(n + \mathcal{R})$ time by sorting all the tuples $(T_1[i], i)$ and $(T_2[j], j)$.

The second step can be done in $O(\mathcal{R}^2)$ time: after sorting M in the x-increasing order, $\llcorner_1$ can be computed in linear time. Then, we remove $\llcorner_1$ from M and repeat the above until there is nothing left. Since every $\llcorner_i$ has at least one point for $1 \le i \le m_1$, we have $m_1 \le \mathcal{R}$, and hence this takes at most $O(\mathcal{R}^2)$ time. Computing $\urcorner_1, \ldots, \urcorner_{m_2}$ is analogous.

Finally, for each i and j, computing $D^{i,j}_{l,r}$ for all $1 \le l \le |\llcorner_i|$ and $1 \le r \le |\urcorner_j|$ takes $O((|\llcorner_i| + |\llcorner_{i+1}|)(|\urcorner_j| + |\urcorner_{j+1}|))$ time. Summing this all over i and j, we see that this step takes $O(\mathcal{R}^2)$ time as

$$\sum_{i=1}^{m_1} \sum_{j=1}^{m_2} (|\llcorner_i| + |\llcorner_{i+1}|)(|\urcorner_j| + |\urcorner_{j+1}|) \le \sum_{i=1}^{m_1} (|\llcorner_i| + |\llcorner_{i+1}|) \cdot 2\mathcal{R} \le 2\mathcal{R} \cdot 2\mathcal{R} = 4\mathcal{R}^2. \quad \square$$

Now we are interested in the average time complexity. We assume that T_1 and T_2 are random strings over the alphabet Σ: each character of Σ has the same probability of appearing at each position of T_1 and T_2.

Lemma 4. *If T_1 and T_2 are random strings over the alphabet Σ, then $E[\mathcal{R}] = n^2/|\Sigma|$. Also, $E[\mathcal{R}^2] = O(n^4/|\Sigma|^2)$.*

Proof. Let X_i be the number of matches between $T_1[i]$ and T_2. We also define X_{ij} such that $X_{ij} = 1$ if $T_1[i] = T_2[j]$ and $X_{ij} = 0$ otherwise. It is straightforward that $X_i = \sum_{1 \le j \le n} X_{ij}$. By linearity of expectation,

$$E[X_i] = \sum_{1 \le j \le n} E[X_{ij}] = n/|\Sigma|.$$

By definition, $\mathcal{R} = \sum_{1 \le i \le n} X_i$. Again, we get

$$E[\mathcal{R}] = \sum_{1 \le i \le n} E[X_i] = n^2/|\Sigma|.$$

Now we are interested in $E[\mathcal{R}^2]$. As all X_i's are mutually independent and $X_i \sim B(n, 1/|\Sigma|)$,

$$Var[\mathcal{R}] = n \cdot Var[X_i] = n^2 \frac{|\Sigma| - 1}{|\Sigma|^2}.$$

Finally,

$$E[\mathcal{R}^2] = \{E[\mathcal{R}]\}^2 + Var[\mathcal{R}] = \frac{n^4}{|\Sigma|^2} + n^2 \frac{|\Sigma| - 1}{|\Sigma|^2} = O(n^4/|\Sigma|^2). \quad \square$$

Theorem 2. *The average time complexity of our algorithm that computes $LCPS(T_1, T_2)$ is $O(n^4/|\Sigma|^2)$.*

Proof. Straight from Lemma 4. □

4. Conclusions

We showed that the longest common palindromic subsequence problem can be solved in $O(n + \mathcal{R}^2)$ time. Of course in the worst case $\mathcal{R} = n^2$, but the merit of our algorithm is that it does not depend on the length of strings, but it depends on the number of matches between two strings. With random strings, the average time complexity is $O(n^4/|\Sigma|^2)$. If we consider an integer alphabet, then the average time complexity will be much smaller than $O(n^4)$.

Acknowledgement

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2011-0013963 and 2015R1D1A1A01057220).

References

[1] A. Apostolico, D. Breslauer, Z. Galil, Parallel detection of all palindromes in a string, Theoret. Comput. Sci. 141 (1–2) (1995) 163–173.
[2] A.V. Aho, D.S. Hirschberg, J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, J. ACM 23 (1) (1976) 1–12.
[3] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: SPIRE 2000, pp. 39–48.
[4] M.A. Babenko, T.A. Starikovskaya, Computing longest common substrings via suffix arrays, in: CSR 2008, pp. 64–75.
[5] S.R. Chowdhury, Md.M. Hassan, S. Iqbal, M.S. Rahman, Computing a longest common palindromic subsequence, Fund. Inform. 129 (2014) 329–340.
[6] M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, Speeding-up Hirschberg and Hunt–Szymanski LCS algorithms, in: SPIRE 2001, pp. 59–67.
[7] V. Chvátal, D. Sankoff, Longest common subsequences of two random sequences, J. Appl. Probab. 12 (1975) 306–315.
[8] S. Deorowicz, A. Danek, Bit-parallel algorithms for the merged longest common subsequence problem, Internat. J. Found. Comput. Sci. 24 (8) (2013) 1281–1298.
[9] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.
[10] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[11] M. Kiwi, M. Loebl, J. Matoušek, Expected length of the longest common subsequence for large alphabets, Adv. Math. 197 (2) (2005) 480–498.
[12] I. Lee, C.S. Iliopoulos, K. Park, Linear time algorithm for the longest common repeat problem, in: SPIRE 2004, pp. 10–17.
[13] G. Manacher, A new linear-time on-line algorithm for finding the smallest initial palindrome of a string, J. ACM 22 (3) (1975) 346–351.