Theoretical Computer Science 922 (2022) 475–485

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

An efficient algorithm for the longest common palindromic subsequence problem ☆,☆☆

Ting-Wei Liang [a], Chang-Biau Yang [b,*], Kuo-Si Huang [b]

[a] Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
[b] Department of Business Computing, National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan

ARTICLE INFO

Article history:
Received 8 September 2021
Received in revised form 25 February 2022
Accepted 25 April 2022
Available online 29 April 2022
Communicated by R. Giancarlo

Keywords:
Longest common subsequence
Longest common palindromic subsequence
Diagonal method
3-D domination

ABSTRACT

The longest common palindromic subsequence (LCPS) problem is a variant of the longest common subsequence (LCS) problem. Given two input sequences A and B, the LCPS problem is to find the common subsequence of A and B such that the answer is a palindrome with the maximal length. The LCPS problem was first proposed by Chowdhury et al. (2014) [9], who proposed a dynamic programming algorithm with $\mathcal{O}(m^2n^2)$ time, where m and n denote the lengths of A and B, respectively. This paper proposes a diagonal-based algorithm for solving the LCPS problem with $\mathcal{O}(L(m-L)R\log n)$ time and $\mathcal{O}(RL)$ space, where R denotes the number of match pairs between A and B, and L denotes the LCPS length. In our algorithm, the 3-dimensional minima finding algorithm is invoked to overcome the domination problem. As experimental results show, our algorithm is efficient practically compared with some previously published algorithms.

1. Introduction

The *longest common subsequence* (LCS) problem has been studied extensively for the past decades. It can be used to calculate the similarity between two strings. The LCS problem has been widely applied to bioinformatics, string comparison, speech recognition, and many other fields [3,5,6].

Given two sequences $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$, where $m = |A|$ and $n = |B|$, the LCS problem aims to find the common subsequence with the maximal length between A and B by deleting zero or more symbols from the two sequences. Lots of LCS algorithms have been developed, and several variants of the LCS problem have also been proposed, such as *longest common increasing subsequence* (LCIS) [19,27], *constrained longest common subsequence* (CLCS) [2,16,25], *longest common square subsequence* (LCSqS) [18] and *longest common palindromic subsequence* (LCPS) [4,9,17]. The definitions and NP-hardness proofs of the 2-dimensional LCS problems, another variant of the LCS problem, were proposed by Chan et al. [8].

Nakatsu et al. [22] proposed a diagonal-based algorithm to solve the LCS problem in $\mathcal{O}(n(m-L'))$ time, where m and n denote the lengths of input sequences A and B, respectively, $m \le n$, and L' denotes the LCS length. Its time complexity implies that its performance is extremely efficient when the two input sequences are similar, that is, L' is large. The diag-

☆ This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 108-2221-E-110-031.
☆☆ A preliminary version of this paper was presented at the 37th Workshop on Combinatorial Mathematics and Computation Theory, July 29-30, 2020, Kaohsiung, Taiwan [20].
* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

https://doi.org/10.1016/j.tcs.2022.04.046
0304-3975/

Table 1
The time complexities and space complexities of the LCPS algorithms. A and B are the two input sequences; $|A| = m \le n = |B|$; Σ: alphabet set; R: number of match pairs between A and B; R_1: number of match pairs in A itself; R_2: number of match pairs in B itself; L: length of LCPS(A, B).

Authors	Year	Time complexity	Space complexity	Note
Chowdhury et al. [9]	2014	$\mathcal{O}(m^2n^2)$ or $\mathcal{O}(R^2 \log^2 n \log\log n)$	$\mathcal{O}(m^2n^2)$ or $\mathcal{O}(R^2)$	DP or Match pair
Hasan et al. [13]	2017	$\mathcal{O}(R_1R_2\|\Sigma\|)$	$\mathcal{O}(R_1R_2)$	Match pair, Automata
Inenaga and Hyyrö [17]	2018	$\mathcal{O}(n + R^2\|\Sigma\|)$	$\mathcal{O}(R^2)$	Match pair, Rectangle
Bae and Lee [4]	2018	$\mathcal{O}(n + R^2)$	$\mathcal{O}(R^2)$	Incorrect, Match pair, Dominant contour
This paper	2022	$\mathcal{O}(L(m-L)R \log n)$	$\mathcal{O}(RL)$	Diagonal, Match pair, Domination

onal concept has been successfully used to solve some variants of the LCS problem, such as the merged longest common subsequence problem [26] and the longest common increasing subsequence problem [21].

A *palindrome* is a sequence whose readings from its forward and backward directions are the same. That is, $A = a_1a_2\cdots a_m = a_ma_{m-1}\cdots a_1$. For example, `abfba` and `abffba` are palindromes. In bioinformatics, palindromic sequences appear extensively in genomes. Palindromic sequences can be found in the DNA of plasmid, virus and bacteria, and they are also present in cancer cells. In addition, many DNA binding sites of proteins, transcription factors and terminators are also palindromic sequences [11,14,23]. Thus, some researchers paid attention to palindromes, such as counting distinct palindromic strings in a sequence [12] and the palindromic length problem (factorizing a sequence into palindromes) [1,7,10,15].

Given two sequences $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$, the *longest common palindromic subsequence* (LCPS) problem is to find a common palindromic subsequence of A and B with the maximum length. In other words, the answer should be a palindrome. For example, suppose A = `cbccbaabb` and B = `bbccabbca`. Then we have LCS(A, B) = `bccabb` and LCPS(A, B) = `bbabb`.

The LCPS problem was first proposed by Chowdhury et al. [9] in 2014. They presented a *dynamic programming* (DP) algorithm with $\mathcal{O}(m^2n^2)$ time and space, and an algorithm with $\mathcal{O}(R^2 \log^2 n \log\log n)$ time and $\mathcal{O}(R^2)$ space by mapping the problem into computational geometry, where R is the total number of match pairs between A and B. In 2018, Inenaga and Hyyrö [17] presented a DP algorithm with $\mathcal{O}(n + R^2|\Sigma|)$ time and $\mathcal{O}(R^2)$ space, and proved that the *LCS with four strings* (FLCS) problem can be reduced to the LCPS problem. In the same year, Bae and Lee [4] claimed that they presented a DP algorithm with $\mathcal{O}(n + R^2)$ time and $\mathcal{O}(R^2)$ space. However, we find that the algorithm of Bae and Lee [4] is not correct. We shall present some counterexamples for their algorithm in Section 2.3. The time complexities and the space complexities of the previous LCPS algorithms are summarized in Table 1.

In this paper, we propose a diagonal-based algorithm for solving the LCPS problem with $\mathcal{O}(L(m-L)R \log n)$ time and $\mathcal{O}(RL)$ space, where R denotes the number of match pairs between input sequences A and B, and L denotes the LCPS length. As experimental results show, our algorithm is efficient practically on some pseudorandom datasets, compared with some previously published algorithms.

The organization of this paper is as follows. Section 2 presents some preliminaries for this paper. In Section 3, we present our diagonal-based algorithm for solving the LCPS problem. In Section 4, we implement our algorithm and some previous algorithms. Then, some experimental results on pseudorandom datasets are illustrated. Finally, the conclusion is given in Section 5.

2. Preliminaries

A sequence (string) $A = a_1a_2\cdots a_m$ is composed of characters over a finite alphabet Σ, where $|A| = m$ denotes its length. $A_{i..j}$ represents the substring of A from index i to index j. $A_{i..j} = \emptyset$ if $j < i$. $\overline{A}$ denotes the reverse sequence of A, that is $\overline{A} = a_ma_{m-1}\cdots a_1$. Note that $\overline{A_{1..i}} = a_ia_{i-1}\cdots a_1$ and $\overline{A}_{1..i} = a_ma_{m-1}\cdots a_{m-i+1} = \overline{A_{m-i+1..m}}$.

2.1. The LCPS Algorithm by Chowdhury et al. [9]

In 2014, Chowdhury et al. [9] first proposed the *longest common palindromic subsequence* (LCPS) problem. Given two sequences A and B, let $V(p, q, r, s)$ denote the length of LCPS($A_{p..q}$, $B_{r..s}$), where $1 \le p \le q \le m = |A|$ and $1 \le r \le s \le n = |B|$. Their DP algorithm, with $\mathcal{O}(m^2n^2)$ time, is shown in Eq. (1).

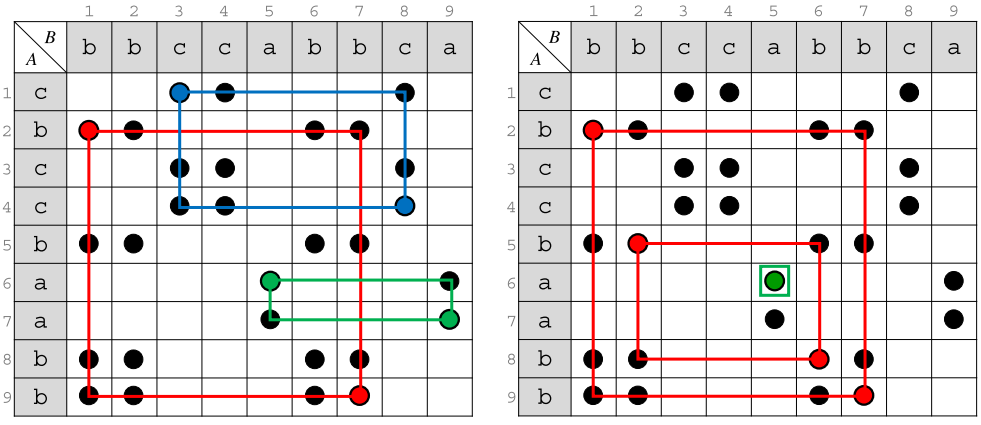


Fig. 1. An example for the algorithm of Inenaga and Hyyrö [17] with $A = \texttt{cbccbaabb}$ and $B = \texttt{bbccabbca}$. (a) $|\Sigma| = 3$ largest rectangles inside rectangle $(\langle 0,0\rangle, \langle 10,10\rangle)$ for each character. Green: a; Red: b; Blue: c. (b) The 3 rectangles that represent the LCPS bbabb with length $2+2+1=5$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

$$V(p,q,r,s) = \begin{cases} 0 & \text{if } p > q \text{ or } r > s; \\ 1 & \text{if } ((p = q \text{ and } r \le s) \\ & \text{or } (p \le q \text{ and } r = s)), \\ & \text{and } a_p = a_q = b_r = b_s; \\ 2 + V(p+1, q-1, r+1, s-1) & \text{if } p < q, r < s, \\ & \text{and } a_p = a_q = b_r = b_s; \\ \max \begin{cases} V(p+1,q,r,s) \\ V(p,q,r+1,s) \\ V(p,q-1,r,s) \\ V(p,q,r,s-1) \end{cases} & \text{otherwise.} \end{cases} \tag{1}$$

2.2. Maximum depth nesting rectangle structures by Inenaga and Hyyrö [17]

Inenaga and Hyyrö [17] gave the solution of the LCPS problem by solving the *maximum depth nesting rectangle structures* (MDNRS) problem. Their algorithm first puts the match pairs between A and B on the 2D plane as points. Thus, every two points with the same matched character form a rectangle. The algorithm calculates the MDNRS size of these rectangles with $\mathcal{O}(n + R^2|\Sigma|)$ time, where R denotes the number of match pairs.

Each rectangle may contain at most $|\Sigma|$ largest rectangles, where each rectangle is for one character. For example, given two sequences $A = \texttt{cbccbaabb}$ and $B = \texttt{bbccabbca}$, Fig. 1(a) shows the $|\Sigma| = |\{\texttt{a},\texttt{b},\texttt{c}\}| = 3$ largest rectangles inside rectangle $(\langle 0,0\rangle, \langle m+1, n+1\rangle)$. Here, the maximum MDNRS is shown in Fig. 1(b). The rectangle $(\langle 0,0\rangle, \langle 10,10\rangle)$ has three rectangles $(\langle 2,1\rangle, \langle 9,7\rangle)$, $(\langle 5,2\rangle, \langle 8,6\rangle)$ and $(\langle 6,5\rangle, \langle 6,5\rangle)$ inside. And $(\langle 6,5\rangle, \langle 6,5\rangle)$ matches the same position in A and B, so this rectangle represents 1 match length in the LCPS answer. Therefore, we get the LCPS length is $2+2+1=5$.

2.3. The LCPS algorithm by Bae and Lee [4]

In 2018, Bae and Lee [4] claimed that they proposed an $\mathcal{O}(n + R^2)$-time algorithm to solve the LCPS problem. However, we will show that their algorithm is not correct in some cases.

The algorithm forms the match pairs between A and B into minima set F and maxima set T. For example, suppose $A = \texttt{cbccbaabb}$ and $B = \texttt{bbccabbca}$. F and T are shown in Fig. 2(a) and Fig. 2(b), respectively. F is partitioned into layers, labeled as $F_1, F_2, \cdots$, starting from the minima. And T is partitioned into layers, labeled as $T_1, T_2, \cdots$, starting from the maxima. Let (x, y) denote the 2D index of a_x in A and b_y in B. $F_i[l]$ denotes the 2D index of the lth match point in F_i with the right to left order, and $T_j[r]$ is defined symmetrically with the left to right order. For example, in Fig. 2, $F_1[1] = (1,8)$, $F_1[2] = (1,4)$, $F_1[5] = (2,1)$, $T_3[1] = (7,5)$, and $T_3[4] = (2,7)$. In addition, $(x,y) \prec (x',y')$ denotes $x < x'$ and $y < y'$; $(x,y) = (x',y')$ denotes $x = x'$ and $y = y'$. Let $D^{i,j}_{l,r}$ denote the number of match characters within $F_i[l]$ and $T_j[r]$. The algorithm calculates $D^{i,j}_{l,r}$, instead of the LCPS length, as shown in Eq. (2), where $C(F_i[l])$ and $C(T_j[r])$ denote the characters of $F_i[l]$ and $T_j[r]$, respectively.

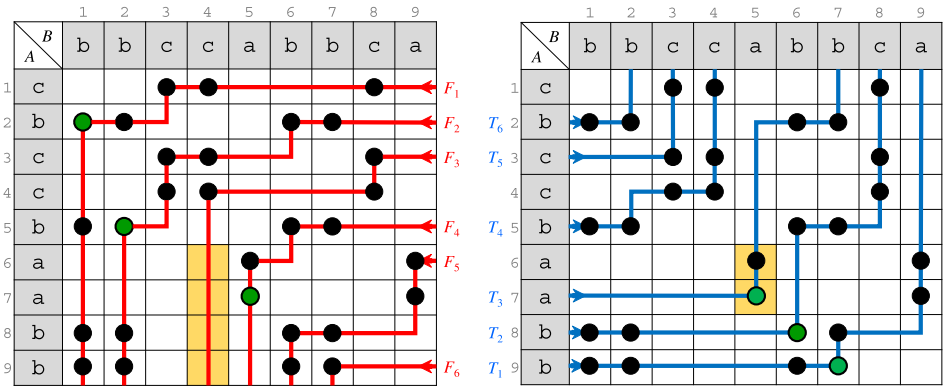


Fig. 2. A counterexample of Bae and Lee's algorithm [4] with $A = \texttt{cbccbaabb}$ and $B = \texttt{bbccabbca}$. The correct $|D(A,B)| = |\texttt{bba}| = 3$ with LCPS answer bbabb. But the result calculated by Bae and Lee is $|D(A,B)| = |\texttt{bb}| = 2$ with LCPS answer bbbb, which is incorrect. (a) The minima set F, partitioned by red lines. (b) The maxima set T, partitioned by blue lines.

$$D_{l,r}^{i,j} = \begin{cases} 1 + \text{FTMax}(i,l,j,r) & \text{if } (F_i[l] \prec T_j[r]) \text{ or } (F_i[l] = T_j[r]), \\ & \text{and } C(F_i[l]) = C(T_j[r]); \\ \max\begin{cases} \text{FMax}(i,l,j,r) \\ \text{TMax}(i,l,j,r) \end{cases} & \text{otherwise;} \end{cases} \tag{2}$$

where

$$\begin{aligned} \text{FTMax}(i,l,j,r) &= \max\{D_{l',r'}^{i+1,j+1} | F_i[l] \prec F_{i+1}[l'], T_{j+1}[r'] \prec T_j[r]\}, \\ \text{FMax}(i,l,j,r) &= \max\{D_{l,r'}^{i,j+1} | T_{j+1}[r'] \prec T_j[r]\}, \\ \text{TMax}(i,l,j,r) &= \max\{D_{l',r}^{i+1,j} | F_i[l] \prec F_{i+1}[l']\}. \end{aligned}$$

Eq. (2) calculates the values with the scheme of layer by layer. The values on layer F_i and layer T_j are obtained from layers F_{i+1} and T_{j+1} with the DP method. In other words, the values on outer layers depend on only the next inner layers. Therefore, Bae and Lee [4] claimed that each $D_{l,r}^{i,j}$ can be calculated in $\mathcal{O}(1)$ amortized time, and the total complexity is $\mathcal{O}(R^2)$ time.

However, the algorithm proposed by Bae and Lee is not correct for all cases. Fig. 2 shows a counterexample. Let $D(A,B)$ denote the LCPS of A and B by Eq. (2). Their algorithm calculates $|D(A,B)| = |\texttt{bb}| = 2$, with LCPS length 4. But the correct value is $|D(A,B)| = |\texttt{bba}| = 3$, with LCPS length 5 since the third match a is overlapped. In the correct situation, the green (or gray) match pairs in Fig. 2(a) and 2(b) represent bba. But, while calculating $D_{6,3}^{2,2}$ with Eq. (2), we have $F_2[6] \prec T_2[3]$ and $C(F_2[6]) = C(T_2[3]) = \texttt{b}$, thus $D_{6,3}^{2,2} = 1 + \text{FTMax}(2,6,2,3)$. For the match $F_2[6] = (5,2)$, we cannot find any match in F_3. We have $(5,2) \prec \emptyset$, since no element in F_3 satisfies the condition. Then, $D_{4,1}^{4,3}$ would never be considered, and thus FTMax(2,6,2,3) would return 0. The algorithm gets $D_{6,3}^{2,2} = 1 + 0 = 1$ and $D_{5,4}^{1,1} = 1 + D_{6,3}^{2,2} = 2$. But the correct value of $D_{5,4}^{1,1}$ is 3 with bba.

The second counterexample is shown in Fig. 3, and more counterexamples are shown in Table 2. According to these counterexamples, we show that the algorithm proposed by Bae and Lee is not correct. The D values cannot be calculated correctly in some cases. For matches in F_i and T_j, we should consider the next match pairs not only in F_{i+1} and T_{j+1}, but also in more inner layers. To correct the algorithm, we have to consider $\mathcal{O}(|\Sigma|)$ next match pairs, similar to the algorithm proposed by Inenaga and Hyyrö [17]. Then, each of D values cannot be computed in $\mathcal{O}(1)$ amortized time. Therefore, the algorithm proposed by Bae and Lee [4] with $\mathcal{O}(n + R^2)$ time cannot solve the LCPS problem correctly.

2.4. Multi-dimensional maxima finding

Our algorithm needs to determine the dominants in a set of 3-dimensional points (explained later). In the multi-dimensional space, a point p in a set S of n points is said to be *maximal* or *dominant* if there is no other point $q \in S$ whose coordinate values are all greater than or equal to the corresponding coordinate values of p. The set of all the maximal points in S is called the *maxima set* of S.

In 1975, Kung et al. [24] proposed an algorithm for finding d-dimensional maxima, where $d = 2$ or 3, and $d \geq 4$. The time complexity of their algorithm is $\mathcal{O}(n \log n)$ when $d = 2$ or 3, or $\mathcal{O}(n(\log n)^{d-2})$ when $d \geq 4$. Here, we introduce their algorithm for $d = 3$. Suppose that the input set S consists of n points $s_1, s_2, \cdots, s_n$ with their 3-dimensional coordinates

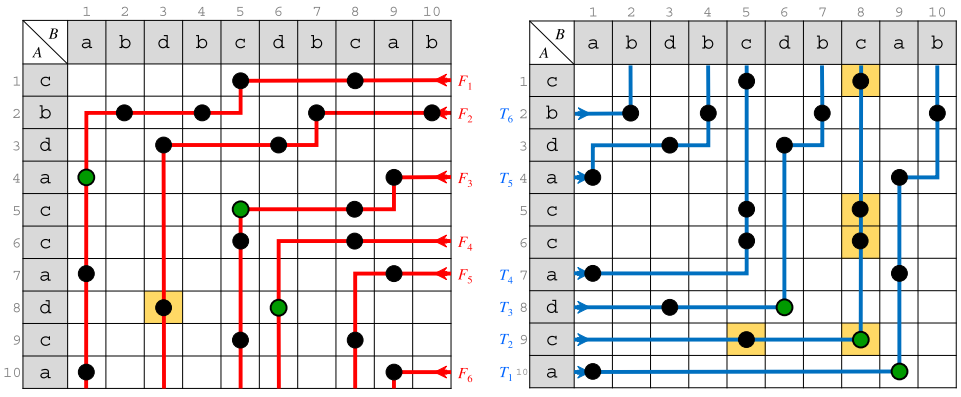


Fig. 3. The second counterexample of Bae and Lee's algorithm [4] with $A = \texttt{cbdaccadca}$ and $B = \texttt{abdbcdbcab}$. The correct $|D(A,B)| = |\texttt{acd}| = 3$ with LCPS answer `acdca`. But the result calculated by Bae and Lee is $|D(A,B)| = |\texttt{ac}| = |\texttt{ad}| = 2$ with LCPS answer `aca` or `ada`, which is incorrect. (a) The minima set F, partitioned by red lines. (b) The maxima set T, partitioned by blue lines.

Table 2
Some counterexamples of the algorithm proposed by Bae and Lee [4].

Input sequences	LCPS length	Correct LCPS	Bae and Lee's answer
$A = \texttt{cbccbaabb}$ $B = \texttt{bbccabbca}$	5	`bbabb`	`bbbb`
$A = \texttt{cbdaccadca}$ $B = \texttt{abdbcdbcab}$	5	`acdca`	`aca` or `ada`
$A = \texttt{aabbba}$ $B = \texttt{baaabb}$	3	`aaa` or `bbb`	`aa` or `bb`
$A = \texttt{bababcaddd}$ $B = \texttt{bacbbbddad}$	4	`abba`	`aba` or `bab`
$A = \texttt{acbcaddaba}$ $B = \texttt{cdcadbdbdb}$	4	`bddb`	`bb` or `dd`
$A = \texttt{dbaccccbbd}$ $B = \texttt{cddacccbaa}$	4	`cccc`	`ccc`
$A = \texttt{aaaacdcbbb}$ $B = \texttt{baaabccbad}$	4	`aaaa`	`aaa` or `bbb`
$A = \texttt{baacdbcbab}$ $B = \texttt{dcdbcdabab}$	5	`babab`	`baab`
$A = \texttt{aacbcabacbcaaad}$ $B = \texttt{cbcabdacaadabdc}$	9	`cbcabacbc`	`cbaabc`

(values of x, y and z). Let s_i^* denote the projection point of s_i onto the yz plane. M stores the 2-dimensional maxima set of these projection points s_i^*. The maxima set of 3-dimensional points can be found as follows.

Step 1: Sort s_i in S by the x-coordinate, so that $x(s_1) \geq x(s_2) \geq \cdots \geq x(s_n)$.
Step 2: Set $i \leftarrow 1$ and $M \leftarrow \emptyset$.
Step 3: If s_i^* is maximal in M, then $M \leftarrow \textsc{Maxima}(M \cup \{s_i^*\})$, and output s_i as one of the maximal points.
Step 4: If $i \neq n$, then $i \leftarrow i + 1$ and go to Step 3.

Kung et al. [24] maintain the set M by an AVL tree for insertion, deletion, successor and predecessor operations. The time complexity of their algorithm is $\mathcal{O}(n \log n)$ for $d = 3$.

3. Our diagonal-based algorithm

Our LCPS algorithm is based on the diagonal concept, inspired by the LCS algorithm proposed by Nakatsu et al. [22]. The time and space complexities of our LCPS algorithm are $\mathcal{O}(L(m-L)R \log n)$ and $\mathcal{O}(RL)$, respectively, where m, n, and L denote the lengths of input sequences A, B, and LCPS(A, B), respectively. Clearly, our LCPS algorithm is more efficient when L is close to m, or L is very small.

Table 3
The construction of $D_{i,s}$ in our LCPS algorithm with A = cbccbaabb and B = bbccabbca.

Round r \ s	0	1	2	3
1	$D_{0,0}$ $\langle 0,0,0\rangle$	$D_{1,1}$ $\langle 6,3,2\rangle$	$D_{2,2}$ $\langle 8,6,3\rangle$	
2	$D_{1,0}$ $\langle 0,0,0\rangle$	$D_{2,1}$ $\langle 6,3,2\rangle$ $\langle 1,1,3\rangle$	$D_{3,2}$ $\langle 8,6,3\rangle$ ~~$\langle 7,4,6\rangle$~~ $\langle 6,3,6\rangle$	
3	$D_{2,0}$ $\langle 0,0,0\rangle$	$D_{3,1}$ $\langle 6,3,2\rangle$ $\langle 1,1,3\rangle$ ~~$\langle 6,3,2\rangle$~~	$D_{4,2}$ $\langle 8,6,3\rangle$ $\langle 6,3,6\rangle$ ~~$\langle 6,3,6\rangle$~~	
4	$D_{3,0}$ $\langle 0,0,0\rangle$	$D_{4,1}$ $\langle 6,3,2\rangle$ $\langle 1,1,3\rangle$ ~~$\langle 6,3,2\rangle$~~	$D_{5,2}$ $\langle 8,6,3\rangle$ ~~$\langle 6,3,6\rangle$~~ $\langle 2,2,4\rangle$	$D_{6,3}$ $\langle 3,5,5\rangle$

The LCPS problem can be solved by the straightforward DP method for solving the FLCS (LCS with four strings) problem. Let A, $\overline{A}$, B and $\overline{B}$ be the four input strings of the FLCS problem. And let $U(i,x,y,z)$ denote the FLCS length of $A_{1..i}$, $\overline{A}_{1..x}$, $B_{1..y}$ and $\overline{B}_{1..z}$. After all lattice cells of U are calculated by DP, the length L of LCPS(A,B) can be obtained as follows.

$$L = \max \begin{cases} U(i,x,y,z)\times 2 & \text{if } i+x=m \text{ and } y+z=n; \\ U(i,x,y,z)\times 2-1 & \text{if } i+x=m+1 \text{ and } y+z=n+1. \end{cases} \tag{3}$$

In Eq. (3), the first term means that the LCPS length is even, and the second term represents that the LCPS length is odd, where one match character overlaps at the same position of A or B. Both time and space complexities of this FLCS method for solving LCPS are $\mathcal{O}(m^2n^2)$.

Now, we realize the above method with the diagonal concept. We first define a dominant set $D_{i,s}$ consisting of FLCS solutions, which are 3-tuple elements with minima.

Definition 1. (Domination) For a pair of 3-tuples $k=\langle x,y,z\rangle$ and $k'=\langle x',y',z'\rangle$, we say that *$k$ dominates k'* if $x\le x'$, $y\le y'$ and $z\le z'$. In a set of 3-tuples, an element is called a *dominant* if it is not dominated by any other elements in the set. In a *dominant set*, each element is a *dominant*.

Definition 2. (FLCS solution $D_{i,s}$) A 3-tuple $\langle x,y,z\rangle \in D_{i,s}$ means that $|\text{FLCS}(A_{1..i},\overline{A}_{1..x},B_{1..y},\overline{B}_{1..z})| = U(i,x,y,z)=s$ and $\langle x,y,z\rangle$ is a dominant in $D_{i,s}$, where $i+x\le m+1$ and $y+z\le n+1$.

Note that $\overline{A}_{1..x}=\overline{A_{m-x+1..m}}=a_m a_{m-1}\cdots a_{m-x+1}$ and $\overline{B}_{1..z}=\overline{B_{n-z+1..n}}=b_n b_{n-1}\cdots b_{n-z+1}$. For $\langle x,y,z\rangle\in D_{i,s}$, there exists a *common palindromic subsequence* (CPS) of length $l=2s$ or $l=2s-1$ (not necessarily the longest) in $A'=A_{1..i}+A_{m-x+1..m}$ and $B'=B_{1..y}+B_{n-z+1..n}$, which is established by these s match characters, and $a_{m-x+1}=b_y=b_{n-z+1}$. That is, in the CPS solution, the numbers of characters which have been used in sequences $\overline{A}$, B, and $\overline{B}$ are x, y and z, respectively. Note that it is possible for either $a_i=a_{m-x+1}$ or $a_i\ne a_{m-x+1}$. For example, A = cbccbaabb and B = bbccabbca, in Table 3, $\langle 2,2,4\rangle\in D_{5,2}$ means that $|\text{FLCS}(A_{1..5},\overline{A}_{1..2},B_{1..2},\overline{B}_{1..4})| = |\text{FLCS}(A_{1..5},\overline{A_{8..9}},B_{1..2},\overline{B_{6..9}})|=2$, with 2 match characters bb. Then, $|\text{CPS}(a_1\cdots a_5a_8a_9,\ b_1b_2b_6\cdots b_9)|=4$ is obtained from these 2 match characters. Furthermore, a dominant 3-tuple has better potential to develop a longer CPS solution. For example, in $D_{5,2}$ of Table 3, $\langle 6,3,6\rangle$ is dominated by $\langle 2,2,4\rangle$. Thus, $\langle 6,3,6\rangle$ is removed since all 3-tuples in $D_{5,2}$ are dominants.

To construct $D_{i,s}$, we need two main operations, extension and domination (minima-finding), defined as follows.

Definition 3. (Extend$(\cdot)$) The extension of all 3-tuples from $D_{i-1,s-1}$ is denoted as Extend$(D_{i-1,s-1},i)$. For $\langle x,y,z\rangle\in D_{i-1,s-1}$, Extend$(\langle x,y,z\rangle,i)$ is to obtain $\langle x',y',z'\rangle$ such that $a_i=a_{m-x'+1}=b_{y'}=b_{n-z'+1}$ and x',y',z' are the smallest, where $x+1\le x'\le m$, $y+1\le y'\le n$, $z+1\le z'\le n$, $i+x'\le m+1$ and $y'+z'\le n+1$.

Definition 4. (Dominate$(\cdot)$) The removal of dominated 3-tuples in $D_{i,s}$ is denoted as Dominate$(D_{i,s})$.

$D_{i,s}$ is obtained by Dominate$(D_{i-1,s}\cup$ Extend$(D_{i-1,s-1},i))$. And $D_{i,s}$ can be recognized as a *3-dimensional minima* set, mentioned in Section 2.4. So, the 3-dimensional minima finding algorithm [24] can be applied to performing Dominate$(\cdot)$.

For initialization, $D_{i,0}=\{\langle 0,0,0\rangle\}$ for $0\le i\le m$; $D_{i,s}=\emptyset$ for $0\le i\le m$ and $s=i+1$. The construction of $D_{i,s}$ is done by the diagonal method with the scheme of round by round. In round r, we compute $D_{r,1}$, $D_{r+1,2}$, $D_{r+2,3}$, $\cdots$ sequentially.

Algorithm 1 Computing the length of LCPS.

Input: Sequences $A = a_1a_2a_3 \dots a_m$, $B = b_1b_2b_3 \dots b_n$, where $m \le n$.
Output: Length of LCPS(A, B)

$D_{i,0} \leftarrow \{\langle 0,0,0\rangle\}$ for $0 \le i \le m$; $D_{i,s} \leftarrow \emptyset$ for $0 \le i \le m$ and $s = i+1$
$L \leftarrow 0$ ▷ L = |LCPS(A, B)|
for $r = 1 \to m$ **do** ▷ round r
  $s \leftarrow 0$
  for $i = r \to m$ **do**
    $s \leftarrow s + 1$
    $D' \leftarrow$ EXTEND$(D_{i-1,s-1}, i)$ ▷ extend 3-tuples from $D_{i-1,s-1}$
    Perform $D_{i-1,s} \cup D'$. When two same 3-tuples are with different *even*/*odd* flags, preserve the one with an *even* flag. Put the union result into D''.
    $D_{i,s} \leftarrow$ DOMINATE(D'') ▷ minima set of $D_{i-1,s} \cup D'$
    if $D_{i,s}$ is empty **then**
      $i \leftarrow i - 1$, $s \leftarrow s - 1$
      break
  if $\exists \langle x, y, z\rangle \in D_{i,s} \ni \langle x, y, z\rangle$ is marked with an *even* flag **then**
    $L \leftarrow 2 \times s$ ▷ L is even
  else
    $L \leftarrow 2 \times s - 1$ ▷ the last match overlaps
  if $m - r \le L$ **then**
    break
return L

Function 1 Extension of the set D with a_i.

function EXTEND(set D, index i)
  ▷ Build the arrays of Next$\overline{A}[\alpha][\ ]$, Next$B[\alpha][\ ]$ and Next$\overline{B}[\alpha][\ ]$ for each $\alpha \in \Sigma$ in the preprocessing stage
  $D' \leftarrow \emptyset$
  for each $\langle x, y, z\rangle \in D$ **do**
    $x' \leftarrow$ Next$\overline{A}[a_i][x]$ ▷ next match a_i after position x in $\overline{A}$
    $y' \leftarrow$ Next$B[a_i][y]$ ▷ next match a_i after position y in B
    $z' \leftarrow$ Next$\overline{B}[a_i][z]$ ▷ next match a_i after position z in $\overline{B}$
    if $i + x' \le m + 1$ and $y' + z' \le n + 1$ **then**
      if $i + x' = m + 1$ or $y' + z' = n + 1$ **then** ▷ overlapped match
        Mark $\langle x', y', z'\rangle$ with an *odd* flag
      else
        Mark $\langle x', y', z'\rangle$ with an *even* flag
      Insert $\langle x', y', z'\rangle$ into D'
  return D'

For example, in round $r = 1$ (first round), $D_{1,1}$, $D_{2,2}$, $D_{3,3}, \cdots$ are calculated. Next, in round $r = 2$, we calculate $D_{2,1}$, $D_{3,2}$, $D_{4,3}, \cdots$.

Now, we illustrate the construction process of $D_{i,s}$ with an example, as shown in Table 3, for A = `cbccbaabb` and B = `bbccabbca`. In round $r = 1$, we start from the character at position $i = 1$ of A (a_1 = `c`). The first matches of `c` in $\overline{A}$, B and $\overline{B}$ are a_4, b_3 and b_8, where 6, 3 and 2 characters are used in $\overline{A}$, B and $\overline{B}$, respectively. Thus, we get $D_{1,1} = \{\langle 6,3,2\rangle\}$, extended from $\langle 0,0,0\rangle$ in $D_{0,0}$. Next, we extend $a_2 =$ `b` in A from $\langle 6,3,2\rangle$, and find that the next matches of `b` in $\overline{A}$, B and $\overline{B}$ are a_2, b_6, and b_7, with 8, 6 and 3 used characters, respectively. So we have $D_{2,2} = \{\langle 8,6,3\rangle\}$. We cannot extend $\langle 8,6,3\rangle$ anymore since no next match of $a_3 =$ `c` can be found in the remaining substrings of A and B. So the first round stops, and we have CPS length 3 (`cbc`) with 2 match characters `cb`. The length is odd because A and $\overline{A}$ have the same match position at $a_2 =$ `b` in A and $\overline{A}$.

In round $r = 2$, we start from the second character $a_2 =$ `b` of A, and find the next matches in $\overline{A}$, B and $\overline{B}$. We get a_9, b_1 and b_7, respectively, where 1, 1 and 3 characters are used in $\overline{A}$, B and $\overline{B}$, respectively. So we have $D_{2,1} = D_{1,1} \cup \{\langle 1,1,3\rangle\} = \{\langle 6,3,2\rangle, \langle 1,1,3\rangle\}$. Next, by the extension of $D_{2,1}$ with $a_3 =$ `c` of A, we get $D' =$ EXTEND$(D_{2,1}, 3) = \{\langle 7,4,6\rangle, \langle 6,3,6\rangle\}$. Then, with the domination operation, we obtain $D_{3,2} =$ DOMINATE$(D_{2,2} \cup D') = \{\langle 8,6,3\rangle, \langle 6,3,6\rangle\}$ since $\langle 7,4,6\rangle$ is dominated by $\langle 6,3,6\rangle$, and it is removed.

Finally, we get $D_{6,3} = \{\langle 3,5,5\rangle\}$ in round $r = 4$. Accordingly, the LCPS length is 5 (`bbabb`), which is odd because $\langle 3,5,5\rangle$ matches the same position at b_5 in B and $\overline{B}$. The LCPS content can be obtained by tracing back from $\langle 3,5,5\rangle$ through $\langle 2,2,4\rangle$ to $\langle 1,1,3\rangle$. The whole algorithm ends since we cannot find any CPS longer than 5 in the next round. That is, only 5 characters of A are remained to extend in round 5.

In addition, each 3-tuple is marked by an *odd* flag if the match overlaps at the same position of A or B; otherwise, it is marked with an *even* flag. For example, $\langle 6,3,2\rangle$ in $D_{1,1}$ is marked with an *even* flag and $\langle 8,6,3\rangle$ in $D_{2,2}$ is marked with an *odd* flag. In round $r = 4$, $\langle 6,3,2\rangle$ is extended from $\langle 0,0,0\rangle$ again and it is marked with an *odd* flag. When uniting $D_{3,1}$ and the new extension, the *even* flag for $\langle 6,3,2\rangle$ is preserved for getting a longer solution.

The pseudo code of our algorithm is presented in Algorithm 1.

Theorem 1. *$D_{i,s}$ can be obtained by* DOMINATE$(D_{i-1,s} \cup$ EXTEND$(D_{i-1,s-1}, i))$.

Function 2 Finding the dominant (minima) set of set D.

```
1: function DOMINATE(set D)
2:     Sort D by the x-coordinate in ascending order          ▷ x_1 ≤ x_2 ≤ ··· ≤ x_|D|
3:     D' ← ∅                                                 ▷ minima set of D
4:     M ← {⟨y_1, z_1⟩}, where d_1 = ⟨x_1, y_1, z_1⟩          ▷ M is the 2-D minima set
5:     for i = 2 → |D| do
6:         if ⟨y_i, z_i⟩ is a dominant in M then              ▷ d_i = ⟨x_i, y_i, z_i⟩
7:             M ← MINIMA(M ∪ {⟨y_i, z_i⟩})                   ▷ update the minima set M
8:             Insert d_i = ⟨x_i, y_i, z_i⟩ into D'
9:     return D'
```

Proof. It is proved by induction on i and s. For $1 \le i = s \le m$, it can be easily seen that $D_{i,s}$ is obtained from $\text{EXTEND}(D_{i-1,s-1}, i)$ since $D_{i-1,s}$ is empty.

Assume that the induction hypotheses $D_{i-1,s}$ and $D_{i-1,s-1}$ are true for $i \ge s$. Now, we want to build $D_{i,s}$. Since $\text{EXTEND}(D_{i-1,s-1}, i)$ is to examine one more character a_i, we can increase the solution length if the extension is successful. So $D_{i-1,s} \cup \text{EXTEND}(D_{i-1,s-1}, i)$ contains all possible solutions corresponding to s. After $\text{DOMINATE}(\cdot)$, we get the dominant elements in $D_{i,s}$. Thus, the theorem holds. □

To analyze the time complexity of our algorithm, we first observe that the number of distinct 3-tuples of one column in Table 3 is bounded, as described in the following theorem.

Theorem 2. *$|D_{s,s} \cup D_{s+1,s} \cup D_{s+2,s} \cup \cdots| = \mathcal{O}(\frac{Rn}{|\Sigma|})$ in the average case, for some fixed value of s, where R denotes the number of match pairs between the two input sequences.*

Proof. For each 3-tuple $\langle x, y, z\rangle \in D_{i,s}$, we have $a_{m-x+1} = b_y = b_{n-z+1}$. $\langle x, y\rangle$ represents a match pair of $\overline{A}$ and B. So, the number of possible distinct $\langle x, y\rangle$ is R. For each distinct $\langle x, y\rangle$, the number of possible z is the number of positions with $b_y = b_{n-z+1}$. It is equal to $\frac{n}{|\Sigma|}$ in average, where $|B| = n$. Thus, the theorem holds. □

Theorem 3. *Algorithm 1 solves the LCPS problem with $\mathcal{O}(L(m-L)R\log n)$ time and $\mathcal{O}(RL)$ space in the worst case, and $\mathcal{O}(L(m-L)mn\log n/|\Sigma|)$ time in the average case.*

Proof. For any two 3-tuples in each $D_{i,s}$, their x and y values cannot be equal at the same time. Otherwise, there should be a domination between the two 3-tuples. So, $|D_{i,s}| \le R$. The extension of $D_{i,s}$ requires $\mathcal{O}(|D_{i,s}|) = \mathcal{O}(R)$ time, where R is $\mathcal{O}(mn) = \mathcal{O}(n^2)$ in the worst case. The operation DOMINATE can be implemented by the 3-dimensional minima finding algorithm with $\mathcal{O}(|D_{i,s}|\log|D_{i,s}|)$ time. Thus, the time required for $\text{DOMINATE}(D_{i,s})$ is $\mathcal{O}(R\log R) = \mathcal{O}(R\log n)$.

The algorithm terminates when round $r \ge m - L$. Each round performs the extension and domination at most $\lceil \frac{L}{2} \rceil$ times. Hence, the LCPS problem can be solved in $\mathcal{O}(L(m-L)R\log n)$ time in the worst case, and $\mathcal{O}(L(m-L)mn\log n/|\Sigma|)$ time in the average case, since $R = \mathcal{O}(\frac{mn}{|\Sigma|})$ in average.

The space of $D_{i,s}$ in each round can be reused in the next round, so the algorithm only requires the space of one round. The number of $D_{i,s}$ in a round is $\mathcal{O}(L)$, and thus the space complexity of the algorithm is $\mathcal{O}(RL)$. □

4. Experimental results

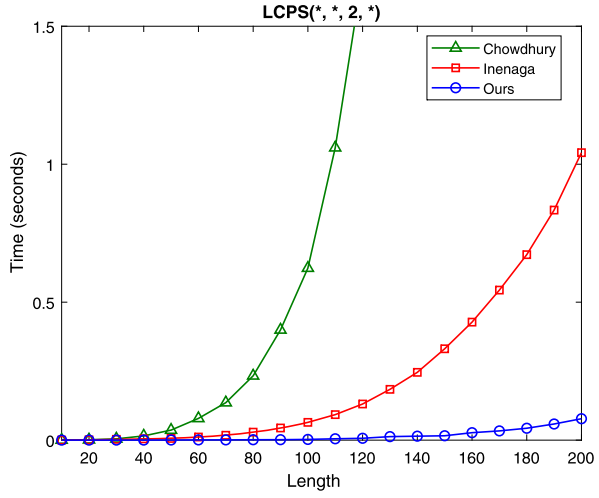
In the experiments, we compare the execution time of our algorithm, the algorithm proposed by Chowdhury et al. [9], and the algorithm proposed by Inenaga and Hyyrö [17]. Each experiment is performed 100 times to get the average execution time. These algorithms are implemented in Java by Eclipse 4.6.3, and they are tested on a computer with 64-bit Windows 10 OS, CPU clock rate of 3.00 GHz (Intel i5-7400 CPU) and RAM with 16 GB.

The following are the algorithms for comparison in the experiments.

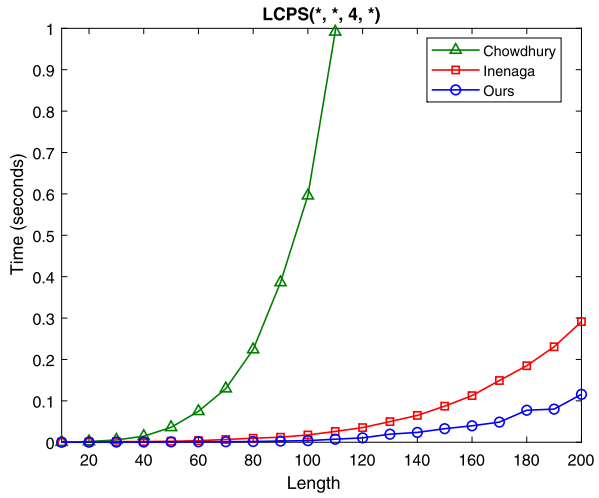
- **Chowdhury**: The DP algorithm proposed by Chowdhury et al. [9] with $\mathcal{O}(m^2n^2)$ time.
- **Inenaga**: The algorithm for the maximum depth nesting rectangle structure proposed by Inenaga and Hyyrö [17] with $\mathcal{O}(n + R^2|\Sigma|)$ time, where R is $\mathcal{O}(mn/|\Sigma|)$ in the average case.
- **Ours**: Our diagonal-based algorithm with $\mathcal{O}(L(m-L)R\log n)$ time in the worst case, and $\mathcal{O}(L(m-L)mn\log n/|\Sigma|)$ time in the average case.

We use a 4-tuple $(|A|, |B|, |\Sigma|, algo)$ to represent the parameters in each performance chart. For example, $(200, 200, 2, *)$ means that $|A| = |B| = 200$, $|\Sigma| = 2$, and "$*$" is a wildcard representing all of the three algorithms.

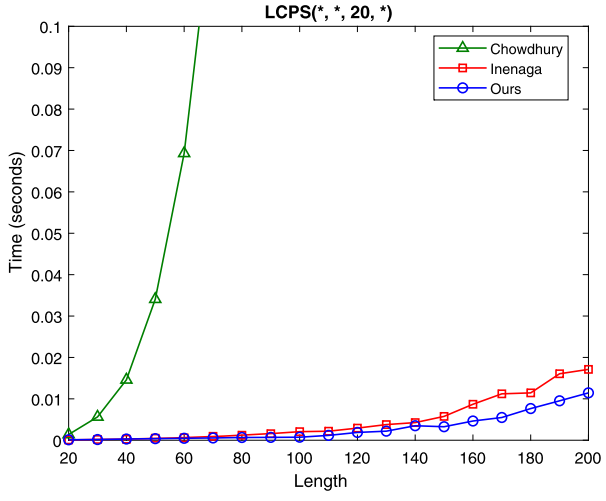
The first experiment is to test the algorithms for various values of $|\Sigma| \in \{2, 4, 20\}$, with the input lengths n ranging from 10 to 200 if $n \ge |\Sigma|$. The input sequences A and B are generated with a fully random manner and they are picked up if the input sequences involve all characters $\sigma \in \Sigma$. The results of the first experiment are shown in Fig. 4.



(a)



(b)



(c)

Fig. 4. The execution time (in seconds) of the three algorithms for various $|\Sigma|$ with input lengths ranging from 10 to 200, where $m = n$. (a) $|\Sigma| = 2$. (b) $|\Sigma| = 4$. (c) $|\Sigma| = 20$.

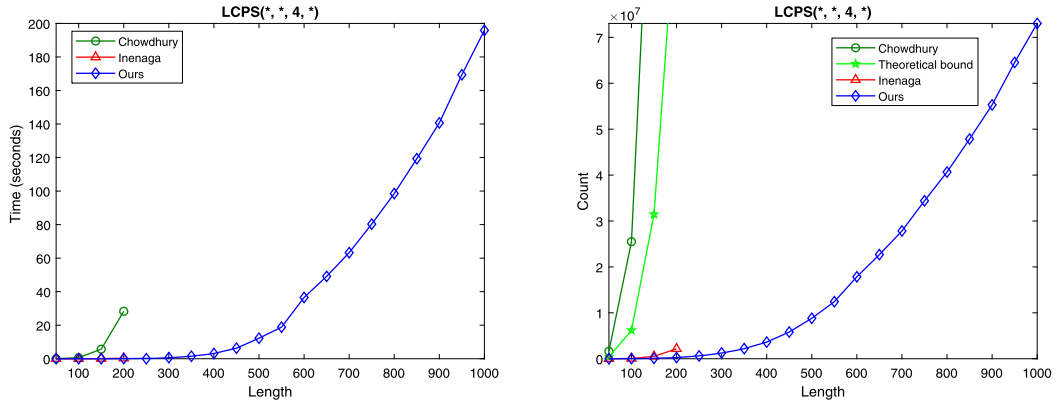


Fig. 5. The comparisons of the three algorithms for the real data with $|\Sigma| = 4$ and input lengths ranging from 50 to 1000, where $m = n$. (a) The execution time (in seconds). (b) The amount of calculations.

The algorithm of Chowdhury et al. solves the LCPS problem with the DP method. Their computational amount is larger than the number R of match pairs. As Fig. 4 shows, the algorithm of Chowdhury et al. takes more time than the others. The algorithm of Inenaga and Hyyrö and our algorithm are both related to the number of match pairs. When $|\Sigma|$ is getting larger, the number of match pairs becomes smaller. Thus, the two algorithms are much more efficient than the algorithm of Chowdhury et al. when $|\Sigma|$ is large. The algorithm of Inenaga and Hyyrö extends all possible rectangles that may form a palindrome, whose size is $\mathcal{O}(R^2)$. Our algorithm extends only the currently optimal 3-tuples. The calculation of our algorithm is practically less than the algorithm of Inenaga and Hyyrö. Thus, our algorithm has better performance than the others in almost all cases.

In the second experiment, we test the execution time and the computational amount of the three algorithms with real bio-sequences of lengths ranging from 50 to 1000. We take the RefSeq transcripts of the ACIN-1 as our input data (https://www.ncbi.nlm.nih.gov/gene/22985). ACIN-1 stands for apoptotic chromatin condensation inducer 1, and we choose the mRNA, transcript variants 1 through 5 of ACIN-1 to be our sample dataset. The lengths of the transcript variants 1 through 5 are 4951, 4909, 4828, 2456, and 2518, respectively, and $\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$, where $|\Sigma| = 4$.

The results of the second experiment are shown in Fig. 5. The algorithm of Chowdhury et al. and the algorithm of Inenaga and Hyyrö are tested for input lengths ranging from 50 to 200. Because the space complexities of the two algorithms are $\mathcal{O}(n^4)$ and $\mathcal{O}(R^2)$, respectively, the algorithms need arrays with sizes n^4 and R^2, respectively. Thus, we cannot perform the experiments for the two algorithms with input lengths more than 200 in the experimental environment. These results are similar to the first experimental result with $|\Sigma| = 4$.

Our algorithm has better performance than the others in either random or real data. In our algorithm, each $D_{i,s}$ only keeps the dominant 3-tuples. And the number of 3-tuples in most $D_{i,s}$ is much smaller than R in the simulations. Thus, the amount of calculations in our algorithm is much smaller than the theoretical complexity of our algorithm, which is $\mathcal{O}(L(m-L)R\log n)$, where R is $\mathcal{O}(\frac{mn}{|\Sigma|})$ in the average case.

5. Conclusion

This paper proposes a diagonal-based algorithm for solving the longest common palindromic subsequence (LCPS) problem with $\mathcal{O}(L(m-L)R\log n)$ time and $\mathcal{O}(RL)$ space in the worst case, where m, n, and L denote the lengths of A, B, and LCPS(A, B), respectively, and R denotes the number of match pairs between A and B. In the future, the theoretical time complexity may be improved by reducing the domination and the upper bound of $|D_{i,s}|$.

The LCPS problem can help to solve the *longest common square subsequence* (LCSqS) problem [18]. An input sequence can be partitioned into a prefix and a suffix, next we reverse the suffix sequence, and then we concatenate the prefix and reversed suffix to form a new sequence. Thus, we can obtain the LCSqS length of the input sequences by finding the even LCPS length of the two new sequences. Inoue et al. [18] proposed an LCSqS algorithm with $\mathcal{O}(n + R^3|\Sigma|)$ time. As a result, we can solve the LCSqS problem in $\mathcal{O}(mnL(m-L)R\log n)$ time by solving all of the partitioned subproblems with our LCPS algorithm.

Furthermore, the *cyclic longest common palindromic subsequence* problem is a variant of the LCPS problem. The cyclic version of the LCPS problem can separate the problem into subproblems by cutting the two cyclic sequences with $\mathcal{O}(mn)$ different ways. And each subproblem can be solved by the LCPS algorithm. In the cyclic problem, shifting a cutting position is similar to appending a new character to the sequence. Therefore, the online version of the LCPS problem deserves the further study.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Alatabbi, C.S. Iliopoulos, M.S. Rahman, Maximal palindromic factorization, in: Proceedings of the Prague Stringology Conference (PSC 2013), Prague, Czech, Jan. 2013, pp. 70–77.
[2] H.Y. Ann, C.B. Yang, C.T. Tseng, C.Y. Hor, Fast algorithms for computing the constrained LCS of run-length encoded strings, Theor. Comput. Sci. 432 (2012) 1–9.
[3] M.J. Atallah, F. Kerschbaum, W. Du, Secure and private sequence comparisons, in: Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society, New York, USA, 2003, pp. 39–44.
[4] S.W. Bae, I. Lee, On finding a longest common palindromic subsequence, Theor. Comput. Sci. 710 (2018) 29–34.
[5] B.S. Baker, R. Giancarlo, Longest common subsequence from fragments via sparse dynamic programming, in: Proceedings of the 6th Annual European Symposium on Algorithms, Venice, Italy, 1998, pp. 79–90.
[6] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proceedings of the 7th International Symposium on String Processing and Information Retrieval, A Coruña, Spain, Sept. 2000, pp. 39–48.
[7] K. Borozdin, D. Kosolobov, M. Rubinchik, A.M. Shur, Palindromic length in linear time, in: Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), Warsaw, Poland, 2017, pp. 23:1–23:12.
[8] H.T. Chan, H.T. Chiu, C.B. Yang, Y.H. Peng, The generalized definitions of the two-dimensional largest common substructure problems, Algorithmica 82 (2020) 2039–2062.
[9] S.R. Chowdhury, M.M. Hasan, S. Iqbal, M.S. Rahman, Computing a longest common palindromic subsequence, Fundam. Inform. 129 (4) (Oct. 2014) 329–340.
[10] G. Fici, T. Gagie, J. Kärkkäinen, D. Kempa, A subquadratic algorithm for minimum palindromic factorization, J. Discret. Algorithms 28 (2014) 41–48.
[11] A. Fuglsang, The relationship between palindrome avoidance and intragenic codon usage variations: a Monte Carlo study, Biochem. Biophys. Res. Commun. 316 (May 2004) 755–762.
[12] R. Groult, E. Prieur, G. Richomme, Counting distinct palindromes in a word in linear time, Inf. Process. Lett. 110 (20) (2010) 908–912.
[13] M.M. Hasan, A.S.M.S. Islam, M.S. Rahman, A. Sen, Palindromic subsequence automata and longest common palindromic subsequence, Math. Comput. Sci. 11 (2) (June 2017) 219–232.
[14] M. Hoffmann, J. Rychlewski, Searching for palindromic sequences in primary structure of proteins, Comput. Methods Sci. Technol. 5 (Jan. 1999) 21–24.
[15] T. I, S. Sugimoto, S. Inenaga, H. Bannai, M. Takeda, Computing palindromic factorizations and palindromic covers on-line, in: Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014), Moscow, Russia, 2014, pp. 150–161.
[16] C.S. Iliopoulos, M.S. Rahman, New efficient algorithms for the LCS and constrained LCS problems, Inf. Process. Lett. 106 (1) (2008) 13–18.
[17] S. Inenaga, H. Hyyrö, A hardness result and new algorithm for the longest common palindromic subsequence problem, Inf. Process. Lett. 129 (2018) 11–15.
[18] T. Inoue, S. Inenaga, H. Hyyrö, H. Bannai, M. Takeda, Computing longest common square subsequences, in: Proceedings of the 29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018), Qingdao, China, 2018, pp. 15:1–15:13.
[19] M. Kutz, G.S. Brodal, K. Kaligosi, I. Katriel, Faster algorithms for computing longest common increasing subsequences, J. Discret. Algorithms 9 (4) (2011) 314–325.
[20] T.W. Liang, C.B. Yang, K.S. Huang, A fast algorithm for the longest common palindromic subsequence problem, in: Proceedings of the 37th Workshop on Combinatorial Mathematics and Computation Theory, Kaohsiung, Taiwan, 2020, pp. 128–133.
[21] S.F. Lo, K.T. Tseng, C.B. Yang, K.S. Huang, A diagonal-based algorithm for the longest common increasing subsequence problem, Theor. Comput. Sci. 815 (2020) 69–78.
[22] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inform. 18 (2) (Nov. 1982) 171–179.
[23] A.H.L. Porto, V.C. Barbosa, Finding approximate palindromes in strings, Pattern Recognit. 35 (11) (2002) 2581–2591.
[24] H.T. Kung, F. Luccio, F.P. Preparata, On finding the maxima of a set of vectors, J. ACM 22 (Oct. 1975) 469–476.
[25] Y.T. Tsai, The constrained longest common subsequence problem, Inf. Process. Lett. 88 (2003) 173–176.
[26] K.T. Tseng, D.S. Chan, C.B. Yang, S.F. Lo, Efficient merged longest common subsequence algorithms for similar sequences, Theor. Comput. Sci. 708 (2018) 75–90.
[27] I.H. Yang, C.P. Huang, K.M. Chao, A fast algorithm for computing a longest common increasing subsequence, Inf. Process. Lett. 93 (5) (2005) 249–253.