Theoretical Computer Science 815 (2020) 69–78

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

A diagonal-based algorithm for the longest common increasing subsequence problem [☆]

Shou-Fu Lo [a], Kuo-Tsung Tseng [b,d], Chang-Biau Yang [a,*], Kuo-Si Huang [c,d]

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

A R T I C L E I N F O

Article history:
Received 8 October 2019
Received in revised form 24 January 2020
Accepted 17 February 2020
Available online 20 February 2020
Communicated by R. Giancarlo

Keywords:
LCS
LCIS
LCWIS
Diagonal algorithm
Similar sequences

A B S T R A C T

The *longest common increasing subsequences* (LCIS) problem is to find out a common increasing subsequence with the maximal length of two given sequences A and B. In this paper, we propose an algorithm for solving the LCIS problem with $O((n + L(m - L))\log\log|\Sigma|)$ time and $O(n)$ space, where m and n denote the lengths of A and B, respectively, $m \leq n$, L denotes the LCIS length, and Σ denotes the alphabet set. The main idea of our algorithm is to extend the answer from some previously feasible solutions, in which the domination function is invoked. To accomplish the extension and domination functions, the data structure of the van Emde Boas tree is utilized. From the time complexity, it is obvious that our algorithm is extremely efficient when L is very small or L is close to m. Some experiments are performed to demonstrate the efficiency of our algorithm.

1. Introduction

The *longest common increasing subsequences* (LCIS) problem is a combined variant of the *longest common subsequence* (LCS) problem and the *longest increasing subsequence* (LIS) problem. For given two sequences A and B, the LCIS problem aims to find the common subsequence such that it is increasing in both A and B, and it is the longest. The LCS can be used to measure the similarity of two sequences in many applications, such as computational biology, pattern matching, plagiarism detection, and voice recognition. Many LCS algorithms have been proposed by researchers [1–5] in the past decades.

For solving the LCS problem, in 1974, Wagner and Fischer presented a dynamic programming (DP) algorithm with $O(mn)$ time and space [5], where m and n denotes the lengths of the input sequences A and B, respectively. Then, in 1977, Hunt and Szymanski presented a practically faster algorithm in $O((R + m)\log m)$ time [3], where R is the total number of match pairs between two given sequences A and B. This algorithm is efficient when R is small; that is, A and B are dissimilar. In 1982, Nakatsu et al. proposed a diagonal algorithm in $O(n(m - L))$ time, where L denotes the LCS length, which is more efficient if two given sequences A and B are similar [4]. Note that R is different from L. For example, suppose that $A = \langle a, d, c, b\rangle$ and $B = \langle a, c, a, d\rangle$. Then $R = 4$ and $L = 2$. The diagonal concept was successfully applied to solving the merged longest common subsequence problem [6].

☆ This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 107-2221-E-110-033.
* Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.B. Yang).

https://doi.org/10.1016/j.tcs.2020.02.024
0304-3975/© 2020 Elsevier B.V. All rights reserved.

Table 1
The time complexities of the LIS, LCIS and LCWIS algorithms. Here, L denotes the length of the answer, R denotes the number of match pairs in A and B, and Σ denotes the alphabet set.

LIS			
Year	Author(s)	Time complexity	Keywords
1961	Schensted [7]	$O(n\log n)$	Binary search
1973	Knuth [8]	$O(n\log n)$	Binary search
1975	Fredman [19]	$O(n\log n)$	Pruned comparison tree
1977	Hunt and Szymanski [3]	$O(n\log\log n)$	Match pair
1977	Bespamyatnikh and Segal [9]	$O(n\log\log n)$	van Emde Boas tree
1977	Crochemore et al. [20]	$O(n\log\log L)$	Split block Priority queue van Emde Boas tree

LCIS			
Year	Author(s)	Time complexity	Keywords
2005	Yang et al. [12]	$O(mn)$	Dynamic programming
2006	Sakai [13]	$O(mn)$	Divide-and-conquer
2006	Brodal et al. [14,15]	$O((n+mL)\log\log\lvert\Sigma\rvert+Sort)$	Divide-and-conquer Bounded heap van Emde Boas tree
2007	Chan et al. [16]	$O(min(R\log L, nL+R)\log\log n+Sort(n))$	Match pair Binary search van Emde Boas tree
2018	Cai et al. [17]	$O(mn)$	No divide-and-conquer
	This paper	$O((n+L(m-L))\log\log\lvert\Sigma\rvert)$	Diagonal van Emde Boas tree

LCWIS for 3-letter			
Year	Author(s)	Time complexity	Keywords
2006	Brodal et al. [14,15]	$O(min\{n+m\log m, n\log\log n\})$	Split diagram Priority queue van Emde Boas tree
2013	Duraj [18]	$O(m+n)$	Update and add

The LIS also has a rich history. When the input is an arbitrary sequence of n numbers, Schensted [7] and Knuth [8] proposed an algorithm with $O(n\log n)$ time. Furthermore, if the input is a permutation of $\{1,2,\ldots,n\}$, Hunt and Szymanski [3], and Bespamyatnikh and Segal [9] both presented algorithms with the same time complexity $O(n\log\log n)$.

The LCIS problem can also be used in some applications, such as pattern recognition and the whole genome alignment [10,11]. Its goal is to align two or more genomes, and to find the longest sets of maximal unique matching subsequences whose sequences exist in ascending order. Some algorithms have been proposed for the LCIS problem. In 2005, Yang et al. designed a dynamic programming method in $O(mn)$ time and space [12]. In 2006, Sakai first designed an algorithm with the linear space complexity [13]. He utilized the divide-and-conquer approach used for the LCS problem [2], which improved the algorithm of Yang et al. In the same year, Brodal et al. [14,15] gave an efficient algorithm with $O((n+mL)\log\log|\Sigma|+Sort)$ time when L is relatively small, where L denotes the LCIS length and Σ denotes the alphabet set of A and B. Next year, Chan et al. also proposed a fast algorithm with $O((nL+R)\log\log n+Sort(n))$ time [16], where R denotes the total number of match pairs between the two sequences. It is obvious that the algorithm is efficient when R is small. In 2018, a linear space algorithm without utilizing the divide-and-conquer concept was proposed by Cai et al. [17].

There is a variant of LCIS called the *longest common weakly-increasing (or non-decreasing) subsequence* (LCWIS) for 3-letter. For example, assume that $A=\langle 0,1,0,1,1,2\rangle$ and $B=\langle 0,1,1,2,1,2\rangle$. Thus, we have $LCWIS(A,B)=\langle 0,1,1,1,2\rangle$. In 2006, Brodal et al. [14,15] proposed an algorithm with $O(min\{n+m\log m, n\log\log n\})$ time. In 2013, Duraj [18] improved the Brodal algorithm to $O(m+n)$ time. The time complexities of these related algorithms are shown in Table 1.

The rest of this paper is organized as follows. In Section 2, we first give the formal definition of the LCIS problem. Then, we introduce the previous algorithm for the LCIS problem, and the LCS algorithm of Nakatsu et al., which is the inspiration of our LCIS algorithm. We present our algorithm for solving the LCIS problem in Section 3. In our algorithm, we extend the answer from some previously feasible solutions to get longer solutions, in which the domination function is used to remove the dominated solutions. The predecessor and successor operations are invoked in the extension and domination functions. The time complexity of our algorithm is $O((n+L(m-L))\log\log|\Sigma|)$ time and the space complexity is O(n). In

Table 2
An example for the LCIS algorithm of Yang et al. with $A = \langle 4, 5, 1, 4, 8\rangle$ and $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$. Here, the red underlined numbers indicate the updates.

	1	5	4	7	2	5	8	4
4			$L_{1,3}[1]=\underline{4}$	$L_{1,4}[1]=\underline{4}$	$L_{1,5}[1]=\underline{4}$	$L_{1,6}[1]=\underline{4}$	$L_{1,7}[1]=\underline{4}$	$L_{1,8}[1]=\underline{4}$
5		$L_{2,2}[1]=\underline{5}$	$L_{2,3}[1]=4$	$L_{2,4}[1]=4$	$L_{2,5}[1]=4$	$L_{2,6}[1]=4$ $L_{2,6}[2]=\underline{5}$	$L_{2,7}[1]=4$ $L_{2,7}[2]=\underline{5}$	$L_{2,8}[1]=4$ $L_{2,8}[2]=\underline{5}$
1	$L_{3,1}[1]=\underline{1}$	$L_{3,2}[1]=\underline{1}$	$L_{3,3}[1]=\underline{1}$	$L_{3,4}[1]=\underline{1}$	$L_{3,5}[1]=\underline{1}$	$L_{3,6}[1]=\underline{1}$ $L_{3,6}[2]=5$	$L_{3,7}[1]=\underline{1}$ $L_{3,7}[2]=5$	$L_{3,8}[1]=\underline{1}$ $L_{3,8}[2]=5$
4	$L_{4,1}[1]=1$	$L_{4,2}[1]=1$	$L_{4,3}[1]=1$ $L_{4,3}[2]=\underline{4}$	$L_{4,4}[1]=1$ $L_{4,4}[2]=\underline{4}$	$L_{4,5}[1]=1$ $L_{4,5}[2]=\underline{4}$	$L_{4,6}[1]=1$ $L_{4,6}[2]=\underline{4}$	$L_{4,7}[1]=1$ $L_{4,7}[2]=\underline{4}$	$L_{4,8}[1]=1$ $L_{4,8}[2]=\underline{4}$
8	$L_{5,1}[1]=1$	$L_{5,2}[1]=1$	$L_{5,3}[1]=1$ $L_{5,3}[2]=4$	$L_{5,4}[1]=1$ $L_{5,4}[2]=4$	$L_{5,5}[1]=1$ $L_{5,5}[2]=4$	$L_{5,6}[1]=1$ $L_{5,6}[2]=4$	$L_{5,7}[1]=1$ $L_{5,7}[2]=4$ $L_{5,7}[3]=\underline{8}$	$L_{5,8}[1]=1$ $L_{5,8}[2]=4$ $L_{5,8}[3]=\underline{8}$

Section 4, some experiments are performed to compare our algorithm with previously published algorithms. Finally, we give our conclusions and future works in Section 5.

2. Preliminaries

We first give the definition of the LCIS problem.

Definition 1. [12] Given two sequences $A = \langle a_1, a_2, a_3, \dots, a_m\rangle$ and $B = \langle b_1, b_2, b_3, \dots, b_n\rangle$, an *increasing subsequence* of A is formed by any arbitrary t elements chosen from A in the original order such that $a_{p_1} < a_{p_2} < a_{p_3} < \cdots < a_{p_t}$, where $1 \le p_i \le m$, $1 \le i \le t$. A *common increasing subsequence (CIS)* is a common subsequence $S = \langle s_1, s_2, s_3, \dots, s_l\rangle$ of A and B, where $s_1 < s_2 < s_3 < \cdots < s_l$. The *longest common increasing subsequence (LCIS)* is the largest one among these CIS answers, and it is denoted by $LCIS(A, B)$.

Without loss of generality, here we assume that $m \le n$. Note that the above definition can be applied to each case that the quantity order of every two elements in A or B can be determined, such as a sequence of numbers. For example, consider sequences $A = \langle 4, 5, 1, 4, 8\rangle$ and $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$, where $|A| = m = 5$ and $|B| = n = 8$. In this example, $\langle 4, 8\rangle$, $\langle 5, 8\rangle$, and $\langle 4, 5, 8\rangle$ are all common increasing subsequences of A and B. $LCIS(A, B) = \langle 4, 5, 8\rangle$ or $\langle 1, 4, 8\rangle$ is the answer of the LCIS problem.

Yang et al. designed a dynamic programming method for solving the LCIS problem in $O(mn)$ time and space [12]. Let $L_{i,j}[k]$ denote the smallest ending number of an LCIS of length k with $\langle a_1, a_2, a_3, \dots, a_i\rangle$ and $\langle b_1, b_2, b_3, \dots, b_j\rangle$, where $1 \le k$, $i \le m$ and $1 \le j \le n$. We use an example to explain their algorithm with $A = \langle 4, 5, 1, 4, 8\rangle$ and $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$, as shown in Table 2.

We first process $a_1 = 4$, and find that 4 is at b_3 of B, thus we have $L_{1,3}[1] = 4$. And then, we set $L_{1,3}[1] = 4$, $L_{1,4}[1] = 4, \dots, L_{1,8}[1] = 4$. Next, we find 5 ($a_2$) in b_2 and b_6, where b_6 can increase the LCIS length. Therefore, we get $L_{2,2}[1] = 5$, $L_{2,6}[2] = 5$, $L_{2,7}[2] = 5$ and $L_{2,8}[2] = 5$. The other values on the second row, $L_{2,3}[1], L_{2,4}[1], \dots, L_{2,8}[1]$, are set by copying the values from $L_{1,3}[1], L_{1,4}[1], \dots, L_{1,8}[1]$, respectively. In the third row, 1 (a_3) is matched in b_1. Here, 1 is better than 5 and 4, because it uses the smaller number in the same length. We can get $L_{3,1}[1] = 1$, $L_{3,2}[1] = 1$, $\dots$, $L_{3,8}[1] = 1$. Repeating the above steps, the solution can be obtained.

Nakatsu et al. proposed an algorithm with $O(n\ (m - L))$ time for the LCS problem of two sequences A and B [4], where $|A| = m \le |B| = n$, and L denotes the LCS length. This algorithm is extremely efficient for similar sequences; that is, L is close to m.

Let $d_{i,s}$ denote the smallest index j of B such that $|LCS(A_{1..i}, B_{1..j})| = s$. Nakatsu et al. solved the LCS problem with $d_{i,s}$, where $0 \le i$, $s \le m$, by the following formula:

$$d_{i,s} = \begin{cases} 0 & \text{if } s = 0; \\ \infty & \text{if } s \ge i + 1; \\ \min(j, d_{i-1,s}) & \text{if there exists the smallest } j \text{ such that } a_i = b_j, \\ & \text{where } j > d_{i-1,s-1} \text{ when } s \ge 1; \\ d_{i-1,s} & \text{otherwise.} \end{cases} \tag{1}$$

3. Our algorithm for the longest common increasing subsequence problem

We solve the LCIS problem by the diagonal-based method, inspired by the concept of Nakatsu et al. for solving the LCS problem [4]. The time complexity of our algorithm is $O((n + L(m - L)) \log\log|\Sigma|)$ time and the space complexity is $O(n)$. We first present a simple example to illustrate our concept in Section 3.1. Then, some formal analyses of our algorithm are given in Section 3.2.

Table 3
The construction of $D_{i,s}$ in the LCIS algorithm with $A = \langle 4, 5, 1, 4, 8\rangle$ and $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$.

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

3.1. An example for illustrating the concept

For solving the LCIS problem, we propose the dominating set $D_{i,s}$ as a basis of our algorithm.

Definition 2. [6] For any two 2-tuples $\langle k_1, j_1\rangle$ and $\langle k_2, j_2\rangle$, $\langle k_1, j_1\rangle \neq \langle k_2, j_2\rangle$, we say that $\langle k_1, j_1\rangle$ dominates $\langle k_2, j_2\rangle$ if $k_1 \leq k_2$ and $j_1 \leq j_2$.

Definition 3. Given two sequences $A = \langle a_1, a_2, a_3, \dots, a_m\rangle$ and $B = \langle b_1, b_2, b_3, \dots, b_n\rangle$, a 2-tuple $\langle k, j\rangle$ is in the dominating set $D_{i,s}$ if and only if there exists one CIS length s of $A_{1..i}$ and $B_{1..j}$ with ending number $b_j = k$. Every pair of 2-tuples in $D_{i,s}$ do not dominate each other.

In the above definition, $A_{1..i}$ denotes the substring of A starting from position indices 1 to i. For example, suppose that $A = \langle 4, 5, 1, 4, 8\rangle$, $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$. We have $\langle 4, 3\rangle, \langle 5, 2\rangle \in D_{2,1}$. That is, $\langle 4, 5\rangle$ and $\langle 1, 5, 4\rangle$ have CIS length 1 with ending number $b_3 = 4$, and $\langle 4, 5\rangle$ and $\langle 1, 5\rangle$ have CIS length 1 with ending number $b_2 = 5$.

We use an extension function to extend each solution (2-tuple) in $D_{i-1,s-1}$ to a longer solution in $D_{i,s}$, defined as follows.

Definition 4. Given a dominating set $D_{i-1,s-1}$, the function $\textsc{Extend}(D_{i-1,s-1})$ is to extend one element $\langle k_h, j_h\rangle \in D_{i-1,s-1}$ to another 2-tuple $\langle k, j\rangle \in D_{i,s}$, where $k_h < k = a_i$ and $j_h < j \leq n$, such that j is the smallest index after j_h for $b_j = a_i$.

Now, we illustrate our idea with an example for the construction of $D_{i,s}$, as shown in Table 3, with $A = \langle 4, 5, 1, 4, 8\rangle$ and $B = \langle 1, 5, 4, 7, 2, 5, 8, 4\rangle$. In the first round, we start with the first element a_1 of A and find the first match b_3 in B. We extend from the initial element $\langle 0, 0\rangle$ to $\langle 4, 3\rangle$ with $a_1 = b_3 = 4$. So we get $D_{1,1} = \{\langle 4, 3\rangle\}$, which means that one CIS of a_1 and $B_{1..3}$ is of length 1 with ending number $b_3 = 4$.

Next, we find a longer CIS answer by $\textsc{Extend}(D_{1,1}) = \textsc{Extend}(\{\langle 4, 3\rangle\})$ with $a_2 = 5$. We get $D_{2,2} = \{\langle 5, 6\rangle\}$. Its meaning is that one CIS of $A_{1..2}$ and $B_{1..6}$ is of length 2 with ending number $a_2 = b_6 = 5$. $\langle 5, 6\rangle$ cannot be extended any more, since the next element 1 (a_3) is less than 5, not increasing. Thus, $D_{3,3} = \emptyset$. Now, the first round stops. In the first round, the possible CIS length 2 is obtained.

In the second round ($r = 2$), we start at the second element a_2 of A and find the first match $a_2 = b_2$ in B. Thus, we can extend from $\langle 0, 0\rangle$ to $\langle 5, 2\rangle$, i.e. $\textsc{Extend}(D_{1,0}) = \{\langle 5, 2\rangle\}$. We combine $D_{1,1}$ and $\langle 5, 2\rangle$ to form $D_{2,1} = \{\langle 4, 3\rangle, \langle 5, 2\rangle\}$. Both $\langle 4, 3\rangle$ and $\langle 5, 2\rangle$ are preserved since they have the same CIS length but do not dominate each other. Next, we cannot extend $D_{2,1}$ with a_3. We get $D_{3,2} = D_{2,2} = \{\langle 5, 6\rangle\}$. $\langle 5, 6\rangle$ cannot be extended, since element 4 (a_4) is less than 5 of $\langle 5, 6\rangle$. Thus, $D_{4,3} = \emptyset$. Now, the second round stops, and the possible CIS answer is of length 2.

In the third round ($r = 3$), we can extend from $\langle 0, 0\rangle$ to $\langle 1, 1\rangle$ with $a_3 = 1$. It can be seen that $\langle 1, 1\rangle$ has more possibility to find a longer CIS than $\langle 4, 3\rangle$ and $\langle 5, 2\rangle$. That is, $\langle 1, 1\rangle$ dominates $\langle 4, 3\rangle$ and $\langle 5, 2\rangle$. Then, $\langle 4, 3\rangle$ and $\langle 5, 2\rangle$ are eliminated. Thus, $D_{3,1} = \textsc{Dominate}(D_{2,1}, \textsc{Extend}(D_{2,0})) = \{\langle 1, 1\rangle\}$. Here, $\textsc{Dominate}$ is defined later. Note that $|\textsc{Extend}(D_{i-1,s-1})| \leq 1$, which will be proved later. Next, we do the extension furthermore. We get $D_{4,2} = \textsc{Dominate}(D_{3,2}, \textsc{Extend}(D_{3,1})) = \{\langle 4, 3\rangle\}$ and $D_{5,3} = \textsc{Dominate}(D_{4,3}, \textsc{Extend}(D_{4,2})) = \{\langle 8, 7\rangle\}$. Finally, we obtain the solution of the LCIS length, which is 3.

The formal definition of the function $\textsc{Dominate}$ is given as follows.

Definition 5. Given a dominating set $D_{i-1,s}$ and a 2-tuple W=$\langle k, j\rangle$, the function $\textsc{Dominate}(D_{i-1,s}, W)$ returns the dominating set which consists of all 2-tuples in $D_{i-1,s}$ and $\langle k, j\rangle$ after eliminating dominated 2-tuples.

3.2. The analyses of our algorithm

Suppose $\langle k_h, j_h \rangle \in D_{i-1,s-1}$. The extension of $\langle k_h, j_h \rangle$ with a_i requires that $k_h < a_i$. And it is to find the smallest j such $j_h < j \le n$ and $a_i = b_j = k$. Then $\langle k, j \rangle$ is added into $D_{i,s}$ if $\langle k, j \rangle$ is not dominated.

Lemma 1. *When $D_{i-1,s-1}$ is extended with a_i, at most one element needs to be extended.*

Proof. Let $D_{i-1,s-1} = \{\langle k_1, j_1 \rangle, \langle k_2, j_2 \rangle, \ldots, \langle k_q, j_q \rangle\}$, where $q = |D_{i-1,s-1}|$, $k_1 < k_2 < \cdots < k_q$ and $j_1 > j_2 > \cdots > j_q$. Suppose that k_h is the largest $k_{h'}$ such that $k_{h'} < a_i$, $1 \le h' \le q$. And suppose that $\langle k_h, j_h \rangle$ is extended to a 2-tuple $\langle k, j \rangle \in D_{i,s}$ with a_i. Thus, $k_h < k = a_i$. For another $\langle k_{h''}, j_{h''} \rangle \in D_{i-1,s-1}$, where $k_{h''} < k_h$, we have $j_{h''} > j_h$ due to the domination property. Thus, $\langle k, j \rangle$ should dominate the extension of $\langle k_{h''}, j_{h''} \rangle$ with a_i. Therefore, at most one element of $D_{i-1,s-1}$ needs to be extended. □

Definition 6. Given a dominating set $D_{i,s}$ and an element x, $Predecessor$ $(D_{i,s}, x)$ is to find the 2-tuple with the largest k_h (first entry of the 2-tuple) in $D_{i,s}$ that is less than x; $Successor$ $(D_{i,s}, x)$ is to find the 2-tuple with the smallest k_h in $D_{i,s}$ that is greater than x.

Let $\langle k_h, j_h \rangle \in D_{i-1,s-1}$. There exists one CIS length $s-1$ between $A_{1..i-1}$ and $B_{1..j_h}$, where $b_{j_h} = k_h$. We check one more element a_i in A. According to Lemma 1, EXTEND($D_{i-1,s-1}$) needs to extend at most one element $\langle k_h, j_h \rangle \in D_{i-1,s-1}$ to another 2-tuple $\langle k, j \rangle \in D_{i,s}$, where $k_h < k = a_i$, $j_h < j \le n$, and j is the smallest for $b_j = k = a_i$. We have to check only the largest k_h, which is smaller than a_i. So, $Predecessor(D_{i-1,s-1}, a_i)$ is used to accomplish this goal. After this extension, we get one CIS length s between $A_{1..i}$ and $B_{1..j}$ if there exists such j after j_h. In other words, we extend one element a_i in A to match one common element in B, and thus the CIS length $s-1$ is extended to s (or more, improved latter). If $\langle k, j \rangle$ does not exist, the extension result is defined to be null.

Definition 7. When a 2-tuple is inserted into $D_{i,s}$ without being dominated, we say that it is a legal insertion.

Lemma 2. *When a 2-tuple $\langle k, j \rangle$ is to be inserted into $D_{i,s}$, at most one 2-tuple in $D_{i,s}$ needs to be checked whether the insertion is legal or not.*

Proof. Let $D_{i,s} = \{\langle k_1, j_1 \rangle, \langle k_2, j_2 \rangle, \ldots, \langle k_q, j_q \rangle\}$, where $q = |D_{i,s}|$ and $k_1 < k_2 < \cdots < k_q$. Suppose that $k_h < k < k_{h+1}$, where $0 \le h \le q$ and assume that $k_0 = -\infty$ and $k_{q+1} = \infty$. Because of the domination property in $D_{i,s}$, we have $j_1 > j_2 > \cdots > j_h$. k_h is the largest number less than k. If $\langle k_h, j_h \rangle$ does not dominate $\langle k, j \rangle$ ($j_h > j$), then any other element of $D_{i,s}$ cannot dominate $\langle k, j \rangle$. At this moment, $\langle k, j \rangle$ can be inserted into $D_{i,s}$ legally. On the other hand, if $\langle k_h, j_h \rangle$ dominates $\langle k, j \rangle$, then $\langle k, j \rangle$ cannot be inserted. Therefore, at most one 2-tuple in $D_{i,s}$ needs to be checked when $\langle k, j \rangle$ is to be inserted into $D_{i,s}$. □

DOMINATE involves two operations, insertion and deletion. Suppose that an element $\langle k, j \rangle$ is to be inserted into $D_{i,s}$, which is extended from $D_{i-1,s-1}$. According to Lemma 2, we can use $Predecessor(D_{i,s}, a_i)$ to find the largest element k_h that is less than k, and then check whether the insertion of $\langle k, j \rangle$ is legal or not. If the insertion is legal, then we perform the deletion operation to remove those elements in $D_{i,s}$ dominated by $\langle k, j \rangle$. The deletion can be done by using $Successor(D_{i,s}, k)$ to find the smallest element that is greater than a_i, say it is $\langle k_1, j_1 \rangle$. If $\langle k, j \rangle$ dominates $\langle k_1, j_1 \rangle$, then $\langle k_1, j_1 \rangle$ is deleted. Repeat the deletion procedure until no dominated element remains in $D_{i,s}$.

For example, suppose $D = \{\langle 1,5 \rangle, \langle 3,4 \rangle, \langle 5,2 \rangle\}$. Suppose that $\langle 2,3 \rangle$ is to be inserted into D. We get DOMINATE($D, \{\langle 2,3 \rangle\}$) $= \{\langle 1,5 \rangle, \langle 2,3 \rangle, \langle 5,2 \rangle\}$, since $\langle 3,4 \rangle$ is dominated by $\langle 2,3 \rangle$.

With the EXTEND and DOMINATE functions, the LCIS problem can be solved by calculating $D_{i,s}$ sequentially. The pseudo code of our LCIS algorithm is shown in Algorithm 1.

The main loop (containing lines 4 through 15) constructs $D_{i,s}$ with the row-major scheme, as shown in Table 3. In round r (in line 4), it scans A sequentially starting from a_r to construct sets $D_{r,1}$, $D_{r+1,2}$, $D_{r+2,3}$, $\cdots$, $D_{i,s}$ until the set cannot be constructed. Then, the maximum CIS length can be obtained in this round. In lines 5 and 6, we terminate the LCIS algorithm when $L > m - r$, because the maximally possible CIS length of round r is $m - r + 1$. Thus, in round $r + 1$, the maximum length will be no more than the current L. Line 9 extends $D_{i-1,s-1}$ with one element a_i in A, which determined by $Predecessor(D_{i-1,s-1}, a_i)$. At most one 2-tuple is produced in $W = \{\langle k, j \rangle\}$. Lines 10 and 11 check W and $D_{i-1,s}$, so we may terminate this round if both W and $D_{i-1,s}$ are empty. Line 12 invokes $Predecessor(D_{i-1,s}, W)$ for insertion and $Successor(D_{i-1,s}, W)$ for removing the dominated 2-tuples in $D_{i-1,s}$. Lines 14 and 15 store the maximum value of s (the maximum CIS length in this round) to L.

Lemma 3. *Each 2-tuple $\langle k, j \rangle$ belongs to at most one $D_{i,s}$ in each round r, where $r \le i \le L + r - 1$ and $1 \le s \le L$.*

Algorithm 1 Computing the LCIS length.

```
Input: Sequences $A = \langle a_1, a_2, a_3, \ldots, a_m \rangle$ and $B = \langle b_1, b_2, b_3, \ldots, b_n \rangle$, $m \le n$
Output: Length of $LCIS(A, B)$
 1: Construct $nextB$                                         ▷ next symbol in $B$
 2: Set $D_{i,0} = \{\langle 0, 0 \rangle\}$ for $0 \le i \le m$
 3: $L \leftarrow 0$
 4: for $r = 1 \to m$ do                                       ▷ round $r$
 5:     if $L > m - r$ then                                    ▷ round $r > m - L$
 6:         break
 7:     for $s = 1 \to m - r + 1$ do
 8:         $i \leftarrow r + s - 1$
 9:         $W \leftarrow$ EXTEND($D_{i-1,s-1}$)                ▷ decided by $Predecessor(D_{i-1,s-1}, a_i)$
10:         if $(D_{i-1,s} \cup W) = \emptyset$ then
11:             break                                          ▷ go to 14
12:         $D_{i,s} \leftarrow$ DOMINATE($D_{i-1,s}, W$)       ▷ decided by $Predecessor(D_{i-1,s}, W)$
13:                                ▷ for insertion and $Successor(D_{i-1,s}, W)$ for deletion
14:     if $s > L$ then
15:         $L \leftarrow s$
16: return $L$
```

Proof. Suppose that $\langle k_1, j_1 \rangle \in D_{i,s}$ and $\langle k_2, j_2 \rangle \in D_{i+1,s+1}$, where $k_1 = k_2$ and $j_1 = j_2$. If $\langle k_2, j_2 \rangle$ is extended from $\langle x, y \rangle \in D_{i,s}$, where $x < k_2$ and $y < j_2$. It means that $\langle k_1, j_1 \rangle$ is dominated by $\langle x, y \rangle$. So, $\langle k_1, j_1 \rangle$ cannot be in $D_{i,s}$. Therefore, the lemma holds. □

The following is obviously true due to the definition of a dominating set.

Property 1. *The number of 2-tuples $\langle k_h, j_h \rangle$ in a dominating set $D_{i,s}$ where $1 \le s \le L$ is at most n.*

Lemma 4. *The number of 2-tuples $\langle k, j \rangle$ in a round is at most n.*

Proof. Suppose there exist $n+1$ 2-tuples in a round which are $\langle k_1, j_1 \rangle, \langle k_2, j_2 \rangle, \cdots, \langle k_n, j_n \rangle, \langle k_{n+1}, j_{n+1} \rangle$. Since j_h represents a position index of B, $1 \le h \le n+1$, $|B| = n$, there must exist $j_x = j_{x'}$, for $1 \le x, x' \le n+1$. Then, $\langle k_x, j_x \rangle = \langle k_{x'}, j_{x'} \rangle$. These two identical 2-tuples cannot be in the same dominating set, and they cannot belong to different dominating sets in a round by Lemma 3. So the number of 2-tuples in a round is at most n. □

With Lemma 4, we obtain the following space complexity, since the memory space can be reused in different rounds.

Theorem 1. *Algorithm 1 solves the LCIS problem with $O(n)$ space.*

The EXTEND function is shown in Function 1. $nextB$ is used to find the index of the first match element a_i in B after $\langle k_h, j_h \rangle$. If we extend successfully $\langle k_h, j_h \rangle$ to another 2-tuple $\langle k, j \rangle$, then it is true that $j \le n$. According to Lemma 3, we use Boolean array *AlreadyExist* to check whether j already exists or not. *AlreadyExist* is initialized to false for each j when we start a new round. The Boolean array *AlreadyExist* can reduce the execution time practically when the algorithm is implemented.

Function 1 The extension of $D_{i-1,s-1}$.

```
Input: $D_{i-1,s-1}$, a global Boolean array $AlreadyExist$
Output: $W = \{\langle k, j \rangle\}$
 1: function EXTEND($D_{i-1,s-1}$)
 2:     $\langle k_h, j_h \rangle \leftarrow Predecessor(D_{i-1,s-1}, a_i)$
 3:     if $\langle k_h, j_h \rangle \ne null$ then
 4:         $k \leftarrow a_i$
 5:         $j \leftarrow next_B(a_i, j_h)$                     ▷ next match $a_i$ in $B$ after position $j_h$
 6:     if $j = null$ or $AlreadyExist[j] = true$ then
 7:         Return
 8:     Update $AlreadyExist[j] \leftarrow true$
 9:     return $W = \{\langle k, j \rangle\}$
10: end function
```

The DOMINATE function is shown in Function 2. DOMINATE inserts a 2-tuple and deletes the dominated 2-tuples. We invoke *Predecessor* to check if $\langle k, j \rangle$ is dominated. *Successor* is used to check if other 2-tuples in $D_{i,s}$ are dominated by $\langle k, j \rangle$.

Function 2 The domination of $D_{i-1,s}$ and W.

```
Input: $D_{i-1,s}$ and $W = \{\langle k, j\rangle\}$
Output: $D_{i,s}$
1: function DOMINATE($D_{i-1,s}$, $W$)
2:     $D_{i,s} \leftarrow D_{i-1,s}$
3:     $\langle k_h, j_h\rangle \leftarrow Predecessor(D_{i,s}, k)$
4:     if $\langle k_h, j_h\rangle = null$ or $j_h > j$ then
5:         Insert $\langle k, j\rangle$ into $D_{i,s}$
6:         while $\langle k_x, j_x\rangle \leftarrow Successor(D_{i,s}, k) \neq null$ do
7:             if $j_x > j$ then
8:                 Delete $\langle k_x, j_x\rangle$ from $D_{i,s}$
9:             else
10:                break
11:        end while
12:    return $D_{i,s}$
13: end function
```

Theorem 2. *Algorithm 1 solves the LCIS problem in $O(nextB + L(m-L)n)$ time, where $O(nextB)$ is the time for building the $nextB$ table.*

Proof. By Lemmas 1 and 2, Algorithm 1 is correct. The preprocessing stage (line 1) constructs *nextB*. The outer loop in lines 4 through 15 is executed exactly $(m-L)$ times (The outer loop breaks when $r > m-L$). The inner loop in lines 7 through 13 is executed at most L times. Therefore, there are at most $L(m-L)$ extensions. It is obvious that there are at most $L(m-L)$ insertions by Lemma 2. And then, the number of deletions is also at most $L(m-L)$. According to Lemma 3, each round has at most n distinct 2-tuples. So, each insertion or each deletion requires $O(n)$ time. Thus, the time complexity is $O(nextB + L(m-L)n)$. □

It is worthwhile to mention that the EXTEND and DOMINATE functions utilize a linear scan scheme to accomplish the job. Thus, each EXTEND or DOMINATE requires linear time. Furthermore, it can be reduced if we implement these two functions with the efficient data structure, the van Emde Boas (vEB) tree [21].

For $D_{1,s}, D_{2,s}, D_{3,s} \ldots, D_{m-L,s}$, we use a *van Emde Boas tree* (vEB tree) T_s to store the elements in $D_{i,s}$, $1 \le i \le m-L$. In other words, in the progress of the algorithm, the answers of the same s-length of CIS are stored in the same tree T_s. Thus, we build L vEB trees.

Each operation of insertion, deletion, predecessor or successor in a vEB tree requires $O(\log\log|\Sigma|)$ time [21], since the number of distinct keys in $D_{i,s}$ is at most $|\Sigma|$.

When we extend $\langle k_h, j_h\rangle$ to another 2-tuple $\langle k, j\rangle$, to find out $\langle k, j\rangle$ efficiently, we build the $next_B$ table with vEB trees. In the table, $nextB(\beta, j_h) = j$ denotes the next position of character β in B after position j_h. We build $|\Sigma|$ vEB trees, each for one symbol. Each symbol with its occurrence positions in B is stored in a vEB tree. Therefore, we use $Successor(nextB, j_h)$ to search the j with $O(\log\log|\Sigma|)$ time. So, it takes $O(L(m-L)\log\log|\Sigma|)$ time totally in searching *nextB*.

The preprocessing stage takes $O(n\log\log|\Sigma|)$ time to construct $next_B$. Each EXTEND or DOMINATE can be done in $O(\log\log|\Sigma|)$ time. Thus, the time complexity is reduced to $O((n + L(m-L))\log\log|\Sigma|)$. We summarize the result in the following theorem.

Theorem 3. *Algorithm 1 solves the LCIS problem in $O((n + L(m-L))\log\log|\Sigma|)$ time.*

4. Experimental results

In this section, we perform some simulation experiments on pseudorandom sequences with various lengths to illustrate the time efficiency of our LCIS and LCWIS algorithms. We compare the execution time of our algorithm and some previously published algorithms. Each experiment is repeated 100 times with different input sequences to get the average execution time. These algorithms are implemented by Code::Blocks 16.01 C++ software, and they are tested on a computer with 64-bit Windows 7 OS, CPU clock rate of 3.20GHZ (Intel(R) Core(TM) i5-4570 CPU) and 16 GB of RAM.

4.1. The longest common increasing subsequence

The experiments concentrate on the relationship of execution time and LCIS lengths. Both algorithms of Sakai [13] and Cai et al. [17] are space-efficient with $O(n)$ space. However, the performances of these two algorithms are close to the DP-based algorithms of Yang et al. [12]. Therefore, we do not test the algorithms of Sakai [13] and Cai et al. [17] in these experiments.

In our experiments, we compare the performance of the following LCIS algorithms:

- **Yang**: The DP algorithm of Yang et al. [12],

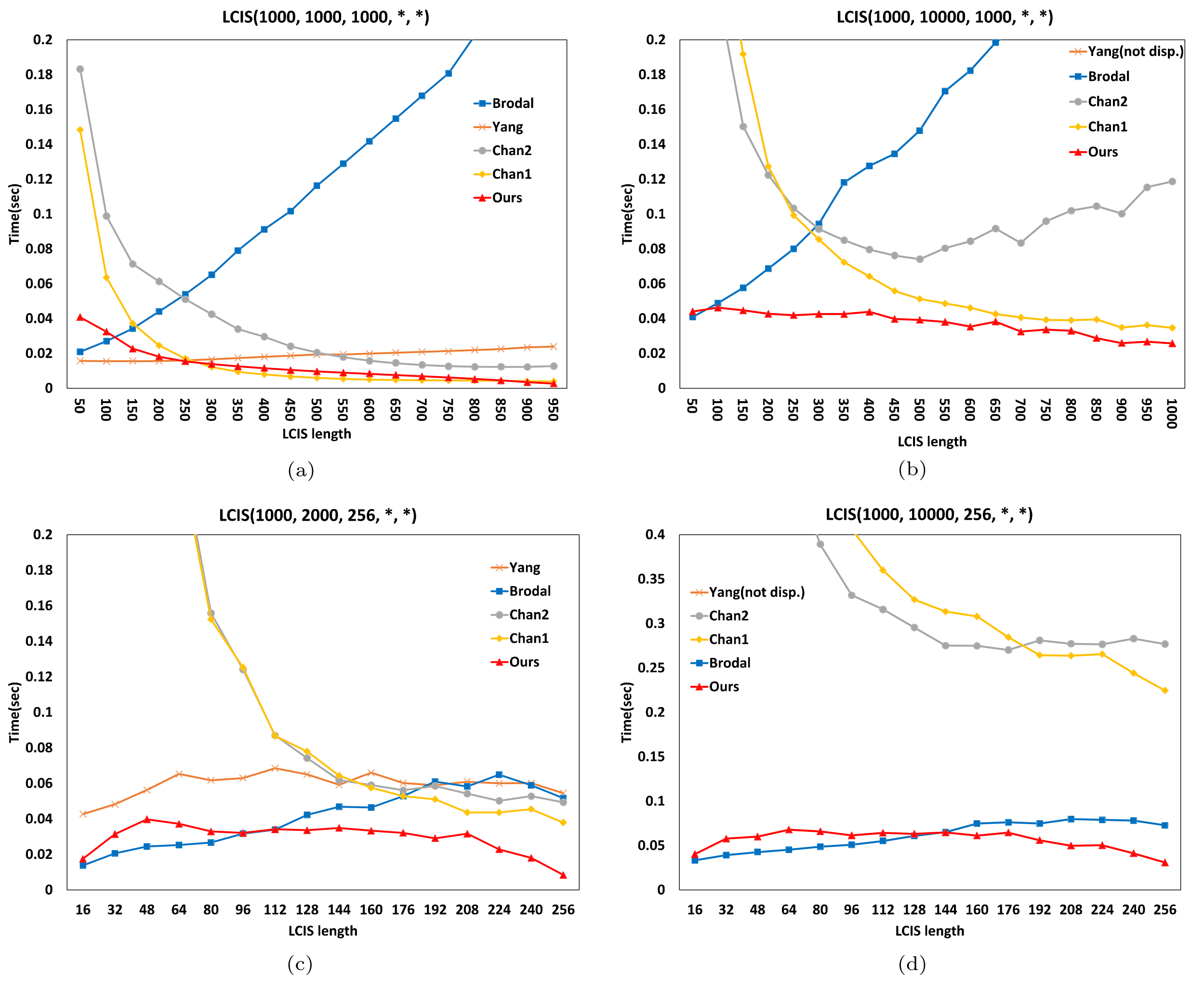

Fig. 1. The execution time for the LCIS algorithms with various parameter values. Here, “not disp.” means that the execution time goes beyond the scope of the figure.

- **Brodal**: The algorithm proposed by Brodal [14],
- **Chan1**: The sparse DP algorithm of Chan et al. by using binary search [16],
- **Chan2**: The sparse DP algorithm of Chan et al. by using sequential search [16],
- **Ours**: Our algorithm presented in this paper.

We generate various pseudorandom datasets to test the relationship between the LCIS length and execution time in our experiments. These pseudorandom datasets are generated or mutated by the pseudorandom function of C++, but we manually change some sequences. If the datasets are generated fully randomly, it is almost impossible to generate the sequences with extremely large LCIS lengths. We omit the details of the data generation procedure here.

Fig. 1 shows the execution time of these algorithms. We use a 5-tuple $(|A|, |B|, |\Sigma|, LCIS\_length, algo)$ to represent the parameters in each performance chart. For example, in Fig. 1a, $(1000, 1000, 1000, *, *)$ means that $|A| = m = 1000$, $|B| = n = 1000$, alphabet size $|\Sigma| = 1000$, the first “$*$” is a wildcard representing all possible LCIS lengths and the second “$*$” is a wildcard representing all possible algorithms. Figs. 1a and 1b illustrate the execution time of various LCIS algorithms for $|B| = 1000$ and $|B| = 10000$, respectively, with $|\Sigma| = 1000$. Figs. 1c and 1d illustrate the execution time of various LCIS algorithms for $|B| = 2000$ and $|B| = 10000$, respectively, with $|\Sigma| = 256$.

It is obviously that DP algorithm of Yang et al. takes more time than other algorithms. The sparse DP algorithm of Chan et al. has two versions, both of them depend on the number of match pairs. When the number of match pairs is small, the sparse DP algorithm of Chan et al. is efficient. We perform the algorithm of Brodal et al. with that a segment tree is utilized to execute the range minimum query. The algorithm of Brodal et al. is fast when the LCIS length is small. Our method is also fast when L is close to m.

Table 4
An example for the LCWIS algorithm modified from Yang et al., where $A = \langle 2, 2, 1, 1, 1\rangle$ and $B = \langle 1, 2, 2, 1, 1\rangle$. Here, the second value in each cell denotes the position index of B.

	1	2	2	1	1
2		$L_{1,2}[1] = 2, \underline{2}$	$L_{1,3}[1] = 2, \underline{2}$	$L_{1,4}[1] = 2, \underline{2}$	$L_{1,5}[1] = 2, \underline{2}$
2		$L_{2,2}[1] = 2, 2$	$L_{2,3}[1] = 2, 2$ $L_{2,3}[2] = 2, \underline{3}$	$L_{2,4}[1] = 2, 2$ $L_{2,4}[2] = 2, \underline{3}$	$L_{2,5}[1] = 2, 2$ $L_{2,5}[2] = 2, \underline{3}$
1	$L_{3,1}[1] = 1, \underline{1}$	$L_{3,2}[1] = 1, \underline{1}$	$L_{3,3}[1] = 1, \underline{1}$ $L_{3,3}[2] = 2, 3$	$L_{3,4}[1] = 1, \underline{1}$ $L_{3,4}[2] = 2, 3$	$L_{3,5}[1] = 1, \underline{1}$ $L_{3,5}[2] = 2, 3$
1	$L_{4,1}[1] = 1, 1$	$L_{4,2}[1] = 1, 1$	$L_{4,3}[1] = 1, 1$ $L_{4,3}[2] = 2, 3$	$L_{4,4}[1] = 1, 1$ $L_{4,4}[2] = 1, \underline{4}$	$L_{4,5}[1] = 1, 1$ $L_{4,5}[2] = 1, \underline{4}$
1	$L_{5,1}[1] = 1, 1$	$L_{5,2}[1] = 1, 1$	$L_{5,3}[1] = 1, 1$ $L_{5,3}[2] = 2, 3$	$L_{5,4}[1] = 1, 1$ $L_{5,4}[2] = 1, 4$	$L_{5,5}[1] = 1, 1$ $L_{5,5}[2] = 1, 4$ $L_{5,5}[3] = 1, \underline{5}$

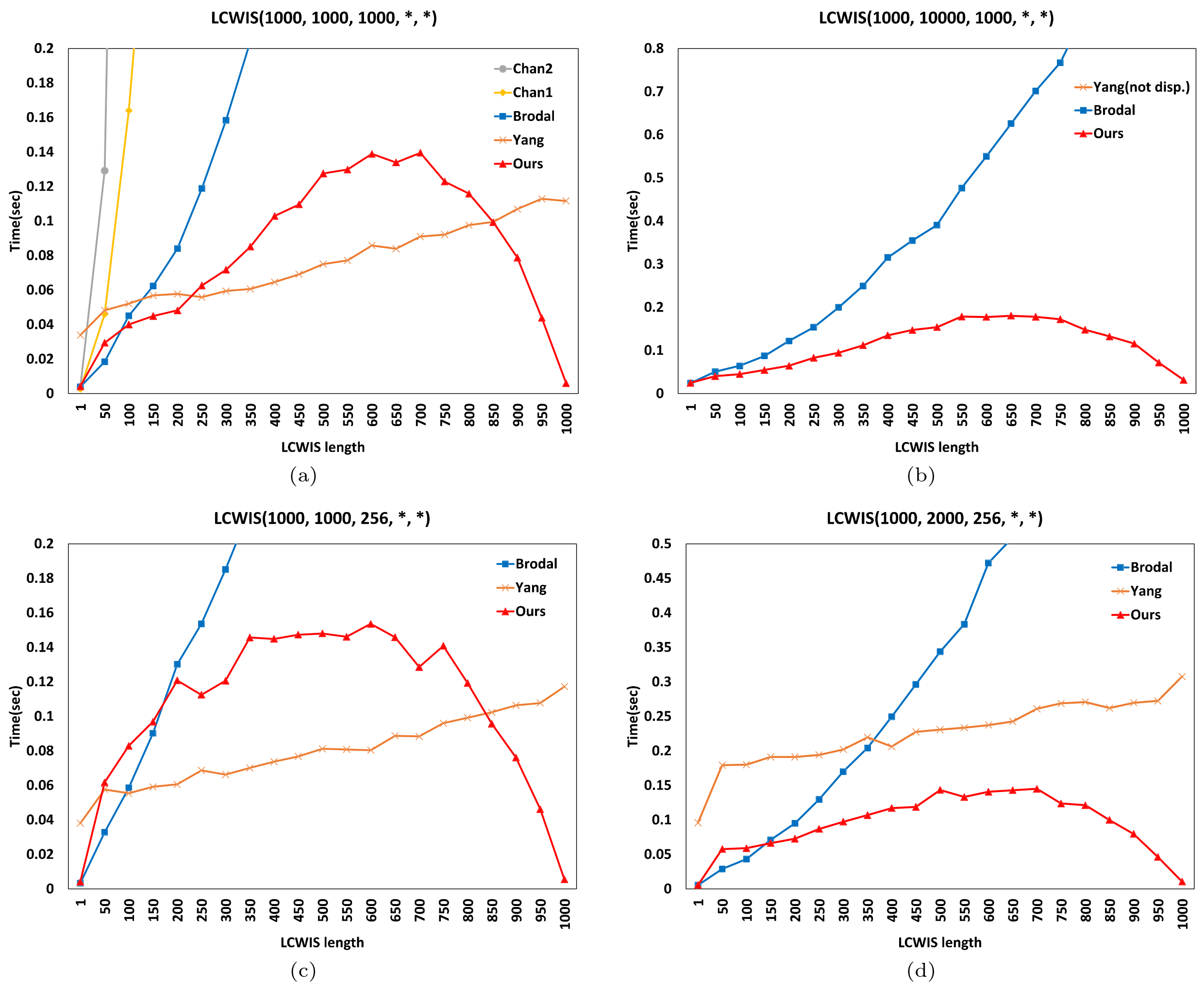

Fig. 2. The execution time for the LCWIS algorithms with various parameter values.

4.2. The longest common weakly increasing subsequence

We modify the LCIS algorithms to solve the LCWIS problem. For our algorithm and an algorithm of Brodal et al., it can be easily modified by replacing $<$ with $\leq$. For the DP algorithm of Yang et al., we can record additionally each match position of B, as shown with the underline in the example of Table 4. With the match positions, the length extension can be done correctly. For the sparse DP algorithms of Chan et al., we have to modify the order of match pairs from the descending order of j and then i, to become the ascending order of i and then j. It avoids to be dominated directly when we insert it into a tree. We test all experiments with different small alphabet sizes. The sparse DP algorithm of Chan et al. takes more time, since there are lots of match pairs.

Fig. 2a shows the execution time of the above LCWIS algorithms with $|A| = m = 1000$, $|B| = n = 1000$, $|\Sigma| = 1000$. There are many match pairs in our generated LCWIS datasets, so the sparse DP algorithms of Chan et al. take too much time. In the following figures, their algorithms are not shown. Figs. 2a and 2b illustrate the execution time for $|B| = 1000$ and $|B| = 10000$ with $|\Sigma| = 1000$. Figs. 2c and 2d illustrate the execution time for $|B| = 1000$ and $|B| = 2000$ with $|\Sigma| = 256$.

5. Conclusion

We propose a diagonal-based algorithm for solving the LCIS problem in $O((n + L(m - L)) \log\log|\Sigma|)$ time and $O(n)$ space, where m, n and L denote the lengths of sequences A, B and the LCIS length, respectively. We perform simulations of our algorithm and previously published LCIS algorithms on pseudorandom datasets. The experimental results illustrate that our algorithm is faster than other algorithms in most cases, especially when L is close to m

In the future, we also hope that the time complexity can be further improved theoretically. The set find-union has been invoked in the incremental suffix maximum query (a special case of range maximum query), and it successfully reduces the time complexity theoretically [22]. In our algorithm for the LCIS problem, the predecessor query and the range maximum query are equivalent. And, one predecessor query in the vEB tree can be accomplished in $O(\log\log|\Sigma|)$ time. Applying the set find-union to this special range maximum query may be a direction for improving the time complexity of an LCIS algorithm.

The bit-parallel algorithm is utilized efficiently in other LCS variant problems, such as MLCS and CLCS [23,24]. In the future, it may speed up our algorithm or previously published LCIS and LCWIS algorithms with the bit-parallel algorithms.

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] H.Y. Ann, C.B. Yang, C.T. Tseng, C.Y. Hor, A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings, Inf. Process. Lett. 108 (2008) 360–364.
[2] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.
[3] J. Hunt, T. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (1977) 350–353.
[4] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inform. 18 (1982) 171–179.
[5] R. Wagner, M. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.
[6] 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.
[7] C. Schensted, Longest increasing and decreasing subsequences, Can. J. Math. 13 (1961) 179–191.
[8] D.E. Knuth, The Art of Computer Programming: Sorting and Searching, 2nd edition, Addison-Wesley, 1973.
[9] S. Bespamyatnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inf. Process. Lett. 76 (2000) 7–11.
[10] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucleic Acids Res. 27 (1999) 2369–2376.
[11] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, S. Salzberg, Versatile and open software for comparing large genomes, Genome Biol. 5 (2) (2004).
[12] 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.
[13] Y. Sakai, A linear space algorithm for computing a longest common increasing subsequence, Inf. Process. Lett. 99 (2006) 203–207.
[14] G.S. Brodal, K. Kaligosi, I. Katriel, M. Kutz, Faster algorithms for computing longest common increasing subsequences, in: Proc. of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM), Barcelona, Spain, 2006, pp. 330–341.
[15] M. Kutz, G.S. Brodal, K. Kaligosi, I. Katriel, Faster algorithms for computing longest common increasing subsequences, J. Discret. Algorithms (2011) 314–325.
[16] W.-T. Chan, Y. Zhang, S.P.Y. Fung, D. Ye, H. Zhu, Efficient algorithms for finding a longest common increasing subsequence, J. Comb. Optim. 13 (3) (2007) 277–288, https://doi.org/10.1007/s10878-006-9031-7.
[17] D. Cai, D. Zhu, L. Wang, X. Wang, A simple linear space algorithm for computing a longest common increasing subsequence, IAENG Int. J. Comput. Sci. 45 (3) (2018).
[18] L. Duraj, A linear algorithm for 3-letter longest common weakly increasing subsequence, Inf. Process. Lett. 113 (2013) 94–99.
[19] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[20] M. Crochemore, E. Porat, Fast computation of a longest increasing subsequence and application, Inf. Comput. 208 (2010) 1054–1059.
[21] P.V.E. Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (3) (1977) 80–82.
[22] Y.H. Peng, C.B. Yang, Finding the gapped longest common subsequence by incremental suffix maximum queries, Inf. Comput. 237 (2014) 95–100.
[23] A. Danek, S. Deorowicz, Bit-parallel algorithm for the block variant of the merged longest common subsequence problem, Adv. Intell. Syst. Comput. 242 (2014) 173–181.
[24] S. Deorowicz, Bit-parallel algorithm for the constrained longest common subsequence problem, Fundam. Inform. 99 (2010) 409–433.