Theoretical Computer Science 1044 (2025) 115269

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# The merged longest common increasing subsequence problem ☆

Chien-Ting Lee [a], Chang-Biau Yang [a,*], 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

Section Editor: Pinyan Lu

Handling Editor: Blerina Sinaimeri

*Keywords:*
Longest increasing subsequence
Merged longest common subsequence
Merged longest common increasing subsequence
Dynamic programming
Diagonal

ABSTRACT

The merged longest common increasing subsequence (MLCIS) problem represents a generalized variant by combining the merged longest common subsequence (merged LCS, MLCS) problem and the longest increasing subsequence (LIS) problem. Given a pair of numeric sequences $A$ and $B$ along with a target numeric sequence $T$, the MLCIS problem aims to identify the longest common subsequence that is increasing in both the merged sequence $E(A, B)$ and the target sequence $T$. Here, $E(A, B)$ represents a new sequence constructed from arbitrarily merging $A$ and $B$ while maintaining their original orders. In this paper, we propose two algorithms for solving the MLCIS problem: dynamic programming and diagonal. The dynamic programming algorithm has a time complexity of $O(mnr)$, where $m$, $n$ and $r$ denote the lengths of sequences $A$, $B$ and $T$, respectively. The time complexity of the diagonal algorithm is $O((L+1)(r-L+1)(m+n))$, where $L$ denotes the length of the MLCIS answer. In general, as the experimental results show, the diagonal algorithm is more efficient than the dynamic programming algorithm in practice. Furthermore, the diagonal algorithm is very efficient when $L$ is either very small or $L$ is close to $r$, which coincides with the theoretical time complexity.

## 1. Introduction

The concept of *longest common subsequence* (LCS) [7,22] can be used to measure the similarity of two given sequences in real life situations, such as bioinformatics, speech recognition and plagiarism detection [14]. Recognizing the diverse applications of LCS, researchers have explored several variants, including *multiple longest common subsequence* [15], *longest common increasing subsequence* (LCIS) [23], and *merged longest common subsequence* (MLCS) [8] problems.

Given two sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, the LCS indicates the common subsequence of both $A$ and $B$ with the maximum length. Without loss of generality, it is assumed that $m \leq n$. In 1974, Wagner and Fischer [22] proposed a dynamic programming (DP) algorithm for solving the LCS problem in $O(mn)$ time. Then, in 1977, Hunt and Szymanski [9] proposed an $O((R+m)\log m)$-time algorithm based on match pairs of $A$ and $B$, where $R$ denotes the number of match pairs between $A$ and $B$, defined as $R = |\{(i,j) \mid a_i = b_j, 1 \leq i \leq |A|, 1 \leq j \leq |B|\}|$. Nakatsu et al. [16] introduced a diagonal method in $O((L+1)(m-L))$ time, where $L$ denotes the length of the LCS answer. The algorithm has better efficiency when $L$ is small or close to $m$.

In 2005, Yang et al. [23] first introduced the *longest common increasing subsequence* (LCIS) problem. The LCIS problem attempts to find the common subsequence of $A$ and $B$, that is increasing and has the maximum length. In 2008, Huang et al. [8] defined

☆ Two preliminary versions of this paper were presented at the 16th Asian Conference on Intelligent Information and Database Systems (ACIIDS 2024), April 2024, Ras Al Khaimah, UAE; and the 41st Workshop on Combinatorial Mathematics and Computation Theory, May, 2024, Taipei, Taiwan.

* Corresponding author.
*E-mail address:* cbyang@cse.nsysu.edu.tw (C.-B. Yang).

https://doi.org/10.1016/j.tcs.2025.115269
Received 18 July 2024; Received in revised form 2 February 2025; Accepted 20 April 2025

Available online 24 April 2025

**Table 1**
The time and space complexities of the previous LCIS algorithms for two input sequences $A$ and $B$. Here, $m = |A|$, $n = |B|$; $L$: length of the LCIS answer; $R$: number of match pairs between $A$ and $B$; $\Sigma$: alphabet set; DP: dynamic programming; BST: balanced search tree [14].

| Year | Author(s) | Time | Space | Notes |
|---|---|---|---|---|
| 2005 | Yang et al. [23] | $O(mn)$ | $O(mn)$ | DP |
| 2006 | Sakai [19] | $O(mn)$ | $O(m+n)$ | Divide-and-conquer |
| 2006 | Brodal et al. [2,11] | $O((n+mL)\log\log\lvert\Sigma\rvert + \text{SORT}(n))$ | $O(n)$ | Divide-and-conquer, bounded heap, van Emde Boas tree |
| 2007 | Chan et al. [4] | $O(\min\{R\log L, nL+R\} \cdot \log\log n + \text{SORT}(n))$ | $O(R)$ | Match pair, binary search, van Emde Boas tree |
| 2018 | Cai et al. [3] | $O(mn)$ | $O(m+n)$ | Divide-and-conquer |
| 2020 | Lo et al. [14] | $O((n+L(m-L))\log\log\lvert\Sigma\rvert)$ | $O(n)$ | Diagonal, van Emde Boas tree |
| 2020 | Duraj [6] | $O(n^2(\log\log n)^2/\log^{1/6} n)$ | $O(n^2/\log n)$ | Significant pairs, van Emde Boas queue |
| 2020 | Agrawal & Gawrychowski [1] | $O(n^2\log\log n/\sqrt{\log n})$ | $O(n^2/\log n)$ | DP, BST, fragment, symbol frequency |

**Table 2**
The time and space complexities of the previous MLCS algorithms for two input sequences $A$ and $B$ as well as the target sequence $T$. Here, $m = |A|$, $n = |B|$, $r = |T|$; $L$: length of the MLCS answer; $w$: word size of the computer; $P$: number of match pairs between $A$ and $T$; $Q$: number of match pairs between $B$ and $T$; $\Sigma$: alphabet set; DP: dynamic programming [21].

| Year | Author(s) | Time | Space | Notes |
|---|---|---|---|---|
| 2008 | Huang et al. [8] | $O(mnr)$ | $O(mn)$ | DP |
| 2010 | Peng et al. [17] | $O(Lmr)$ | $O(n+Lm)$ | DP |
| 2013 | Deorowicz and Danek [5] | $O(\lceil r/w\rceil mn\log w)$ | $\Theta(\lceil r/w\rceil mn)$ | Bit-parallel |
| 2014 | Rahman and Rahman [18] | $O((Pn+Qm)\log\log m)$ | $\Theta(\max\{nr, m\})$ | Bounded heap |
| 2018 | Tseng et al. [21] | $O(n\lvert\Sigma\rvert + (r-L+1)Lm)$ | $O(n\lvert\Sigma\rvert + Lm)$ | Diagonal |

the *merged longest common subsequence* (MLCS) problem, aiming to find the LCS between a merged sequence $E(A, B)$ and the target sequence $T$. Here, $E(A, B)$ denotes a new sequence obtained from arbitrarily merging two subsequences in $A$ and $B$ while retaining their original orders in $A$ and $B$ individually. The previous algorithms for solving the LCIS and the MLCS problems are summarized in Tables 1 and 2, respectively.

In molecular biology, certain blocks within a sequence may retain their significance. Through alternative splicing, multiple proteins can be produced from a single gene by recombining blocks of introns and exons [20]. The selected blocks maintain the syntenic order in the original sequence, corresponding to the increasing sequence of block labels. To detect doubly conserved synteny (DCS) blocks between organisms [10], the merged LCS algorithms can be applied to biological sequences of amino acids or peptides. For longer genome sequences, analyzing related blocks between organisms, rather than individual matched characters, offers more meaningful insights. Hence, an MLCS algorithm tailored to increasing sequences is both necessary and effective, as illustrated by DCS blocks in whole-genome duplication studies [10,17].

In this paper, we first define the *merged longest common increasing subsequence* (MLCIS) problem, and then propose two algorithms to address it. Given two sequences $A$ and $B$, along with a target sequence $T$, the MLCIS problem is to identify a common subsequence of $E(A, B)$ and $T$ with the maximum length such that the answer is increasing. Here, $E(A, B)$ represents an arbitrary sequence obtained by merging $A$ and $B$ while retaining the original orders in $A$ and $B$ individually. For example, consider $A = \langle \underline{2}, \underline{5}, \underline{4}, \underline{8}\rangle$, $B = \langle 7, 4, 1, 8, 7\rangle$ and $T = \langle 2, 7, 4, 5, 9, 7, 8\rangle$. $E(A, B)$ could be $\langle \underline{2}, \underline{5}, \underline{4}, \underline{8}, 7, 4, 1, 8, 7\rangle$, $\langle \underline{2}, 7, 4, \underline{5}, \underline{4}, 1, 8, 7, \underline{8}\rangle$, or others. The MLCIS answer is $\langle \underline{2}, 4, \underline{5}, 7, \underline{8}\rangle$ with length 5, where $\langle a_1 = 2, b_2 = 4, a_2 = 5, b_5 = 7, a_4 = 8\rangle$.

The MLCIS problem is then solved in $O(mnr)$ time using the dynamic programming (DP) approach, and in $O((L+1)(r-L+1)(m+n))$ time using the diagonal approach, where $m$, $n$, $r$, and $L$ denote the lengths of sequences $A$, $B$, $T$, and the MLCIS answer, respectively. The experimental results show that in the DP approach, the larger the $|A| \times |B|$ is, the more time it takes. The diagonal method has better efficiency when either the MLCIS length $L$ is very small or $L$ is close to $r$.

This paper is arranged as follows. We introduce the background knowledge of the MLCIS problem in Section 2. Section 3 presents the dynamic programming algorithm for solving the MLCIS problem. Section 4 gives the diagonal MLCIS algorithm. In Section 5, we compare the performance of the DP and the diagonal algorithms. At the end, conclusions and future work are given in Section 6.

## 2. Preliminaries

A subsequence is obtained by removing zero or some elements from the original sequence, where the relative order of the original sequence is preserved in the subsequence. Given two sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, the longest common subsequence (LCS) refers to a common subsequence of both $A$ and $B$ with the maximum length. Without loss of generality, it is assumed that $m \le n$. As an illustration, $A = \langle \mathtt{a}, \mathtt{t}, \mathtt{g}, \mathtt{g}, \mathtt{t}, \mathtt{c} \rangle$ and $B = \langle \mathtt{c}, \mathtt{a}, \mathtt{g}, \mathtt{a}, \mathtt{c}, \mathtt{t} \rangle$, the LCS length of $A$ and $B$ is 3, which may be $\langle \mathtt{a}, \mathtt{g}, \mathtt{t} \rangle$ or $\langle \mathtt{a}, \mathtt{g}, \mathtt{c} \rangle$. Note that there may be multiple LCSs of the same length.

Suppose two numeric sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$ are given, where $m \le n$. A subsequence $S = \langle s_1, s_2, \cdots, s_l \rangle$ is a *common increasing subsequence* (CIS) of $A$ and $B$, where $a_{i_1} = b_{j_1} = s_1$, $a_{i_2} = b_{j_2} = s_2$, $\cdots$, $a_{i_l} = b_{j_l} = s_l$, $i_1 < i_2 < \cdots < i_l$, $j_1 < j_2 < \cdots < j_l$, and $s_1 < s_2 < \cdots < s_l$. The *longest common increasing subsequence* (LCIS) of $A$ and $B$ is any common increasing subsequence with the maximum length. For example, given $A = \langle 3, 1, 9, 6, 3, 4 \rangle$, $B = \langle 6, 1, 9, 3, 6, 7, 2, 4 \rangle$, the LCIS answer is $\langle 1, 3, 4 \rangle$ with length 3.

Yang et al. [23] proposed a dynamic programming algorithm in both O($mn$) time and space for solving the LCIS problem. Inspired by the diagonal concept of LCS algorithm proposed by Nakatsu et al. [16], Lo et al. [14] proposed a diagonal method for solving the LCIS problem, with time complexity $O((n + L(m - L)) \log \log |\Sigma|)$. That is, the algorithm is highly efficient when the two sequences are either highly dissimilar or extremely similar.

Given two sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, along with a target sequence $T = \langle t_1, t_2, \cdots, t_r \rangle$, the *merged LCS* (MLCS) problem [8] is represented as MLCS($A, B, T$). The MLCS problem is to find the LCS of $E(A, B)$ and $T$, where, $E(A, B)$ denotes a sequence built by merging $A$ and $B$ while retaining their original orders in $A$ and $B$ individually. Let $E(A, B) = \langle e_1, e_2, \cdots, e_{m+n} \rangle$. We define $C = \langle c_{l_1}, c_{l_2}, \cdots, c_{l_m} \rangle = A$ as a subsequence of $E(A, B)$, where $1 \le l_1 < l_2 < \cdots < l_m \le m+n$. The remaining sequence, $E(A, B) \setminus C$, is precisely equal to $B$. Without loss of generality, we assume that $m \le n$. For example, suppose that $A = \langle \mathtt{g}, \mathtt{a}, \mathtt{t} \rangle$, $B = \langle \mathtt{a}, \mathtt{t}, \mathtt{g}, \mathtt{a} \rangle$ and $T = \langle \mathtt{a}, \mathtt{g}, \mathtt{c}, \mathtt{a}, \mathtt{t}, \mathtt{a} \rangle$. The MLCS answer is $\langle \mathtt{a}, \mathtt{g}, \mathtt{a}, \mathtt{t}, \mathtt{a} \rangle$ with length 5.

The *merged longest common increasing subsequence* (MLCIS) problem is a generalized variant of the LCIS problem combined with the MLCS problem.

**Definition 1** *(MLCIS problem).* Given a pair of numeric sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, along with a target sequence $T = \langle t_1, t_2, \cdots, t_r \rangle$, the *merged longest common increasing subsequence* (MLCIS) problem is to find the longest common increasing subsequence of $E(A, B)$ and $T$, where $E(A, B)$ is a sequence obtained by merging $A$ and $B$ arbitrarily while retaining their original orders in $A$ and $B$ individually.

In this paper, we use MLCIS($A, B, T$) to represent the sequence answer of the MLCIS problem. Note that $E(A, B)$ is not a function; it merely describes an arbitrarily merging operation. Thus, there are many possible results for obtaining $E(A, B)$. For example, consider two numeric sequences $A = \langle \underline{2}, \underline{5}, \underline{4}, \underline{8} \rangle$, $B = \langle 7, 4, 1, 8, 7 \rangle$ and $T = \langle 2, 7, 4, 5, 9, 7, 8 \rangle$. $E(A, B)$ could be $\langle \underline{2}, \underline{5}, \underline{4}, \underline{8}, 7, 4, 1, 8, 7 \rangle$, $\langle \underline{2}, 7, 4, \underline{5}, \underline{4}, 1, 8, 7, \underline{8} \rangle$, or others. We get the MLCIS($A, B, T$) = $\langle \underline{2}, 4, \underline{5}, 7, \underline{8} \rangle$, whose length is 5. More precisely, we observe that $\langle a_1 = 2, b_2 = 4, a_2 = 5, b_5 = 7, a_4 = 8 \rangle$ in this answer.

## 3. The dynamic programming algorithm

In the DP algorithm for MLCIS, two tables $\alpha$ and $\beta$ are employed to represent the MLCIS lengths ending at elements of $A$ and $B$, respectively. To establish the boundary conditions, we add dummy elements $a_0$, $b_0$, and $t_0$ in front of sequences $A$, $B$, and $T$, respectively. Here, $a_0 = b_0 = t_0 = \epsilon$.

**Definition 2** *($\alpha(i, j, k) = l$).* The notation $\alpha(i, j, k) = l$ means that the MLCIS length $l$ can be constructed from sequence $A_{0..i}$, $B_{0..j}$ and $T_{0..k}$, where $a_i$ is the ending element of the corresponding MLCIS answer.

**Definition 3** *($\beta(i, j, k) = l$).* The notation $\beta(i, j, k) = l$ means that the MLCIS length $l$ can be constructed from sequence $A_{0..i}$, $B_{0..j}$ and $T_{0..k}$, where $b_j$ is the ending element of the corresponding MLCIS answer.

To calculate the MLCIS length, when $t_k$ in $T$ matches $a_i$ in $A$ ($t_k = a_i$), it refers to the maximum MLCIS length before $a_i$ with an ending value less than $t_k$. Similarly, if $t_k$ in $T$ matches $b_j$ in $B$ ($t_k = b_j$), it refers to the maximum MLCIS length before $b_j$ whose ending value is less than $t_k$.

By Definitions 2 and 3, we get Theorem 1.

**Theorem 1.** *In the $\alpha$ table, $\alpha(i, j-1, k) \le \alpha(i, j, k)$, for $0 \le i \le m$, $1 \le j \le n$ and $0 \le k \le r$. In the $\beta$ table, $\beta(i-1, j, k) \le \beta(i, j, k)$, for $1 \le i \le m$, $0 \le j \le n$ and $0 \le k \le r$.*

**Proof.** $\alpha(i, j-1, k)$ is the maximum MLCIS length built from $A_{0..i}$, $B_{0..j-1}$ and $T_{0..k}$, ending at $a_i$. The sequence merged by $A_{0..i}$ and $B_{0..j-1}$ can be a subsequence of the one merged by $A_{0..i}$ and $B_{0..j}$. Therefore, the length of $\alpha(i, j-1, k)$ that can form an MLCIS ending at $a_i$ is less than or equal to $\alpha(i, j, k)$.

Similarly, the same result can be obtained for $\beta(i-1, j, k) \le \beta(i, j, k)$. □

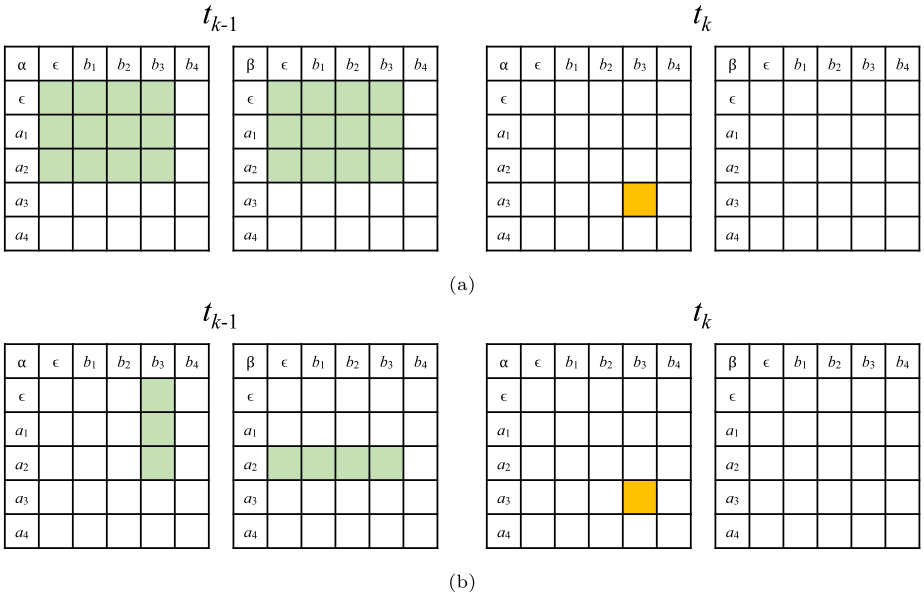



**Fig. 1.** The reference area for $\alpha(3,3,k)$ calculation. Colored (yellow or gray) block in $t_k$: $\alpha(3,3,k)$ for a match of $a_3 = t_k$; colored (green or gray) blocks in $t_{k-1}$: the reference area for $\alpha(3,3,k)$. (a) The original reference area. (b) The shrunk reference area based on Theorem 1. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

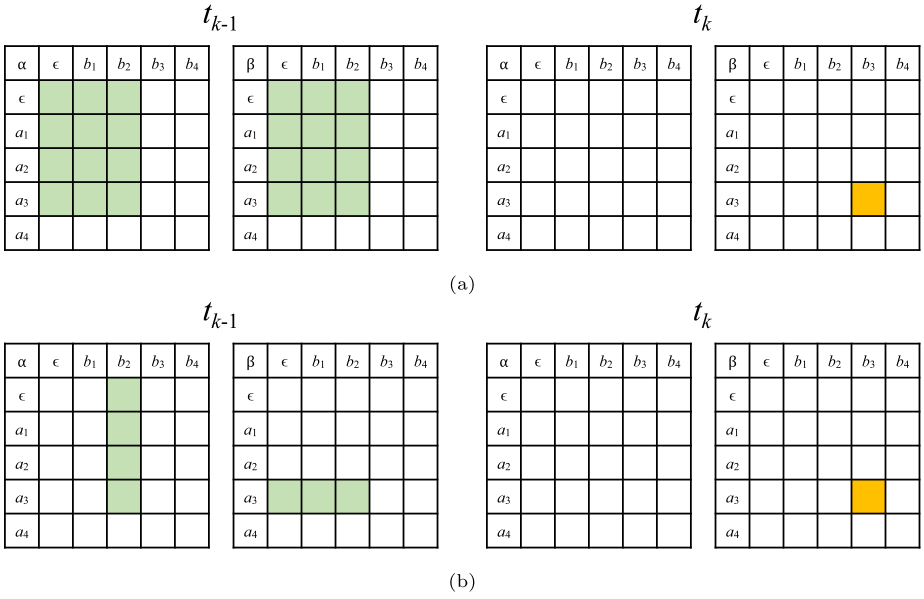



**Fig. 2.** The reference area for $\beta(3,3,k)$ calculation. Colored (yellow or gray) block in $t_k$: $\beta(3,3,k)$ for a match of $b_3 = t_k$; colored (green or gray) blocks in $t_{k-1}$: the reference area for $\beta(3,3,k)$. (a) The original reference area. (b) The shrunk reference area based on Theorem 1.

Based on Theorem 1, when $t_k = a_i$, the calculation of $\alpha(i,j,k)$ requires referencing only the maximum value from $\alpha(0,j,k-1)$ to $\alpha(i-1,j,k-1)$ and the maximum value from $\beta(i-1,0,k-1)$ to $\beta(i-1,j,k-1)$. When $t_k = b_j$, the calculation of $\beta(i,j,k)$ needs only to refer to the maximum from $\alpha(0,j-1,k-1)$ to $\alpha(i,j-1,k-1)$ and the maximum from $\beta(i,0,k-1)$ to $\beta(i,j-1,k-1)$. Figs. 1 and 2 show the examples of the reference areas for $\alpha(i,j,k)$ and $\beta(i,j,k)$, respectively.

When a match occurs at $t_k$ ($a_i = t_k$ or $b_j = t_k$), to avoid duplicate calculation of $\alpha$ and $\beta$, and to gather current maxima of reference areas in $\alpha$ and $\beta$ (the elements in $A$ and $B$ less than $t_k$), we introduce two additional tables, $\alpha^\star$ and $\beta^\star$, defined in Equations (1) to (2), respectively.

$$\alpha^\star(i,j,k-1) = \max\{\alpha(i',j,k-1) | a_{i'} < t_k, 0 \le i' \le i\}. \tag{1}$$

$$\beta^\star(i,j,k-1) = \max\{\beta(i,j',k-1) | b_{j'} < t_k, 0 \le j' \le j\}. \tag{2}$$

Using $\alpha^\star$ and $\beta^\star$, the DP method for computing the MLCIS length is given in Equations (3) through (6).

$$\alpha^\star(i,j,k-1)=\begin{cases}0 & \text{if } i=0;\\ \alpha^\star(i-1,j,k-1) & \text{if } a_i \geq t_k;\\ \max\{\alpha^\star(i-1,j,k-1),\alpha(i,j,k-1)\} & \text{if } a_i < t_k.\end{cases} \tag{3}$$

$$\beta^\star(i,j,k-1)=\begin{cases}0 & \text{if } j=0;\\ \beta^\star(i,j-1,k-1) & \text{if } b_j \geq t_k;\\ \max\{\beta^\star(i,j-1,k-1),\beta(i,j,k-1)\} & \text{if } b_j < t_k.\end{cases} \tag{4}$$

$$\alpha(i,j,k)=\begin{cases}0 & \text{if } i=0 \text{ or } k=0;\\ \max\{\alpha^\star(i-1,j,k-1),\beta^\star(i-1,j,k-1)\}+1 & \text{if } a_i=t_k;\\ \alpha(i,j,k-1) & \text{if } a_i\neq t_k.\end{cases} \tag{5}$$

$$\beta(i,j,k)=\begin{cases}0 & \text{if } j=0 \text{ or } k=0;\\ \max\{\alpha^\star(i,j-1,k-1),\beta^\star(i,j-1,k-1)\}+1 & \text{if } b_j=t_k;\\ \beta(i,j,k-1) & \text{if } b_j\neq t_k.\end{cases} \tag{6}$$

Finally, the MLCIS length can be obtained by Equation (7).

$$|\text{MLCIS}(A,B,T)|=\max\{\alpha(i,n,r),\beta(m,j,r)|1\leq i\leq m,1\leq j\leq n\}. \tag{7}$$

Fig. 3 illustrates an example to explain the proposed DP algorithm.

If we need to output the MLCIS$(A,B,T)$ content, a backtracking technique can be applied to these 3-dimensional tables by recording the incremental paths of the MLCIS solution.

**Theorem 2.** *The MLCIS problem can be solved by Equations (3) to (7) in O($mnr$) time and O($mn$) space.*

**Proof.** The computation of each $\alpha(i,j,k)$, $\beta(i,j,k)$, $\alpha^\star(i,j,k)$ and $\beta^\star(i,j,k)$ requires constant time according to Equations (3) to (6). Tables $\alpha(\cdot)$, $\beta(\cdot)$, $\alpha^\star(\cdot)$ and $\beta^\star(\cdot)$ are all 3-dimensional tables with index $(i,j,k)$ for $0\leq i\leq m$, $0\leq j\leq n$ and $0\leq k\leq r$. So the time complexity is O($mnr$).

Even though $\alpha(\cdot)$, $\beta(\cdot)$, $\alpha^\star(\cdot)$ and $\beta^\star(\cdot)$ are 3-dimensional tables, they can be reduced to 2-dimensional tables. This is due to the reuse of each set of indices for different $k$ independently. Thus, each of them needs O($mn$) space. In other words, the MLCIS problem can be solved in O($mn$) space. □

## 4. The diagonal algorithm

### 4.1. The main concept

Our diagonal algorithm for the MLCIS problem [12] is inspired by the algorithm proposed by Nakatsu et al. [16], which was originally designed for solving the LCS problem. The diagonal method is more efficient in solving the LCS problem when the input sequences are highly similar.

**Definition 4** *(3-tuple $(i,j,v)$).* Given two numeric sequences $A=\langle a_1,a_2,\cdots,a_m\rangle$ and $B=\langle b_1,b_2,\cdots,b_n\rangle$, along with a target sequence $T=\langle t_1,t_2,\cdots,t_r\rangle$, we use a 3-tuple $(i,j,v)$ to represent a potential MLCIS solution of $A_{1..i}$ and $B_{1..j}$ ending at value $v$.

**Definition 5** *(Domination).* Given two 3-tuples $(i_1,j_1,v_1)\neq(i_2,j_2,v_2)$, if $i_1\leq i_2$, $j_1\leq j_2$ and $v_1\leq v_2$, it is said that $(i_1,j_1,v_1)$ *dominates* $(i_2,j_2,v_2)$.

**Definition 6** *(Dominating set).* A set is a *dominating set* if no element in it dominates any other element.

**Definition 7** *($(i,j,v)\in S_{k,q}$).* Let $S_{k,q}$ be a dominating set, where $k,q\geq 0$. Then, each 3-tuple $(i,j,v)\in S_{k,q}$ represents $|\text{MLCIS}(A_{1..i},B_{1..j},T_{1..k})|=q$, where $v=\max\{a_i,b_j\}$ and the MLCIS answer ends at $v$. In addition, $S_{k,q}=\{(0,0,0)\}$ for $q=0$ as initialization.

To solve the MLCIS problem, we need two auxiliary functions, EXTEND$(\cdot)$ and DOMINATE$(\cdot)$, where EXTEND$(\cdot)$ performs the solution extension for each 3-tuple in a set, and DOMINATE$(\cdot)$ maintains a dominating set by removing dominated 3-tuples.

**Definition 8** *(EXTEND$(\cdot)$).* Given $t_k$, we can extend each 3-tuple $(i_1,j_1,v_1)\in S_{k-1,q-1}$ to $(i_2,j_1,t_k)$ if $a_{i_2}=t_k$, $i_1<i_2$ and $v_1<t_k$, and to $(i_1,j_2,t_k)$ if $b_{j_2}=t_k$, $j_1<j_2$ and $v_1<t_k$. If no such $a_{i_2}$ or $b_{j_2}$ exists, the extension result is set to null. Such feasible $(i_2,j_1,t_k)$ and $(i_1,j_2,t_k)$ are possible elements of $S_{k,q}$. This extension is denoted by EXTEND$(S_{k-1,q-1},t_k)$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) $t_0 = \epsilon$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) $t_1 = 2$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 2 | 0 | 0 | 0 | 2 |
| 5 | 0 | 2 | 0 | 0 | 0 | 2 |
| 4 | 0 | 2 | 0 | 0 | 0 | 2 |
| 8 | 0 | 2 | 0 | 0 | 0 | 2 |

(c) $t_2 = 7$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 2 | 2 | 0 | 0 | 2 |
| 5 | 0 | 2 | 2 | 0 | 0 | 2 |
| 4 | 0 | 2 | 2 | 0 | 0 | 2 |
| 8 | 0 | 2 | 2 | 0 | 0 | 2 |

(d) $t_3 = 4$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 2 | 3 | 3 | 3 | 3 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 2 | 2 | 0 | 0 | 2 |
| 5 | 0 | 2 | 2 | 0 | 0 | 2 |
| 4 | 0 | 2 | 2 | 0 | 0 | 2 |
| 8 | 0 | 2 | 2 | 0 | 0 | 2 |

(e) $t_4 = 5$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 2 | 3 | 3 | 3 | 3 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 2 | 2 | 0 | 0 | 2 |
| 5 | 0 | 2 | 2 | 0 | 0 | 2 |
| 4 | 0 | 2 | 2 | 0 | 0 | 2 |
| 8 | 0 | 2 | 2 | 0 | 0 | 2 |

(f) $t_5 = 9$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 2 | 3 | 3 | 3 | 3 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 1 | 0 | 0 | 2 |
| 2 | 0 | 2 | 2 | 0 | 0 | 3 |
| 5 | 0 | 3 | 2 | 0 | 0 | 4 |
| 4 | 0 | 3 | 2 | 0 | 0 | 4 |
| 8 | 0 | 3 | 2 | 0 | 0 | 4 |

(g) $t_6 = 7$.

| α | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 2 | 2 | 3 | 3 | 3 | 3 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |
| 8 | 3 | 4 | 4 | 4 | 4 | 5 |

| β | ϵ | 7 | 4 | 1 | 8 | 7 |
|---|---|---|---|---|---|---|
| ϵ | 0 | 1 | 1 | 0 | 2 | 2 |
| 2 | 0 | 2 | 2 | 0 | 3 | 3 |
| 5 | 0 | 3 | 2 | 0 | 4 | 4 |
| 4 | 0 | 3 | 2 | 0 | 4 | 4 |
| 8 | 0 | 3 | 2 | 0 | 4 | 4 |

(h) $t_7 = 8$.

**Fig. 3.** An example of the DP algorithm, where $A = \langle 2,5,4,8\rangle$, $B = \langle 7,4,1,8,7\rangle$ and $T = \langle 2,7,4,5,9,7,8\rangle$. Here, $\epsilon$ denotes a dummy element. The answer MLCIS($A, B, T$) $= \langle 2,4,5,7,8\rangle$ with length 5.

**Definition 9** *(DOMINATE(·)).* Given a set $S$, the domination function, denoted by DOMINATE($S$), is to remove all dominated 3-tuples in $S$ such that $S$ becomes a dominating set.

With EXTEND(·) and DOMINATE(·), Theorem 3 is our main operation for generating $S_{k,q}$.

**Theorem 3.** $S_{k,q} = DOMINATE(S_{k-1,q} \cup EXTEND(S_{k-1,q-1}, t_k))$.

**Proof.** The theorem is proved by mathematical induction. When $k = 1$ and $q = 1$, we have $S_{k-1,q} = S_{0,1} = \emptyset$ obviously and $S_{k-1,q-1} = S_{0,0} = \{(0,0,0)\}$. Then, $S_{k,q} = S_{1,1}$ should only consist of elements extended from $(0,0,0)$ in $S_{0,0}$ by $t_1$. So, $S_{k,q} =$ DOMINATE($S_{k-1,q} \cup$ EXTEND($S_{k-1,q-1}, t_k$)) holds when $k = 1$ and $q = 1$.

By introduction hypothesis, assume that when $1 \le q \le k$, the theorem holds for $S_{k-1,q}$ and $S_{k-1,q-1}$. To establish $S_{k,q}$, its elements are obtained through two possible ways.

(1) The extension of the elements in $S_{k-1,q-1}$ is made with $t_k$. If the extension is successful, the solution length will increase to $q$. So the resulting extensions become feasible elements in $S_{k,q}$.

(2) If $(i,j,v) \in S_{k-1,q}$, it means $|\text{MLCIS}(A_{1..i}, B_{1..j}, T_{1..k-1})| = q$. So $(i,j,v)$ is a feasible element in $S_{k,q}$, and its MLCIS length is $q$.

Therefore, $S_{k-1,q} \cup$ EXTEND($S_{k-1,q-1}, t_k$) contains all feasible elements. Finally, DOMINATE($S_{k-1,q} \cup$ EXTEND($S_{k-1,q-1}, t_k$)) removes the dominated elements. Thus, the theorem holds. □

We use the example shown in Table 3 to explain the details, where $A = \langle 2,5,4,8\rangle$, $B = \langle 7,4,1,8,7\rangle$, and $T = \langle 2,7,4,5,9,7,8\rangle$. In Round 1, we initialize $S_{0,0} = \{(0,0,0)\}$. By Theorem 3, $S_{1,1} =$ DOMINATE($S_{0,1} \cup$ EXTEND($S_{0,0}, t_1$)). By Definition 7, we have $S_{0,1} = \emptyset$.

**Table 3**
The construction of $S_{k,q}$ in the diagonal MLCIS algorithm with $A = \langle 2,5,4,8\rangle$, $B = \langle 7,4,1,8,7\rangle$, and $T = \langle 2,7,4,5,9,7,8\rangle$.

| Round $f$ \ $q$ (length) | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 1 | $S_{0,0}$ | $S_{1,1}$ | $S_{2,2}$ | $S_{3,3}$ | | |
| | $(0,0,0)$ | $(1,0,2)$ | $(1,1,7)$ | | | |
| 2 | $S_{1,0}$ | $S_{2,1}$ | $S_{3,2}$ | $S_{4,3}$ | $S_{5,4}$ | |
| | $(0,0,0)$ | $(0,1,7)$ | $(1,1,7)$ | $(2,2,5)$ | | |
| | | $(1,0,2)$ | $(1,2,4)$ | | | |
| | | | $(3,0,4)$ | | | |
| 3 | $S_{2,0}$ | $S_{3,1}$ | $S_{4,2}$ | $S_{5,3}$ | $S_{6,4}$ | $S_{7,5}$ |
| | $(0,0,0)$ | $(0,1,7)$ | $(1,1,7)$ | $(2,2,5)$ | $(2,5,7)$ | $(4,5,8)$ |
| | | $(0,2,4)$ | $(1,2,4)$ | | | |
| | | $(1,0,2)$ | $(2,0,5)$ | | | |
| | | ~~$(3,0,4)$~~ | ~~$(2,2,5)$~~ | | | |
| | | | $(3,0,4)$ | | | |

For $\textsc{Extend}(S_{0,0}, t_1)$, we can find $a_1 = t_1 = 2$, and there is no $b_j$ equal to $t_1 = 2$ in $B$, resulting in $(0,0,0)$ being only extended to $(1,0,2)$. Thus, $S_{1,1} = \{(1,0,2)\}$. Next, $S_{2,2} = \textsc{Dominate}(S_{1,2} \cup \textsc{Extend}(S_{1,1}, t_2))$, where $S_{1,2} = \emptyset$. We find that $b_1 = t_2 = 7$, resulting in $S_{2,2} = \{(1,1,7)\}$. Note that, $(1,1,7)$ cannot be further extended, thus $S_{3,3} = \emptyset$.

In Round 2, we repeat the above steps to get the results in Table 3.

In Round 3, $S_{3,1} = \textsc{Dominate}(S_{2,1} \cup \textsc{Extend}(S_{2,0}, t_3))$. We can find $a_3 = b_2 = t_3 = 4$, so $S_{2,0} = \{(0,0,0)\}$ can be extended to $(3,0,4)$ and $(0,2,4)$. Thus, $S_{2,1} \cup \textsc{Extend}(S_{2,0}, 4) = \{(0,1,7),(0,2,4),(1,0,2),(3,0,4)\}$. Here, $(3,0,4)$ is dominated by $(1,0,2)$. Finally, $S_{3,1} = \{(0,1,7),(0,2,4),(1,0,2)\}$. Similarly, $S_{4,2}$, $S_{5,3}$, $S_{6,4}$ and $S_{7,5}$ are calculated sequentially. In the end, according to $S_{7,5} = \{(4,5,8)\}$, we obtain the MLCIS length to be 5.

### 4.2. The detailed algorithm and data structures

Our diagonal method for solving the MLCIS problem is formally presented in Algorithm 1.

We use two doubly linked lists to maintain the 3-tuples in $S_{k,q}$. These two lists consist of the same 3-tuples, but their orders are different. One list, named the i-list, is sorted increasingly by the $i$ values, and the other, named the j-list, is sorted increasingly by the $j$ values. In terms of data structures, each node has four pointers, where next_i and prev_i are used for the i-list, and next_j and prev_j are used for the j-list.

The kernel of Algorithm 1 is $S_{k,q} \leftarrow \textsc{Dominate}(S_{k-1,q} \cup \textsc{Extend}(S_{k-1,q-1}, t_k))$. Function 1 presents the detail to obtain the i-list of $\textsc{Extend}(S_{k-1,q-1}, t_k)$ (extension of $S_{k-1,q-1}$ with $t_k$). Function 2 implements $S_{k,q} = \textsc{Dominate}(S_{k-1,q} \cup \textsc{Extend}(S_{k-1,q-1}, t_k))$. It employs a linear merge scheme to efficiently merge the i-list of $\textsc{Extend}(S_{k-1,q-1}, t_k)$ with $S_{k-1,q}$, and then remove the dominated elements.

**Algorithm 1** The diagonal algorithm for computing the MLCIS length.

**Input:** Two numeric sequences $A = \langle a_1, a_2, \cdots, a_m\rangle$ and $B = \langle b_1, b_2, \cdots, b_n\rangle$, along with a target sequence $T = \langle t_1, t_2, \cdots, t_r\rangle$.
**Output:** Length of MLCIS$(A, B, T)$

```
1: Construct the arrays of next_A and next_B                ▷ next character position
2: L ← 0                                                    ▷ answer length
3: for i = 1 → r do
4:     S_{i-1,0} ← {(0,0,0)}
5:     for q = 1 → r − i + 1 do
6:         k ← i + q − 1
7:         S_{k,q} ← Dominate(S_{k-1,q} ∪ Extend(S_{k-1,q-1}, t_k))
8:         if S_{k,q} = ∅ then
9:             break
10:        if q > L then
11:            L ← q
12:    if i ≥ r − L then
13:        break
14: return L
```

We explain the details of the domination operation $\textsc{Dominate}(\cdot)$ with the example shown in Fig. 4. Suppose that we have $S_{k-1,q} = \{(2,6,9),(4,8,6),(6,2,9),(8,4,6)\}$, and we have got $\textsc{Extend}(S_{k-1,q-1}, t_k = 7) = \{(0,7,7),(1,4,7),(4,1,7),(7,0,7)\}$, denoted by the i-list $H$. The i-list of $S_{k-1,q}$, denoted by $S'$, is used for checking the insertion possibility of elements in $H$. The j-list of $S_{k-1,q}$, denoted by $S''$, is used for checking the domination of these inserted elements.

**Function 1** The extension from $S_{k-1,q-1}$ with $t_k$.

**Input:** $S_{k-1,q-1}$, $t_k$, arrays $\text{next}_A$ and $\text{next}_B$
**Output:** $H \leftarrow \text{EXTEND}(S_{k-1,q-1}, t_k)$ ▷ i-list with increasing i-values

```
Set an array of bucket[0..m] ← ∞   ▷ store j-values, bucket[i] = j means that a 3-tuple (i, j, v) is stored, where v = max{a_i, b_j} and m = |A|
H ← ∅
for each (i, j, v) ∈ S_{k-1,q-1} do
    if v < t_k then                                   ▷ can be extended with t_k
        if (i' ← next_A[t_k][i]) ≤ m then             ▷ next match of t_k in A
            bucket[i'] ← min{bucket[i'], j}
        if (j' ← next_B[t_k][j]) ≤ n then             ▷ next match of t_k in B
            bucket[i] ← min{bucket[i], j'}
j'' ← ∞
for i = 0 → m do                                      ▷ build i-list H with increasing i-values
    if bucket[i] ≤ j'' then                           ▷ also with decreasing j-values
        j'' ← bucket[i]
        Insert (i, j'', t_k) into H                   ▷ only non-dominated 3-tuples are inserted
return H
```

**Function 2** The dominating merge of $S_{k-1,q}$ and $\text{EXTEND}(S_{k-1,q-1}, t_k)$.

**Input:** $S_{k-1,q}$ and $S_{k-1,q-1}$
**Output:** $S_{k,q} \leftarrow \text{DOMINATE}(S_{k-1,q} \cup \text{EXTEND}(S_{k-1,q-1}, t_k))$

```
H ← EXTEND(S_{k-1,q-1}, t_k)                          ▷ H is an i-list
(i', j', v') ← first element of i-list S' of S_{k-1,q}   ▷ start of i-list S'
(i'', j'', v'') ← last element of j-list S'' of S_{k-1,q}   ▷ start of j-list S''
while (i*, j*, v*) ← next non-empty element of H do   ▷ increasing i-values
    do                                                ▷ insertion check
        if (i*, j*, v*) is dominated by (i', j', v') then
            break
        if i' > i* or reach end of i-list S' then     ▷ (i*, j*, v*) is not dominated by (i', j', v')
            Insert (i*, j*, v*) into i-list S'
            break
    while (i', j', v') ← next element of i-list S'    ▷ forward visit
    if (i*, j*, v*) is not inserted into i-list S' then
        break
    do                                                ▷ domination check
        if (i'', j'', v'') is dominated by (i*, j*, v*) then
            Delete (i'', j'', v'') from i-list S' and j-list S''
        if j* > j'' or reach end of j-list S'' then   ▷ (i'', j'', v'') is not dominated by (i*, j*, v*)
            Insert (i*, j*, v*) into j-list S''
            break
    while (i'', j'', v'') ← previous element of j-list S''   ▷ backward visit
return i-list S' and j-list S'' as the content of S_{k,q}
```

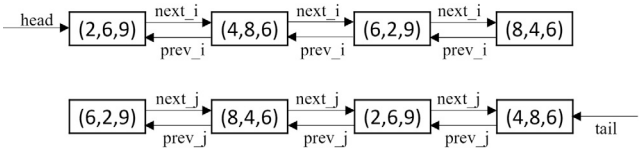
We aim to insert the 3-tuple elements in $H$ sequentially into $S_{k-1,q}$. We first check whether $(0,7,7)$ in $H$ can be inserted to $S'$. The i-value of $(0,7,7)$ is 0. We check only the elements in $S'$ with i-values less than or equal to 0, since only these elements may dominate $(0,7,7)$. Consequently, $(0,7,7)$ can be inserted into $S'$. Next, we check if any element in $S''$ is dominated by $(0,7,7)$. We need to check only the elements with j-values in $S''$ greater than or equal to 7, since only these elements may be dominated by $(0,7,7)$. So we only check $(4,8,6)$, which is not dominated by $(0,7,7)$. Finally, $(0,7,7)$ is inserted into both lists $S'$ and $S''$.

The next element in $H$ is $(1,4,7)$. We check only the elements in $S'$ with i-values less than or equal to 1. Note that we need not check the elements that have been checked for $(0,7,7)$. Because the j-values or v-values of the checked elements are greater than 7, they will be also greater than 4 or greater than 7, respectively. Thus, the checked elements of $(0,7,7)$ cannot dominate $(1,4,7)$. As a result, $(1,4,7)$ can be inserted into $S'$. Next, we check the elements in $S''$ dominated by $(1,4,7)$. Similarly, we need not check the elements that have been checked for $(0,7,7)$. For example, we need not check $(4,8,6)$, since $(4,8,6)$ is not dominated by $(0,7,7)$, resulting in that $(4,8,6)$ is not dominated by $(1,4,7)$ due to the v-value. In a word, we check the elements in $S''$ with j-values for $7 > j \geq 4$. In $S''$, $(2,6,9)$ is dominated and $(8,4,6)$ is not. Finally, $(1,4,7)$ is inserted into both lists $S'$ and $S''$, while $(2,6,9)$ is removed from both lists $S'$ and $S''$.

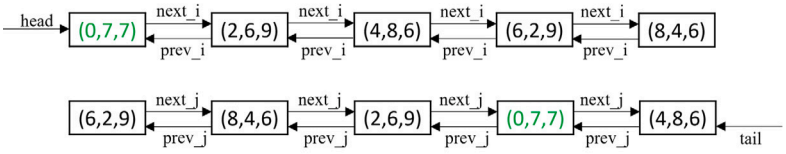
We can perform the same operations for the next elements $(4,1,7)$ and $(7,0,7)$ in $H$ similarly.

In brief, for each new element of $H$, insertion and domination can be performed gradually on the increasing i-list and decreasing j-list. Thus, the dominating merge of $H$ and $S_{k-1,q}$ is handled with a linear merge scheme, and $S_{k,q}$ is set to the merged result. It is clear that the linear merge scheme can be done in linear time.

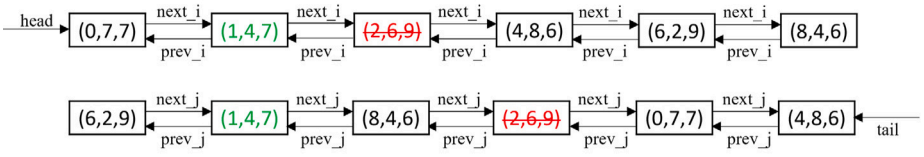
**Theorem 4.** *Function 1 obtains the i-list of the dominating set $H = \text{EXTEND}(S_{k-1,q-1}, t_k)$ with a linear bucket array. In $H$, for any two elements $(i_1, j_1, v_1)$ and $(i_2, j_2, v_2)$, it holds that $i_1 \neq i_2$, $j_1 \neq j_2$ and $v_1 = v_2 = t_k$.*



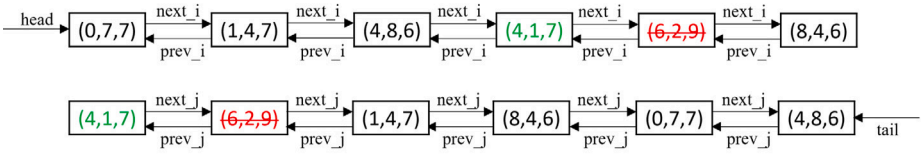

(a) Initialization, where $S_{k-1,q}$ is represented by the two doubly linked lists.



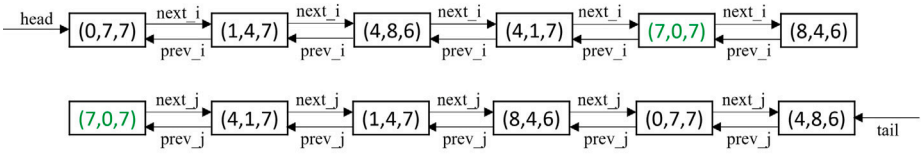

(b) Insertion of $(0, 7, 7)$.




(c) Insertion of $(1, 4, 7)$.




(d) Insertion of $(4, 1, 7)$.




(e) Insertion of $(7, 0, 7)$.

**Fig. 4.** An example of DOMINATE(·), where the results of EXTEND($S_{k-1,q-1}, t_k = 7$) = $\{(0,7,7),(1,4,7),(4,1,7),(7,0,7)\}$ are inserted into $S_{k-1,q} = \{(2,6,9),(4,8,6),(6,2,9),(8,4,6)\}$. Green or gray: a newly inserted element; strikethrough: a deleted element.

**Proof.** By Theorem 3, $S_{k-1,q-1}$ is a dominating set. By Definition 8, for obtaining $H = \text{EXTEND}(S_{k-1,q-1}, t_k)$, each element $(i, j, v)$ in $S_{k-1,q-1}$ can be extended to at most two new elements $(i', j, t_k)$ and $(i, j', t_k)$ by possibly matching $t_k$ with $a_{i'}$ of $A$ and $b_{j'}$ of $B$, respectively. Lines 3 to 8 of Function 1 use a linear array to store $(i', j, t_k)$, $(i, j', t_k)$ temporarily, where bucket$[i'] = j$ indicates $a_{i'} = t_k$, and bucket$[i] = j'$ indicates $b_{j'} = t_k$.

In the linear array, bucket$[i']$ stores the smallest j-value, and bucket$[i]$ also stores the smallest j-value, as done in Lines 6 and 8. If there exist $i_1 \neq i_2$, but bucket$[i_1]$ = bucket$[i_2] = j$, then only the dominating element is retained in $H$, as implemented in Lines 10 to 13. Moreover, all elements are stored in $H$ with increasing i-values. Thus, the theorem holds. □

**Theorem 5.** *Function 2 maintains the i-list and the j-list of dominating set* $S_{k,q} = \text{DOMINATE}(S_{k-1,q} \cup \text{EXTEND}(S_{k-1,q-1}, t_k))$ *by the linear merge scheme.*

**Proof.** Let $H = \text{EXTEND}(S_{k-1,q-1}, t_k)$. By Theorem 4, suppose there are two elements $(i_1^\star, j_1^\star, v_1^\star)$ and $(i_2^\star, j_2^\star, v_2^\star)$ in $H$, where $i_1^\star < i_2^\star$ and $j_1^\star > j_2^\star$, and $v_1^\star = v_2^\star = t_k$. Now, consider $(i_1, j_1, v_1)$ in the i-list of $S_{k-1,q}$, where $i_1 \leq i_1^\star$. If $(i_1, j_1, v_1)$ does not dominate $(i_1^\star, j_1^\star, v_1^\star)$, then $j_1 > j_1^\star$ or $v_1 > v_1^\star$. Consequently, it follows that $j_1 > j_2^\star$ or $v_1 > v_2^\star = v_1^\star$, implying that $(i_1, j_1, v_1)$ does not dominate $(i_2^\star, j_2^\star, v_2^\star)$ either. This implementation is done in Lines 5 to 11.

If $(i_1, j_1, v_1)$ in the j-list of $S_{k-1,q}$ is not dominated by $(i_1^\star, j_1^\star, v_1^\star)$, where $j_1 \geq j_1^\star$, then $i_1 < i_1^\star$, or $v_1 < v_1^\star$. So, we also have $i_1 < i_2^\star$, or $v_1 < v_2^\star = v_1^\star$. Hence, $(i_1, j_1, v_1)$ is not dominated by $(i_2^\star, j_2^\star, v_2^\star)$. This procedure is implemented in Lines 14 to 20. Thus, the domination operation based on the linear merge scheme works correctly. □

### 4.3. Analysis of time complexity

The key point for analyzing the time complexity is the upper bound on the number of elements in $S_{k,q}$.

**Lemma 1.** *The three 3-tuples $(i_1, j_1, v_1)$, $(i_1, j_2, v_2)$ and $(i_2, j_1, v_3)$ cannot all belong to the same $S_{k,q}$, where $v_1 = \max\{a_{i_1}, b_{j_1}\}$, $v_2 = \max\{a_{i_1}, b_{j_2}\}$ and $v_3 = \max\{a_{i_2}, b_{j_1}\}$, for $1 \leq i_1 < i_2 \leq m$ and $1 \leq j_1 < j_2 \leq n$.*

**Proof.** We prove it by contradiction. Assume that $(i_1, j_1, v_1)$, $(i_1, j_2, v_2)$ and $(i_2, j_1, v_3)$ are all in the same $S_{k,q}$. We discuss the following two conditions.

Case 1: $(i_1, j_1, v_1)$ and $(i_1, j_2, v_2)$ are in the same $S_{k,q}$, where $v_1 = \max\{a_{i_1}, b_{j_1}\}$ and $v_2 = \max\{a_{i_1}, b_{j_2}\}$. There are four subcases as follows.

Subcase 1.1: $v_1 = a_{i_1}$ and $v_2 = a_{i_1}$. Consequently, $(i_1, j_1, v_1)$ dominates $(i_1, j_2, v_2)$, since $j_1 < j_2$. Thus, $(i_1, j_1, v_1)$ and $(i_1, j_2, v_2)$ cannot both belong to the same $S_{k,q}$. We conclude that this subcase is impossible.

Subcase 1.2: $v_1 = a_{i_1}$ and $v_2 = b_{j_2}$. Since $a_{i_1} < b_{j_2}$, it follows that $(i_1, j_1, v_1)$ dominates $(i_1, j_2, v_2)$. Thus, it is impossible for this subcase.

Subcase 1.3: $v_1 = b_{j_1}$ and $v_2 = a_{i_1}$. Then $b_{j_1} > a_{i_1}$, meaning there is no domination. We conclude that it is possible for this subcase.

Subcase 1.4: $v_1 = b_{j_1}$ and $v_2 = b_{j_2}$. It is possible that no domination occurs in this subcase.

Hence, in Case 1, we have that $b_{j_1} > a_{i_1}$.

Case 2: $(i_1, j_1, v_1)$ and $(i_2, j_1, v_3)$ are in the same $S_{k,q}$, where $v_1 = \max\{a_{i_1}, b_{j_1}\}$ and $v_3 = \max\{a_{i_2}, b_{j_1}\}$. Similar to Case 1, we can conclude that $a_{i_1} > b_{j_1}$.

It is a contradiction, since we get $b_{j_1} > a_{i_1}$ in Case 1 and $a_{i_1} > b_{j_1}$ in Case 2.

Therefore, the assumption that $(i_1, j_1, v_1)$, $(i_1, j_2, v_2)$ and $(i_2, j_1, v_3)$ are all in the same $S_{k,q}$ is not true. □

To calculate the possibly maximal number of 3-tuple elements in $S_{k,q}$, we need to transform it to a problem that an upper-left triangle cannot be marked in a grid at the same time.

**Definition 10** *(Upper-left triangle problem).* An *upper-left triangle* represents a right triangle (90-degree angle) with the right angle at the upper-left corner, in which the first index is counted from top to bottom, and the second index is counted from left to right, similar to the row-major mapping in a 2-dimensional array or grid. The *upper-left triangle problem* aims to find a set of points in a grid such that every three points in the set cannot form an upper-left triangle.

For example, in a two-dimensional grid with the row-major mapping (row $i$, column $j$), $(2,0)$, $(2,2)$, and $(3,0)$ form an upper-left triangle. We first present a lemma for the transformed problem as follows.

**Lemma 2.** *Given an $m \times n$ grid, the maximal cardinality of the set obtained in the upper-left triangle problem is at most $m + n - 1$.*

**Proof.** Suppose that $h_i$ points are marked in row $i$ and included in the set, where $1 \leq i \leq m$. Our goal is to find an upper bound for $\sum_{i=1}^{m} h_i$.

Since every three marked points cannot form an upper-left triangle, we obtain a property. If row $i$ has been marked with $h_i$ points, none of the lower rows can be marked with points in the same columns except for the column with the last marked point. So, when the first row has marked with $h_1$ points, the second row can be marked with at most $n - h_1 + 1$ points. And when the second row has marked with $h_2$ points, the third row can be marked with at most $n - (h_1 + h_2) + 2$ points. The following rows can be calculated similarly. Suppose there is a virtual row, numbered as $m + 1$. Then row $m + 1$ can be marked up to $n - (h_1 + h_2 + \cdots + h_m) + m$ points. Since the point at the last column in each row can be marked, we get

$$n - (h_1 + h_2 + \cdots + h_m) + m \geq 1$$

Rearrange the inequality,

$$m + n - 1 \geq h_1 + h_2 + \cdots + h_m$$

Thus, the theorem holds. □

As an example, in the above proof, suppose we are given a $5 \times 6$ grid, and we mark the first row at $(1,1)$, $(1,2)$, $(1,5)$, and $(1,6)$. Then the lower rows can only be marked in columns 3, 4, and 6.

The following lemma describes how to convert the problem of finding the maximal number of elements (3-tuples) in $S_{k,q}$ to the upper-left triangle problem.

**Lemma 3.** *The maximal number of 3-tuple elements in $S_{k,q}$ is equivalent to the maximum cardinality of the set obtained in the upper-left triangle problem with an $m \times n$ grid.*

**Proof.** Suppose that the lengths of sequences $A$ and $B$ are $m$ and $n$, respectively. Then, there are $m \times n$ 3-tuple combinations. When $i$ and $j$ in a 3-tuple $(i, j, v)$ are fixed, the value of $v$ is also fixed, because $v = \max\{a_i, b_j\}$. Thus, an $m \times n$ grid is equivalent to all possible 3-tuples. When a 3-tuple $(i_1, j_1, v_1) \in S_{k,q}$, we can represent $(i_1, j_1, v_1)$ as a marked point $(i_1, j_1)$ in the grid. By Lemma 1, it is impossible that $(i_1, j_1, v_1)$, $(i_1, j_2, v_2)$ and $(i_2, j_1, v_3)$, where $1 \le i_1 < i_2 \le m$ and $1 \le j_1 < j_2 \le n$, are all in the same $S_{k,q}$. These three 3-tuples correspond to an upper-left triangle. Thus, the maximal number of 3-tuple elements in $S_{k,q}$ is equivalent to the maximum cardinality of the set obtained in the upper-left triangle problem. □

**Theorem 6.** $|S_{k,q}| \le m + n - 1$.

**Proof.** By Lemmas 2 and 3, there are at most $m + n - 1$ elements in $S_{k,q}$. □

**Theorem 7.** *DOMINATE*$(S_{k-1,q} \cup$ *EXTEND*$(S_{k-1,q-1}, t_k))$ *can be done in $O(m + n)$ time.*

**Proof.** By Theorem 4, each element of $S_{k-1,q-1}$ can be extended to at most two new elements by the extension with $t_k$. By Theorem 6, the number of extended elements is $O(m + n)$. With the bucket method presented in Function 1, all dominated elements within the extension have already been removed, which requires $O(m + n)$ time. Then, by Theorems 5 and 6, $S_{k-1,q}$ has at most $m + n - 1$ elements. Thus, the domination by linear merging of $S_{k-1,q}$ and $H =$ EXTEND$(S_{k-1,q-1}, t_k)$ can be done in $O(m + n)$ time. □

**Theorem 8.** *Algorithm 1 solves the MLCIS problem in $O((L + 1)(r - L + 1)(m + n))$ time and $O(r(m + n))$ space.*

**Proof.** In Line 1, the preprocessing of arrays $\text{next}_A$ and $\text{next}_B$ requires $O(|\Sigma|(m + n))$ time and space for the feasible alphabet $\Sigma$ in $T$. In the worst case, where all values in $T$ are distinct, it requires $O(r(m + n))$ time and space.

The outer loop in Line 3 is executed $\max\{1, r - L\}$ times, because it stops when $i \ge r - L$. The inner loop in Line 5 is executed at most $L + 1$ times. So, Line 7 is invoked $O((L + 1)(r - L + 1))$ times. When $r \ge 1$, we have $r \le (L + 1)(r - L + 1)$, because $0 \le L \le r$. Based on Theorem 7, the MLCIS problem can be solved in $O((L + 1)(r - L + 1)(m + n))$ time.

Each $S_{k,q}$ has $O(m+n)$ elements by Theorem 6. The space of $S_{k,q}$ in each round can be reused. So we only need $O(L(m+n))$ space for one round. Since $L \le r$ and $|\Sigma| \le r$, the total space complexity is $O(r(m + n))$. □

## 5. Experimental results

In this section, we compare the execution time of the dynamic programming approach and the diagonal method. These two algorithms are implemented using Visual Studio 2022 using the C++17 compiler. We conduct experiments on a computer equipped with a 64-bit Windows 10 operating system, a CPU clocked at 2.9GHz (Intel Core i7-10700), and 16 GB of memory.

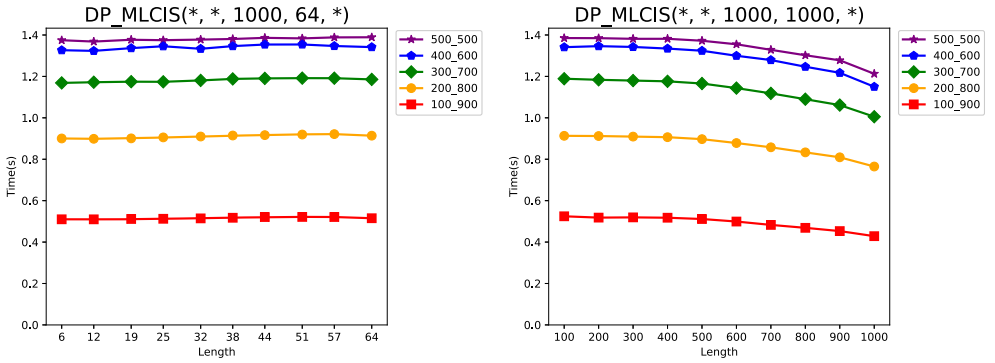
With slight modification, these two algorithms can also solve the *merged longest common weakly increasing subsequence* (MLCWIS) problem, where the answer is weakly increasing (non-decreasing), rather than strictly increasing.

**Definition 11** *(MLCWIS).* Given a pair of numeric sequences $A = \langle a_1, a_2, \cdots, a_m \rangle$ and $B = \langle b_1, b_2, \cdots, b_n \rangle$, along with a target sequence $T = \langle t_1, t_2, \cdots, t_r \rangle$, the *merged longest common weakly increasing subsequence* (MLCWIS) problem is to find the longest common non-decreasing (weakly increasing) subsequence of $E(A, B)$ and $T$, where $E(A, B)$ is the merged sequence obtained from merging subsequences of $A$ and $B$ arbitrarily while retaining their original orders in $A$ and $B$ individually.

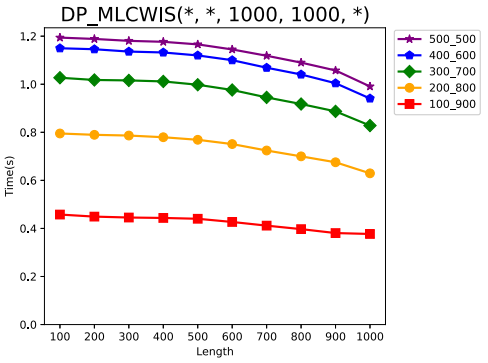
For solving the MLCWIS problem, we can slightly modify the MLCIS algorithm, by replacing $(v < t_k)$ with $(v \le t_k)$ in Line 4 of Function 1. In the experiments, we test the relationship between execution time and the lengths of MLCIS and MLCWIS with pseudo-random datasets. The pseudo-random datasets are generated with combinations of various parameters $(|A|, |B|, |T|, |\Sigma|, L)$, where $|A| = \lambda \times 1000$, $|B| = (1 - \lambda) \times 1000$, $|T| = 1000$, $|\Sigma| \in \{4, 64, 256, 1000\}$, $L$ is the answer length and $\lambda \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. The maximum answer length in the MLCIS problem is $|\Sigma|$, and the maximum answer length in the MLCWIS problem is $|T|$.

In order to get more accurate execution time, we perform 100 experiments and calculate the average. We use a 5-tuple ($|A|$, $|B|$, $|T|$, $|\Sigma|$, $L$) to represent various parameters settings in the figures. For example, in the notation ($*$, $*$, 1000, 4, $*$), the first "$*$", second "$*$", and third "$*$" represent all possible lengths of $A$, $B$, and $L$, respectively, while keeping $|T| = 1000$ and $|\Sigma| = 4$.

Here, we illustrate some of the experimental results. For the MLCIS and MLCWIS problems, Figs. 5 and 6 show the average execution time by using the DP algorithm and the diagonal algorithm [12,13], respectively. As we can see in Fig. 5, the larger $|A| \times |B|$ is, the more time it takes. In Figs. 6(b) and 6(c), the maximum execution time occurs at $L = 500$, because each dominating set has almost the maximum number of elements at length $L = 500$. This is also consistent with the time complexity $O((L + 1)(r - L + 1)(m + n))$, whose maximum value occurs at about $L = \frac{r}{2} = 500$. In Fig. 7, we compare the DP algorithm and the diagonal algorithm for different answer lengths. The execution time of the diagonal algorithm is much less than that of the DP algorithm.




(a) MLCIS, $|\Sigma| = 64$. (b) MLCIS, $|\Sigma| = 1000$.




(c) MLCWIS, $|\Sigma| = 1000$.

**Fig. 5.** The average execution time of the DP algorithm for different answer lengths, where $|T| = 1000$ [12,13].

## 6. Conclusion

In this paper, we propose two algorithms to solve the merged longest common increasing subsequence (MLCIS) problem, the dynamic programming approach in O($mnr$) time and the diagonal method in O($(L+1)(r-L+1)(m+n)$) time, where $m$, $n$, $r$, and $L$ denote the lengths of sequences $A$, $B$, $T$, and the MLCIS answer, respectively. Experimental results demonstrate that the diagonal algorithm outperforms the dynamic programming algorithm. The diagonal algorithm exhibits high efficiency when $L$ is either very small or close to $r$.

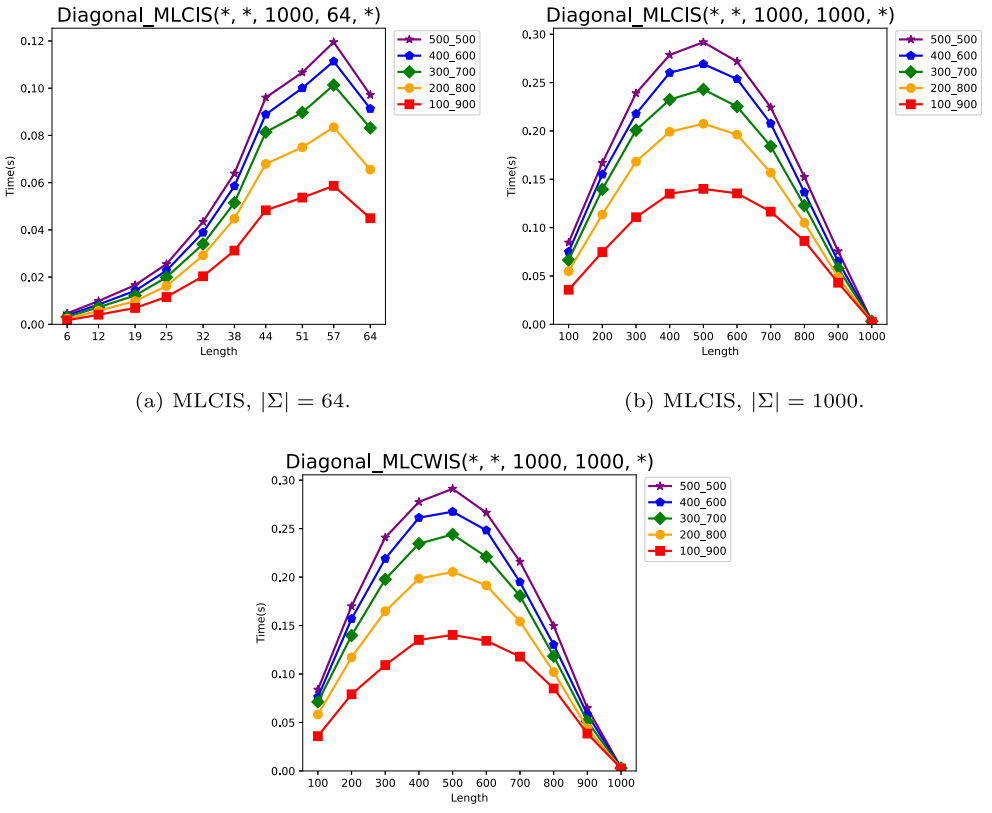
In the future, we may study other related problems. For example, given a sequence $T$, the *split longest common subsequence* problem aims to maximize $LCS(A, B)$ by splitting $T$ into two sequences $A$ and $B$ in order. Similarly, given a sequence $T$, the *split longest common increasing subsequence* problem aims to split $T$ into two sequences $A$ and $B$ in order such that $LCIS(A, B)$ is maximized. These are all interesting questions.

## CRediT authorship contribution statement

**Chien-Ting Lee:** Writing – original draft, Methodology, Investigation, Conceptualization. **Chang-Biau Yang:** Writing – review & editing, Validation, Supervision, Funding acquisition, Conceptualization. **Kuo-Si Huang:** Writing – review & editing, Validation, Methodology, Conceptualization.

## Declaration of competing interest

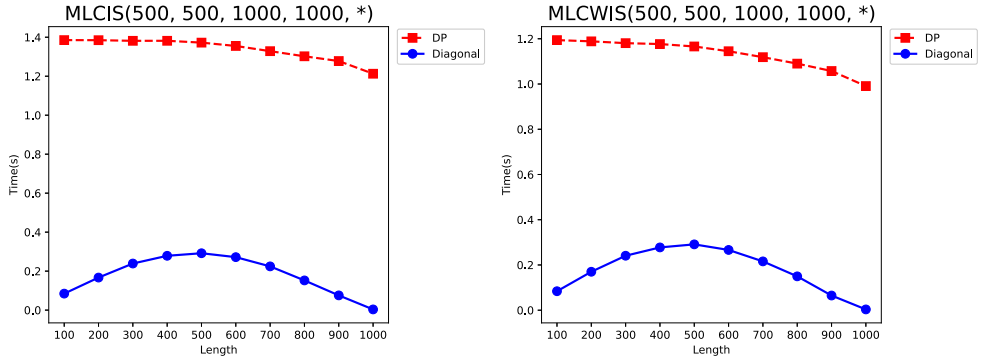
The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Chang-Biau Yang reports financial support was provided by National Science and Technology Council of Taiwan under contract NSTC 112-2221-E-110-026-MY2. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.




(a) MLCIS, $|\Sigma| = 64$. (b) MLCIS, $|\Sigma| = 1000$.

(c) MLCWIS, $|\Sigma| = 1000$.

**Fig. 6.** The average execution time of the diagonal algorithm for different answer lengths, where $|T| = 1000$ [13].




(a) MLCIS (b) MLCWIS

**Fig. 7.** The average execution time of the DP algorithm and diagonal algorithm on the MLCIS and MLCWIS problems for different answer lengths, where $|A| = 500$, $|B| = 500$, $|T| = 1000$ and $|\Sigma| = 1000$ [12,13].

## Acknowledgements

This research work was partially supported by National Science and Technology Council of Taiwan under contract NSTC 112-2221-E-110-026-MY2.

## Data availability

No data was used for the research described in the article.

## References


[1] Anadi Agrawal, Pawel Gawrychowski, A faster subquadratic algorithm for the longest common increasing subsequence problem, in: Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC 2020), Hong Kong, December 14-18, 2020, pp. 4:1–4:12.

[2] Gerth Stølting Brodal, Kanela Kaligosi, Irit Katriel, Martin Kutz, Faster algorithms for computing longest common increasing subsequences, in: Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM), Barcelona, Spain, 2006, pp. 330–341.

[3] Danlin Cai, Daxin Zhu, Lei Wang, Xiaodong Wang, A simple linear space algorithm for computing a longest common increasing subsequence, IAENG Int. J. Comput. Sci. 45 (3) (2018) 472–477.

[4] Wun-Tat Chan, Yong Zhang, Stanley P.Y. Fung, Deshi Ye, Hong Zhu, Efficient algorithms for finding a longest common increasing subsequence, J. Comb. Optim. 13 (2007) 277–288.

[5] Sebastian Deorowicz, Agnieszka Danek, Bit-parallel algorithms for the merged longest common subsequence problem, Int. J. Found. Comput. Sci. 24 (i) (2013) 1281–1298.

[6] Lech Duraj, A sub-quadratic algorithm for the longest common increasing subsequence problem, in: Proceedings of the 37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020), Montpellier, France, March 10-13, 2020, pp. 41:1–41:18.

[7] Daniel S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.

[8] Kuo-Si Huang, Chang-Biau Yang, Kuo-Tsung Tseng, Hsing-Yen Ann, Yung-Hsing Peng, Efficient algorithms for finding interleaving relationship between sequences, Inf. Process. Lett. 105 (5) (2008) 188–193.

[9] James W. Hunt, Thomas G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (5) (1977) 350–353.

[10] Manolis Kellis, Bruce W. Birren, Eric S. Lander, Proof and evolutionary analysis of ancient genome duplication in the yeast Saccharomyces cerevisiae, Nature 428 (2004) 617–624.

[11] Martin Kutz, Gerth Stølting Brodal, Kanela Kaligosi, Irit Katriel, Faster algorithms for computing longest common increasing subsequences, J. Discret. Algorithms 9 (4) (2011) 314–325.

[12] Chien-Ting Lee, Chang-Biau Yang, Kuo-Si Huang, The merged longest common increasing subsequence problem, in: Proceedings of the 16th Asian Conference on Intelligent Information and Database Systems (ACIIDS 2024), Ras Al Khaimah, UAE, April 15-18, 2024, pp. 202–212.

[13] Chien-Ting Lee, Chang-Biau Yang, Kuo-Si Huang, The diagonal method for the merged longest common increasing subsequence problem, in: 41st Workshop on Combinatorial Mathematics and Computation Theory, Taipei, Taiwan, May 2024, pp. 2–7.

[14] Shou-Fu Lo, Kuo-Tsung Tseng, Chang-Biau Yang, Kuo-Si Huang, A diagonal-based algorithm for the longest common increasing subsequence problem, Theor. Comput. Sci. 815 (2020) 69–78.

[15] David Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.

[16] Narao Nakatsu, Yahiko Kambayashi, Shuzo Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inform. 18 (2) (1982) 171–179.

[17] Yung-Hsing Peng, Chang-Biau Yang, Kuo-Si Huang, Chiou-Ting Tseng, Chiou-Yi Hor, Efficient sparse dynamic programming for the merged LCS problem with block constraints, Int. J. Innov. Comput. Inf. Control 6 (4) (2010) 1935–1947.

[18] Mahfuzur Rahman, M. Sohel Rahman, Effective sparse dynamic programming algorithms for merged and block merged LCS problems, J. Comput. 9 (8) (2014) 1743–1754.

[19] Yoshifumi Sakai, A linear space algorithm for computing a longest common increasing subsequence, Inf. Process. Lett. 99 (5) (2006) 203–207.

[20] Cheng-Kai Shiau, Jia-Hsin Huang, Yu-Ting Liu, Huai-Kuang Tsai, Genome-wide identification of associations between enhancer and alternative splicing in human and mouse, BMC Genomics 22 (2022) 919.

[21] Kuo-Tsung Tseng, De-Sheng Chan, Chang-Biau Yang, Shou-Fu Lo, Efficient merged longest common subsequence algorithms for similar sequences, Theor. Comput. Sci. 708 (2018) 75–90.

[22] Robert A. Wagner, Michael J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.

[23] I-Hsuan Yang, Chien-Pin Huang, Kun-Mao Chao, A fast algorithm for computing a longest common increasing subsequence, Inf. Process. Lett. 93 (5) (2005) 249–253.