

An Efficient Algorithm for Solving the Longest Common Square Subsequence Problem*

Shao-Xuan Lee^a, Kuo-Si Huang^b and Chang-Biau Yang^{a†}

^aDepartment of Computer Science and Engineering

National Sun Yat-sen University, Kaohsiung, Taiwan

^bDepartment of Business Computing

National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan

Abstract

A string is a square if it can be divided into two identical substrings. The longest common square subsequence (LCSqS) problem is to find a longest square such that it is also a subsequence of both two given sequences A and B of lengths m and n ($m \leq n$), respectively. This problem can be directly solved in $\mathcal{O}(n^6)$ time or $\mathcal{O}(|M|n^4)$ time with $\mathcal{O}(n^4)$ space, where M is the set of matching points between A and B . In 2018, Inoue et al. proposed the LCSqS algorithm in $\mathcal{O}(|\Sigma||M|^3 + n)$ time and $\mathcal{O}(|M|^2 + n)$ space, or in $\mathcal{O}(|M|^3 \log^2 n \log \log n + n)$ time and $\mathcal{O}(|M|^3 + n)$ space, where Σ is the alphabet in A and B . This paper proposes an efficient LCSqS algorithm by combining the S-table and the diagonal method for solving the 4-LCS problem, where the S-table is a useful data structure for the LCS and the edit distance problems. Our time and space complexities are $\mathcal{O}(L(m - L)m^2n^2 \log n/|\Sigma|)$ and $\mathcal{O}(mnL/|\Sigma|)$, respectively, where L is the LCSqS length. As experimental results show, our algorithm is more efficient in some cases such as when $|\Sigma| \leq 4$, and the sequence similarity is higher than 60% for $|\Sigma| = 20$ or higher than 90% for $|\Sigma| = 64$.

Keywords: longest common square subsequence (LCSqS), longest common subsequence (LCS), S-table, diagonal-based algorithm

1 Introduction

The *longest common subsequence* (LCS) problem is well-known and widely used to measure the similarity of strings [2–5, 12, 19, 22]. Given two sequences A and B , the LCS can indicate the degree of their similarity. The LCS problem has been discussed for decades of years, and it has also been widely used in various fields, such as bioinformatics, speech recognition, string comparison, pattern matching and so on [6, 8]. The LCS problem has several variations for various requirements, such as the *longest common increasing subsequence* (LCIS) [9, 15, 20, 24], the *merged longest common subsequence* (MLCS) [13, 18, 23], the *longest common palindromic subsequence* (LCPS) [7, 10, 16, 21], and the *longest common square subsequence* (LCSqS) [11, 14] problems.

Let two input sequences $A = a_1a_2 \dots a_m$ and $B = b_1b_2 \dots b_n$, $m \leq n$, where m and n denote the lengths of A and B , respectively. A *subsequence* can be obtained by deleting some characters from the original sequence, while a *substring* is a segment of some contiguous characters in the original sequence. The LCS problem is to find the common subsequence of A and B with the maximal length. A string is *square* if it can be partitioned into two identical substrings. This paper focuses on the LCSqS problem to find the LCS of A and B such that this LCS is also a square. For example, given $A = \text{cgatttgat}$ and $B = \text{tcgttagttg}$, we have $\text{LCS}(A, B) = \text{cgattg}$, $\text{LCSqS}(A, B) = \text{gtgt}$.

The LCSqS problem was first proposed by Inoue et al. [14], and they solved the LCSqS problem in $\mathcal{O}(|\Sigma||M|^3 + n)$ time and $\mathcal{O}(|M|^2 + n)$ space, where M and Σ denote the set of matching points and the alphabet, respectively, in A and B . In this paper, we propose an efficient algorithm by combining the S-table and the diagonal method for solving the LCSqS problem, where the

*This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 109-2221-E-110-040-MY2.

†Corresponding author. E-mail: cbyang@cse.nsysu.edu.tw (Chang-Biau Yang).

S-table is useful for the LCS and the edit distance problems. Our time and space complexities are $\mathcal{O}(L(m-L)m^2n^2\log n/|\Sigma|)$ and $\mathcal{O}(mnL/|\Sigma|)$, respectively, where L is the LCSqS length. As experimental results show, our algorithm is more efficient especially for a small alphabet or sequences with high similarity.

The organization of this paper is given as follows. Section 2 introduces the LCSqS problem and the S-table. We present our algorithm in Section 3 and show the experimental results in Section 4. Finally, we give our conclusions in Section 5.

2 Preliminaries

A sequence or string is denoted by an upper-case letter, such as A or B . Taking sequence A as an example, the symbols used in this paper are given as follows.

- $|A|$: the length of sequence A .
- a_i : the i th character of A .
- $A_{i..j}$: the substring of A from indices i to j . Note that $A_{i..j} = \emptyset$ if $i > j$.
- \overleftarrow{A}_i : the prefix substring of A from indices 1 to i , that is, $\overleftarrow{A}_i = A_{1..i}$.
- \overrightarrow{A}_i : the suffix substring of A from indices $i+1$ to $|A| = m$, that is, $\overrightarrow{A}_i = A_{i+1..m}$.

2.1 The Longest Common Square Subsequence Problem

Let X be a non-empty sequence of even length. If X can be represented as YY , we regard X as a *square*. For example, `aa`, `acac`, `aataat` and `agcgagcg` are squares. The *longest common square subsequence* (LCSqS) problem is to find the common square subsequence of A and B with the maximal length. Let $\text{LCSqS}(A, B)$ denote the LCSqS content of sequences A and B . For example, given $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$, we have $\text{LCSqS}(A, B) = \text{gtgt}$ and $|\text{LCSqS}(A, B)| = 4$.

Given $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$, let M denote the set of all matching points (i, j) of A and B for $a_i = b_j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Inoue *et al.* first defined the LCSqS problem in 2018 [14]. They showed that the LCSqS problem can be solved in $\mathcal{O}(n^6)$ time or $\mathcal{O}(|M|n^4)$ time with $\mathcal{O}(n^4)$ space. They also proposed an improved algorithm with $\mathcal{O}(|\Sigma||M|^3 + n)$ time and $\mathcal{O}(|M|^2 + n)$ space, where Σ is the alphabet in A and B .

By dividing A into two substrings and B into two substrings, there are $\mathcal{O}(n^2)$ pairs of cut points. The LCSqS problem can be solved by computing the 4-LCS (the LCS of four strings) with these $\mathcal{O}(n^2)$ pairs of partitioning of A and B . Since the 4-LCS can be computed in $\mathcal{O}(n^4)$ time, the LCSqS length can be found by the dynamic programming (DP) formula in Equation 1 [14] with $\mathcal{O}(n^6)$ time and $\mathcal{O}(n^4)$ space.

$$\begin{aligned} \text{LCSqS}(A, B) = \\ 2 \times \max_{1 \leq i < m, 1 \leq j < n} \{\text{LCS}(A_{1..i}, A_{i+1..m}, B_{1..j}, B_{j+1..n})\} \end{aligned} \quad (1)$$

For any matching point $(i, j) \in M$, if there exists the smallest position $(i', j') \in M$ such that $i < i'$ and $j < j'$, then for any (p, q) , $i \leq p < i'$ and $j \leq q < j'$, we have $\text{LCS}(A_{1..p}, A_{p+1..m}, B_{1..q}, B_{q+1..n}) = \text{LCS}(A_{1..i}, A_{i+1..m}, B_{1..j}, B_{j+1..n})$. This algorithm considers only $|M|$ partition points. Hence, it can solve the LCSqS problem in $\mathcal{O}(|M|n^4)$ time and $\mathcal{O}(n^4)$ space.

Inoue *et al.* [14] also proposed a rectangle-based algorithm with $\mathcal{O}(|\Sigma||M|^3 + n)$ time for solving the LCSqS problem. Let a 4-tuple $r = (i, j, k, l)$ denote a *matching rectangle* for $a_i = a_j = b_k = b_l = \alpha$, where $1 \leq i < j \leq m$, $1 \leq k < l \leq n$, and $\alpha \in \Sigma$. They try to fix the rectangles with nondominated lower left points, and extends the rectangles in the upper right direction. The number of the nondominated lower left rectangles is $\mathcal{O}(|M|)$. Then, it examines all $\mathcal{O}(|M|^2)$ matching rectangles and calculates $\mathcal{O}(|\Sigma|)$ rectangles in each extension round. Thus, the algorithm can compute the length of $\text{LCSqS}(A, B)$ with $\mathcal{O}(|\Sigma||M|^3 + n)$ time and $\mathcal{O}(|M|^2 + n)$ space.

2.2 The S-table of $A\alpha$ and B

Given two sequences $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$, the S-table records the LCS length between A and each suffix of B . The S-table is an efficient data structure for the LCS and the edit distance problems. Some properties of the S-table, including v_i in Definition 2, were proposed by Alves *et al.* [11].

Definition 1. [11] (S-table) *In the 2-D S-table of $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$, $S_{i,k}$ denotes the minimal index j of B such that the LCS length of A and $B_{i+1..j}$ is k , where $1 \leq i+1 \leq j \leq n$.*

Definition 2. [11] (V) *v_i records the element that appears in $S_{i,*}$, but does not appear in $S_{i-1,*}$, where $1 \leq i \leq n$. If there is no such element, $v_i = \infty$.*

Table 1: The S-table S of $A = \text{cgga}$ and $B = \text{tcgttagttg}$. The value i in the leftmost column indicates the suffix $B_{i+1..n}$ in row i , and the value v_i in column V indicates that v_i is a new element in row i , which does not exist in row $i - 1$.

i	Length				V	LCS
	0	1	2	3		
0	0	2	3	5	∞	3
1	1	2	3	5	1	3
2	2	3	5	∞	∞	2
3	3	5	9	∞	9	2
4	4	5	9	∞	4	2
5	5	6	9	∞	6	2
6	6	9	∞	∞	∞	1
7	7	9	∞	∞	7	1
8	8	9	∞	∞	8	1
9	9	∞	∞	∞	∞	0

Table 1 shows the S-table S of $A = \text{cgga}$ and $B = \text{tcgttagttg}$. For example, $S_{1,3} = 5$ means that the LCS length of A and $B_{2..5} = \text{cgta}$ is 3, but $|\text{LCS}(A, B_{2..4})| = 2$, since index 5 of B is the minimal index to have the LCS length 3. In Table 1, each v_i is shown in the rightmost column. For example, $v_1 = 1$ means that element 1 in $S_{1,*}$ does not exist in $S_{0,*}$, $v_3 = 9$ means that element 9 in $S_{3,*}$ is not in $S_{2,*}$, and $v_6 = \infty$ means that each element in $S_{6,*}$ is in $S_{5,*}$. The linear-space S-table stores only the first row $S_{0,*}$ and column V of the S-table, not the full S-table.

Let S and V denote the S-table and linear-space S-table of A and B , respectively, and let S' and V' denote the S-table and linear-space S-table of $A\alpha$ and B , respectively. Note that α denotes a new character appended to the tail of A . Table 2 shows S' and V' of $A\alpha$ and B , where $\alpha = t$, corresponding to Table 1. Lin [17] proposed an efficient algorithm with $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space to build V' when V is given.

3 Our LCSqS Algorithm

In this paper, we propose an algorithm for solving the LCSqS problem, which is inspired by the concept of the linear-space S-table and the diagonal LCPS algorithm. A string is a *palindrome* if it is of the form $A = a_1a_2 \cdots a_m = a_ma_{m-1} \cdots a_2a_1$. Liang *et al.* [16] proposed the diagonal algorithm for solving the longest common palindromic subsequence (LCPS) problem with $\mathcal{O}(L(m - L)mn \log n / |\Sigma|)$ time, where m and n denote the lengths of A and B , respectively, L

Table 2: The S-table S' and V' of $A\alpha = \text{cggat}$ and $B = \text{tcgttagttg}$.

i	Length					V'	LCS
	0	1	2	3	4		
0	0	1	3	4	7	∞	4
1	1	2	3	4	7	2	4
2	2	3	4	7	∞	∞	3
3	3	4	7	∞	∞	∞	2
4	4	5	7	∞	∞	5	2
5	5	6	7	∞	∞	6	2
6	6	7	∞	∞	∞	∞	1
7	7	8	∞	∞	∞	8	1
8	8	9	∞	∞	∞	9	1
9	9	∞	∞	∞	∞	∞	0

denotes the LCPS length, and $|\Sigma|$ denotes the alphabet size. Here, we combine the linear-space S-table and the diagonal algorithm to solve the LCSqS problem.

Suppose two sequences $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$ are given. We divide A into the prefix substrings $A_{1..i}$, denoted as \overleftarrow{A}_i , and the suffix substring $A_{i+1..m}$, denoted as \overrightarrow{A}_i , where $1 \leq i \leq m - 1$, and divide B into the prefix substring $B_{1..j}$, denoted as \overleftarrow{B}_j , and the suffix substring $B_{j+1..n}$, denoted as \overrightarrow{B}_j , where $1 \leq j \leq n - 1$. For example, $\overleftarrow{A}_5 = a_1a_2 \cdots a_5$, $\overrightarrow{A}_5 = a_6a_7 \cdots a_m$, $\overleftarrow{B}_8 = b_1b_2 \cdots b_8$, and $\overrightarrow{B}_8 = b_9b_{10} \cdots b_n$.

Our goal is to get the LCS of \overleftarrow{A}_i , \overrightarrow{A}_i , \overleftarrow{B}_j and \overrightarrow{B}_j , for all possible combinations of i and j , $1 \leq i \leq m - 1$ and $1 \leq j \leq n - 1$. For example, suppose $A = \text{cggatttgc}$ and $B = \text{tcgttagttg}$, the length of $\text{LCS}(\overleftarrow{A}_2, \overrightarrow{A}_2, \overleftarrow{B}_3, \overrightarrow{B}_3) = g$ is 1, where $\overleftarrow{A}_2 = \text{cg}$, $\overrightarrow{A}_2 = \text{atttgc}$, $\overleftarrow{B}_3 = \text{tcg}$ and $\overrightarrow{B}_3 = \text{tagttg}$; the length of $\text{LCS}(\overleftarrow{A}_5, \overrightarrow{A}_5, \overleftarrow{B}_4, \overrightarrow{B}_4) = gt$ is 2, where $\overleftarrow{A}_5 = \text{cggtt}$, $\overrightarrow{A}_5 = \text{tgat}$, $\overleftarrow{B}_4 = \text{tcgt}$ and $\overrightarrow{B}_4 = \text{agttg}$, and so on. Then we can get the maximal length of $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{A}_i, \overleftarrow{B}_j, \overrightarrow{B}_j)$ is 2 when $i = 5$ and $j = 4$.

There are $\mathcal{O}(mn)$ possible cut points totally. For each pair (i, j) , $1 \leq i \leq m - 1$ and $1 \leq j \leq n - 1$, we first calculate the lengths of $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{A}_i)$, $\text{LCS}(\overleftarrow{A}_i, \overleftarrow{B}_j)$, $\text{LCS}(\overrightarrow{A}_i, \overleftarrow{B}_j)$, $\text{LCS}(\overrightarrow{A}_i, \overrightarrow{B}_j)$, $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{B}_j)$ and $\text{LCS}(\overrightarrow{B}_j, \overrightarrow{B}_j)$. Figure 1 shows the diagram of the above six LCS combinations. Since the LCS length is less than or equal to the above six LCS lengths, we keep the minimal length of them and sort these cut points according to these minimal lengths.

Let \overline{A} and \overline{B} denote the reverse sequences of A and B , respectively. For example, given two

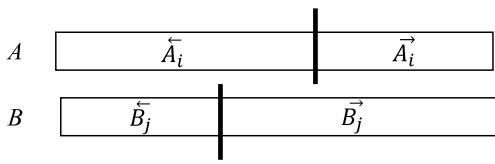


Figure 1: The schematic diagram of four substrings. According to the cut points, A is divided into two substrings \bar{A}_i and \vec{A}_i , and B is divided into two substrings \bar{B}_j and \vec{B}_j , where $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$.

Table 3: The LCS table of $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$. It shows the lengths of $\text{LCS}(\bar{A}_i, \bar{B}_j)$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.

	t	c	g	t	a	g	t	t	g	
0	0	0	0	0	0	0	0	0	0	0
c	1	0	0	1	1	1	1	1	1	1
g	2	0	0	1	2	2	2	2	2	2
g	3	0	0	1	2	2	2	3	3	3
a	4	0	0	1	2	2	3	3	3	3
t	5	0	1	1	2	3	3	3	4	4
t	6	0	1	1	2	3	3	3	4	5
g	7	0	1	1	2	3	3	4	4	5
a	8	0	1	1	2	3	4	4	4	5
t	9	0	1	1	2	3	4	4	5	6

sequence $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$, then $\bar{A} = \text{tagttaggc}$ and $\bar{B} = \text{gttgatgtt}$. We can use the traditional DP method to find the length of $\text{LCS}(\bar{A}_i, \bar{B}_j)$ and $\text{LCS}(\vec{A}_i, \vec{B}_j)$. Table 3 shows the lengths of all $\text{LCS}(\bar{A}_i, \bar{B}_j)$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. It requires $\mathcal{O}(mn)$ time to get the lengths of all $\text{LCS}(\bar{A}_i, \bar{B}_j)$. Similarly, we use the DP method to calculate $\text{LCS}(\bar{A}, \bar{B})$ to get the lengths of all $\text{LCS}(\bar{A}_i, \bar{B}_j)$ in $\mathcal{O}(mn)$ time. And, in the following, we will use the linear-space S-table to find the lengths of $\text{LCS}(\bar{A}_i, \vec{A}_i)$, $\text{LCS}(\bar{A}_i, \vec{B}_j)$, $\text{LCS}(\bar{B}_j, \vec{B}_j)$, and $\text{LCS}(\bar{A}_i, \vec{B}_j)$.

Based on the characteristics of S-table, we can obtain the lengths of A and \vec{B}_j , for $1 \leq j \leq n-1$. From the linear-space S-table of A and B , we can construct the linear-space S-table of $A\alpha$ and B with $\mathcal{O}(n)$ time. For example, as shown in Table 2, we can get the linear-space S-table of $A\alpha = \text{cgat}$ and $B = \text{tcgttagttg}$ from the given linear-space S-table of $A = \text{cgga}$ and $B = \text{tcgttagttg}$. In other words, with $\mathcal{O}(n)$ time, we can com-

pute $\text{LCS}(\bar{A}_{i+1}, \bar{B}_j)$ from $\text{LCS}(\bar{A}_i, \bar{B}_j)$, for all $1 \leq j \leq n-1$. Thus, the computation of $\text{LCS}(\bar{A}_i, \bar{B}_j)$, for all $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$, requires $\mathcal{O}(mn)$ time. The computation of $\text{LCS}(\bar{A}_i, \vec{A}_i)$, $\text{LCS}(\bar{B}_j, \bar{B}_j)$, and $\text{LCS}(\bar{B}_j, \vec{A}_i)$ can be done similarly.

Definition 3. $L_{i,j}$ records the minimal length of $\text{LCS}(\bar{A}_i, \vec{A}_i)$, $\text{LCS}(\bar{A}_i, \bar{B}_j)$, $\text{LCS}(\bar{A}_i, \vec{B}_j)$, $\text{LCS}(\vec{A}_i, \bar{B}_j)$, $\text{LCS}(\bar{A}_i, \vec{B}_j)$ and $\text{LCS}(\bar{B}_j, \bar{B}_j)$, for $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$, where m and n denote the lengths of A and B , respectively.

For example, suppose that $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$. Table 4 shows the LCS lengths of the six pairs of substrings. Table 5 shows $L_{i,j}$ retrieved from Table 4. For example, $L_{3,4} = \min\{\text{LCS}(\bar{A}_3, \bar{A}_4), \text{LCS}(\bar{A}_3, \bar{B}_4), \text{LCS}(\bar{A}_3, \vec{B}_4), \text{LCS}(\vec{A}_3, \bar{B}_4), \text{LCS}(\vec{A}_3, \vec{B}_4)\} = \min\{1, 2, 2, 3, 4, 2\}; L_{6,7} = \min\{3, 4, 1, 3, 1, 2\}$. We sort $L_{i,j}$ with the nonincreasing order, and then perform the 4-LCS algorithm starting from the maximal $L_{i,j}$ until the $L_{i,j}$ is less than the current maximal 4-LCS length.

By applying the DP approach with the linear-space S-table, we can get the minima of the LCS lengths of the six pairs of substrings in $\mathcal{O}(n^2)$ time. Then, we sort $L_{i,j}$ of $\mathcal{O}(mn)$ cut points nonincreasingly in $\mathcal{O}(mn \log(mn))$ time. And finally apply the diagonal algorithm to calculate the 4-LCS length starting from the cut point (i, j) corresponding to the maximal $L_{i,j}$.

For $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$, $L_{4,3}$ is maximal in Table 5. We have $\bar{A}_4 = \text{cgga}$, $\vec{A}_4 = \text{ttgat}$, $\bar{B}_3 = \text{tcg}$, $\vec{B}_3 = \text{tagttg}$. By the diagonal algorithm, $\text{LCS}(\bar{A}_4, \bar{A}_4, \bar{B}_3, \bar{B}_3) = 1$. Next, for cut point $(5,4)$, we get $\text{LCS}(\bar{A}_5, \bar{A}_5, \bar{B}_4, \bar{B}_4) = 2$ and $\text{LCS}_{\text{Sqs}}(A, B) = 2 \times \text{LCS} = 4$. Since $\text{LCS}(\bar{A}_5, \bar{A}_5, \bar{B}_4, \bar{B}_4)$ is equal to the length of the next cut point $L_{5,6}$, so we stop the algorithm and get the solution of $\text{LCS}_{\text{Sqs}}(A, B)$ is 4.

The pseudo code of our algorithm for solving the LCS_{Sqs} problem is presented in Algorithm 1. Its time complexity is $\mathcal{O}(L(m-L)m^2n^2 \log n / |\Sigma|)$ in the worst case, where m and n denote the lengths of A and B , respectively, L denotes the solution length and $|\Sigma|$ denotes the alphabet size. Theoretically, the time complexity is the same as the 4-LCS algorithm. We use the linear-space S-table to reduce the required calculation of cut points, it means that we can greatly reduce the computing time.

Table 4: The lengths of $\text{LCS}(\overleftarrow{A_i}, \overrightarrow{A_i})$, $\text{LCS}(\overleftarrow{A_i}, \overleftarrow{B_j})$, $\text{LCS}(\overleftarrow{A_i}, \overrightarrow{B_j})$, $\text{LCS}(\overrightarrow{A_i}, \overleftarrow{B_j})$, $\text{LCS}(\overrightarrow{A_i}, \overrightarrow{B_j})$ and $\text{LCS}(\overleftarrow{B_j}, \overrightarrow{B_j})$, for $1 \leq i \leq m - 1$ and $1 \leq j \leq n - 1$, where $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$.

	1	2	3	4	5	6	7	8
1	0, 0, 1 1, 5, 1	0, 1, 0 1, 5, 1	0, 1, 0 2, 4, 2	0, 1, 0 3, 4, 2	0, 1, 0 3, 4, 2	0, 1, 0 3, 3, 3	0, 1, 0 4, 2, 2	0, 1, 0 4, 1, 1
2	1, 0, 2 1, 5, 1	1, 1, 1 1, 5, 1	1, 2, 1 2, 4, 2	1, 2, 1 3, 4, 2	1, 2, 1 3, 4, 2	1, 2, 1 3, 3, 3	1, 2, 1 4, 2, 2	1, 2, 1 4, 1, 1
3	1, 0, 3 1, 4, 1	1, 1, 2 1, 4, 1	1, 2, 2 2, 4, 2	1, 2, 2 3, 4, 2	1, 2, 2 3, 3, 2	1, 3, 1 3, 3, 3	1, 3, 1 4, 2, 2	1, 3, 1 4, 1, 1
4	2, 0, 3 1, 3, 1	2, 1, 2 1, 3, 1	2, 2, 2 2, 3, 2	2, 2, 2 3, 3, 2	2, 3, 2 3, 3, 2	2, 3, 1 3, 3, 3	2, 3, 1 4, 2, 2	2, 3, 1 4, 1, 1
5	3, 1, 4 1, 3, 1	3, 1, 3 1, 3, 1	3, 2, 2 2, 3, 2	3, 3, 2 3, 2, 2	3, 3, 2 3, 2, 2	3, 3, 1 3, 2, 3	3, 4, 1 4, 2, 2	3, 4, 1 4, 1, 1
6	3, 1, 5 1, 3, 1	3, 1, 4 1, 3, 1	3, 2, 3 1, 2, 2	3, 3, 3 2, 2, 2	3, 3, 3 2, 2, 2	3, 3, 2 2, 1, 3	3, 4, 1 3, 1, 2	3, 5, 1 3, 1, 1
7	2, 1, 6 1, 2, 1	2, 1, 5 1, 2, 1	2, 2, 4 1, 2, 2	2, 3, 4 1, 2, 2	2, 3, 4 1, 1, 2	2, 4, 3 1, 1, 3	2, 4, 2 2, 1, 2	2, 5, 1 2, 0, 1
8	1, 1, 6 1, 1, 1	1, 1, 5 1, 1, 1	1, 2, 4 1, 1, 2	1, 3, 4 1, 1, 2	1, 4, 4 1, 1, 2	1, 4, 3 1, 1, 3	1, 4, 2 1, 1, 2	1, 5, 1 1, 0, 1

Table 5: The $L_{i,j}$ table of $A = \text{cggatttgat}$ and $B = \text{tcgttagttg}$.

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	1
3	0	1	1	1	1	1	1	1
4	0	1	2	2	2	1	1	1
5	1	1	2	2	2	1	1	1
6	1	1	1	2	2	1	1	1
7	1	1	1	1	1	1	1	0
8	1	1	1	1	1	1	1	0

Theorem 1. Algorithm 1 solves the LCSqS problem with $\mathcal{O}(L(m - L)m^2n^2 \log n / |\Sigma|)$ time and $\mathcal{O}(mnL / |\Sigma|)$ space in the worst case.

4 Experimental Results

The experiments were performed on a computer with CPU clock rate of 3.00GHz (Intel(R) Core(TM) i5-9500 CPU), 64-bit OS and RAM of 8 GB. Several sets of sequences with different parameter combinations are provided to evaluate our algorithm. We perform the experiments, 100 times for each parameter combination, to get the average execution time for the following algorithms.

- **DP:** It uses the traditional DP method to calculate the 4-LCS lengths for the $\mathcal{O}(mn)$ pos-

sible cut points. Its time and space complexities are $\mathcal{O}(m^3n^3)$ and $\mathcal{O}(m^2n^2)$, respectively.

- **Inoue:** It is the rectangle algorithm proposed by Inoue *et al.* [14]. Its time and space complexities are $\mathcal{O}(|\Sigma||M|^3 + n)$ and $\mathcal{O}(|M|^2 + n)$, respectively.
- **Diagonal:** It uses the diagonal method to calculate the 4-LCS lengths for the $\mathcal{O}(mn)$ possible cut points. Its time and space complexities are $\mathcal{O}(L(m - L)m^2n^2 \log n / |\Sigma|)$ and $\mathcal{O}(mnL / |\Sigma|)$, respectively.
- **Ours:** It is our algorithm proposed in Algorithm 1 with the linear space S-table. Its time and space complexities are $\mathcal{O}(L(m - L)m^2n^2 \log n / |\Sigma|)$ and $\mathcal{O}(mnL / |\Sigma|)$, respectively.

The 5-tuple $(|A|, |B|, |\Sigma|, \text{algorithm}, \text{similarity})$ represents parameters of each experiment. For example, $(100, 200, 2, *, *)$ means $|A| = 100$, $|B| = 200$, $|\Sigma| = 2$, and the two wildcard symbols stand for various algorithms and various similarities, respectively. The similarity θ is defined in Equation 2.

$$\theta(A, B) = \frac{\text{LCSqS}(A, B)}{\min(|A|, |B|)} \quad (2)$$

We use different methods for generating sequences of different similarities. In low similarities, we set $|A| = 100$, $\theta \in \{50\%, 60\%\}$ for $|\Sigma| = 2$; $\theta \in \{30\%, 40\%\}$ for $|\Sigma| = 4$; and $\theta \in \{10\%\}$ for

Algorithm 1 Computing LCSqS(A, B) Length.

Input: $A = a_1 a_2 \cdots a_m$, $B = b_1 b_2 \cdots b_n$
Output: Length of LCSqS(A, B)

- 1: Calculate $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{B}_j)$, for $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$. $\triangleright \text{LCS}(\overleftarrow{A}_i, \overleftarrow{B}_j)$
- 2: Calculate $\text{LCS}(\overrightarrow{A}_i, \overrightarrow{B}_j)$, for $1 \leq i \leq m-1$ and $1 \leq j \leq n-1$. $\triangleright \text{LCS}(\overrightarrow{A}_i, \overrightarrow{B}_j)$
- 3: Build the linear-space S-table V_{i+1} of $A_{1..i+1}$ and B from V_i , $A_{1..i}$ and B , $1 \leq i \leq m-1$.
- 4: $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{B}_j) \leftarrow$ number of infinite values in $V_i[j..n]$, $1 \leq j \leq n-1$. $\triangleright \text{LCS}(\overleftarrow{A}_i, \overrightarrow{B}_j)$
- 5: Build the linear-space S-table V_{j+1} of $B_{1..j+1}$ and A from V_j , $B_{1..j}$ and A , $1 \leq j \leq n-1$.
- 6: $\text{LCS}(\overrightarrow{A}_i, \overleftarrow{B}_j) \leftarrow$ number of infinite values in $V_j[i..m]$, $1 \leq i \leq m-1$. $\triangleright \text{LCS}(\overrightarrow{A}_i, \overleftarrow{B}_j)$
- 7: Build the linear-space S-table V_{i+1} of $A_{1..i+1}$ and A from V_i , $A_{1..i}$ and A , $1 \leq i \leq m-1$.
- 8: $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{A}_i) \leftarrow$ number of infinite values in $V_i[i..m]$, $1 \leq i \leq m-1$. $\triangleright \text{LCS}(\overleftarrow{A}_i, \overrightarrow{A}_i)$
- 9: Build the linear-space S-table V_{j+1} of $B_{1..j+1}$ and B from V_j , $B_{1..j}$ and B , $1 \leq j \leq n-1$.
- 10: $\text{LCS}(\overleftarrow{B}_j, \overrightarrow{B}_j) \leftarrow$ number of infinite values in $V_j[j..n]$, $1 \leq j \leq n-1$. $\triangleright \text{LCS}(\overleftarrow{B}_j, \overrightarrow{B}_j)$
- 11: $L_{i,j} \leftarrow \min\{\text{LCS}(\overleftarrow{A}_i, \overleftarrow{B}_j), \text{LCS}(\overrightarrow{A}_i, \overrightarrow{B}_j), \text{LCS}(\overleftarrow{A}_i, \overrightarrow{B}_j), \text{LCS}(\overrightarrow{A}_i, \overleftarrow{B}_j), \text{LCS}(\overrightarrow{A}_i, \overrightarrow{A}_i), \text{LCS}(\overleftarrow{B}_j, \overrightarrow{B}_j)\}$.
- 12: Sort($L_{i,j}$) with nondecreasing order
- 13: **for** $L_{i,j}$ nondecreasingly **do**
- 14: $l \leftarrow$ length of $\text{LCS}(\overleftarrow{A}_i, \overrightarrow{A}_i, \overleftarrow{B}_j, \overrightarrow{B}_j)$ with the diagonal method
- 15: **if** $l \geq$ next $L_{i,j}$ **then**
- 16: break
- 17: **return** $l \times 2$

$|\Sigma| = 20$. The procedure (S1 to S4) is used to generate experimental sequences of low similarities.

- S1: To generate sequences of length $|A| = |B| = n$ with similarity $= \theta$, we first generate a sequence X with length $n \times \theta$ in random.
- S2: Let $A = XX$ and randomly insert $n-2(n \times \theta)$ characters in random positions.
- S3: Let $B = XX$. Randomly select a character $\alpha \in \Sigma$, and randomly select a number t , where $1 \leq t \leq n/|\Sigma|$. Append t characters of α to B . Repeat this step until $|B| = n$.
- S4: Calculate LCSqS(A, B). If it is close to the target similarity ($\theta \pm 2\%$), take A and B into the experimental dataset, and stop; otherwise, go to step S1.

After treating the low similarities as described in the above procedure, we use the following procedure (G1 to G3) to generate the remaining sequence sets of various similarities.

- G1: To generate $|A| = |B| = n$, we first generate a sequence X in random such that $|X| = n/2$. If all of the characters in Σ are used, then go to step G2; otherwise, repeat step G1.
- G2: Set $A = XX$ and $B = XX$. In A and B , we randomly delete h_1 characters, randomly insert h_1 characters, and randomly replace h_2 characters, where $h_1 : h_2 = 2 : 1$. The value

of h_1 is related to the similarity. If the similarity is low, then h_1 will be large. For example, when $|\Sigma| = 2$, we set $h_1 = 2$ and $h_2 = 1$ for $\theta = 90\%$; set $h_1 = 6$ and $h_2 = 3$ for $\theta = 80\%$; and set $h_1 = 20$ and $h_2 = 10$ for $\theta = 70\%$.

- G3: Calculate LCSqS(A, B). If it is close to the target similarity ($\theta \pm 2\%$), take A and B into the experimental dataset, and stop; otherwise, go to step G1.

Figure 2 shows the execution time of the four algorithms for various input lengths and $|\Sigma| \in \{2, 4, 20, 64\}$. We observe that our algorithm is more efficient than the Inoue algorithm [14] when $|\Sigma| \in \{2, 4\}$, whereas the Inoue algorithm is more efficient for $|\Sigma| \in \{20, 64\}$. Our algorithm performs better when Σ is small. As $|\Sigma|$ increases, the execution time of the Inoue algorithm decreases. The execution time of the Inoue algorithm depends on the matching points between A and B . The smaller Σ is, the more matching points exist. Hence, for small Σ , the Inoue algorithm cannot be accomplished for long input sequences, because no enough space to deal with all rectangles.

Figures 3 and 4 show the computational amounts of the diagonal algorithm with or without the linear-space S-table. It shows that when the alphabet is small or the similarity is high, our algorithm, the diagonal algorithm with the linear-space S-table, reduces computation significantly.

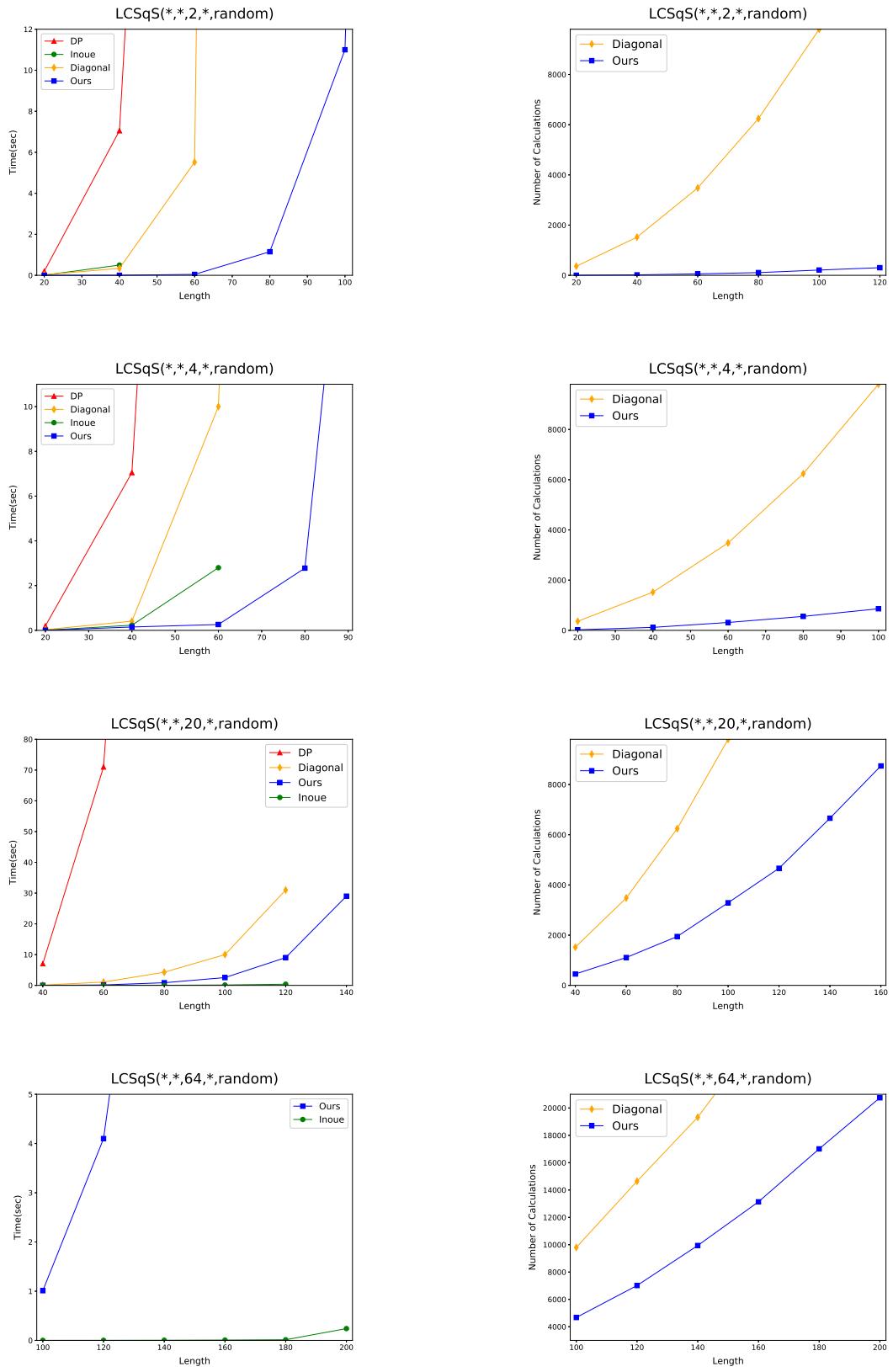


Figure 2: The execution time (in seconds) of the four algorithms for various sequence lengths.

Figure 3: The computational amounts of the diagonal and our algorithms for various sequence lengths.

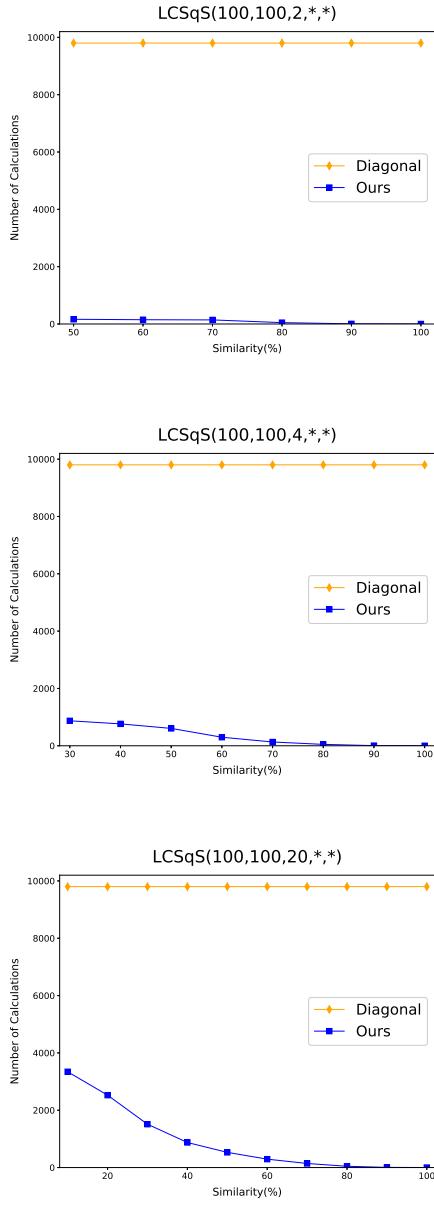


Figure 4: The computational amounts of the diagonal and our algorithms for $|A| = |B| = 100$ and various similarities.

In Figure 5, when the similarity is high, fewer cut points need to be examined, and thus less execution time is required. It shows that when the similarity is high, our algorithm is more efficient than the Inoue algorithm, and when the similarity is low, the Inoue algorithm is better.

5 Conclusion

This paper proposes an efficient algorithm to solve the longest common square subsequence (LCSqS) problem by combining the linear-space S-table [17] and the diagonal strategy for the longest common palindromic subsequence (LCPS) problem [16]. The linear-space S-table can get the 2-LCS of each pair of cut points in $\mathcal{O}(n^2)$ time, where n is the length of sequences, and it can be used to filter necessary cut points. The diagonal LCPS algorithm [16] is more efficient than the dynamic programming approach. These two algorithms help us reduce the execution time for solving the LCSqS problem.

As the experimental results show, our algorithm is more efficient for small Σ or high similarities, while the Inoue algorithm [14] is more efficient for large Σ or low similarities. According to this observation, one can adopt the suitable algorithm in different situations.

Our algorithm might be improved in the execution of the diagonal algorithm. It may be unnecessary to reconstruct the full diagonal table for each pair of cut points, while it is desired to update the diagonal table when a new character is appended progressively. If it can be improved, the time complexity of the LCSqS problem may be further reduced.

References

- [1] C. E. R. Alves, E. N. Cáceres, and S. W. Song, “An all-substrings common subsequence algorithm,” *Discrete Applied Mathematics*, Vol. 156, No. 7, pp. 1025–1035, 2008.
- [2] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, “A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings,” *Information Processing Letters*, Vol. 108, No. 6, pp. 360–364, 2008.
- [3] A. Apostolico, “Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of

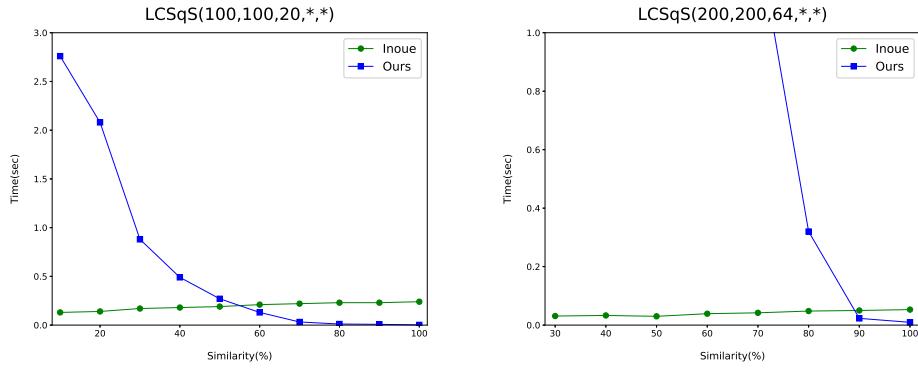


Figure 5: The execution time (in seconds) of the Inoue algorithm and ours for various similarities.

- two strings,” *Information Processing Letters*, Vol. 23, No. 2, pp. 63–69, 1986.
- [4] A. Apostolico, “String editing and longest common subsequences,” *Handbook of Formal Languages*, Vol. 2, Springer, Berlin, Heidelberg, pp. 361–398, 1997.
 - [5] A. Apostolico, S. Browne, and C. Guerra, “Fast linear-space computations of longest common subsequences,” *Theoretical Computer Science*, Vol. 92, No. 1, pp. 3–17, 1992.
 - [6] M. J. Atallah, F. Kerschbaum, and W. Du, “Secure and private sequence comparisons,” *Proceedings of 2003 ACM Workshop on Privacy in the Electronic Society*, Washington, USA, pp. 39–44, 2003.
 - [7] S. W. Bae and I. Lee, “On finding a longest common palindromic subsequence,” *Theoretical Computer Science*, Vol. 710, pp. 29–34, 2018.
 - [8] L. Bergrøth, H. Hakonen, and T. Raita, “A survey of longest common subsequence algorithms,” *Proceedings of Seventh International Symposium on String Processing and Information Retrieval*, A Coruna, Spain, pp. 39–48, 2000.
 - [9] W.-T. Chan, Y. Zhang, S. P. Y. Fung, D. Ye, and H. Zhu, “Efficient algorithms for finding a longest common increasing subsequence,” *Journal of Combinatorial Optimization*, Vol. 13, No. 3, pp. 277–288, 2007.
 - [10] S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman, “Computing a longest common palindromic subsequence,” *Fundamenta Informaticae*, Vol. 129, No. 4, pp. 329–340, 2014.
 - [11] M. Djukanovic, G. R. Raidl, and C. Blum, “A heuristic approach for solving the longest common square subsequence problem,” *Proceedings of 17th International Conference on Computer Aided Systems Theory*, Vol. 12013, Springer, pp. 429–437, 2019.
 - [12] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Communications of the ACM*, Vol. 18, No. 6, pp. 341–343, 1975.
 - [13] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng, “Efficient algorithms for finding interleaving relationship between sequences,” *Information Processing Letters*, Vol. 105, No. 5, pp. 188–193, 2008.
 - [14] T. Inoue, S. Inenaga, H. Hyryö, H. Bannai, and M. Takeda, “Computing longest common square subsequences,” *Proceedings of 9th Annual Symposium on Combinatorial Pattern Matching*, Vol. 15, Qingdao, China, pp. 1–13, 2018.
 - [15] M. Kutz, G. S. Brodal, K. Kaligosi, and I. Katriel, “Faster algorithms for computing longest common increasing subsequences,” *Journal of Discrete Algorithms*, Vol. 9, No. 4, pp. 314–325, 2011.
 - [16] T.-W. Liang, C.-B. Yang, and K.-S. Huang, “A fast algorithm for the longest common palindromic subsequence problem,” *Proceedings of 37th Workshop on Combinatorial Mathematics and Computation Theory*, Taipei, Taiwan, pp. 128–133, 2019.
 - [17] B.-S. Lin, *The Algorithms for the Linear Space S-table on the Longest Common Subsequence Problem*. Master’s Thesis, National Sun Yat-sen University, Kaohsiung, Taiwan, 2017.
 - [18] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor, “Efficient sparse

- dynamic programming for the merged LCS problem with block constraints,” *International Journal of Innovative Computing, Information and Control*, Vol. 6, No. 4, pp. 1935–1947, 2010.
- [19] C. Rick, “Simple and fast linear space computation of longest common subsequences,” *Information Processing Letters*, Vol. 75, No. 6, pp. 275–281, 2000.
 - [20] Y. Sakai, “A linear space algorithm for computing a longest common increasing subsequence,” *Information Processing Letters*, Vol. 99, No. 5, pp. 203–207, 2006.
 - [21] I. Shunsuke and H. Hyryö, “A hardness result and new algorithm for the longest common palindromic subsequence problem,” *Information Processing Letters*, Vol. 129, No. 1, pp. 11–15, 2018.
 - [22] C.-T. Tseng, C.-B. Yang, and H.-Y. Ann, “Efficient algorithms for the longest common subsequence problem with sequential substring constraints,” *Journal of Complexity*, Vol. 29, No. 1, pp. 44–52, 2013.
 - [23] K.-T. Tseng, D.-S. Chan, C.-B. Yang, and S.-F. Lo, “Efficient merged longest common subsequence algorithms for similar sequences,” *Theoretical Computer Science*, Vol. 708, pp. 75–90, 2018.
 - [24] I. H. Yang, C. P. Huang, and K. M. Chao, “A fast algorithm for computing a longest common increasing subsequence,” *Information Processing Letters*, Vol. 93, No. 5, pp. 249–253, 2005.