A Fast Algorithm for the Constrained Longest Common Subsequence Problem with Small Alphabet

Wen-Chuan Ho[a], Kuo-Si Huang[b] and Chang-Biau Yang[a*]
[a]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
*cbyang@cse.nsysu.edu.tw
[b] Department of Information Management
National Kaohsiung Marine University, Kaohsiung, Taiwan

Abstract

Given three sequences A, B and C with lengths of m, n and r, respectively, the constrained longest common subsequence *(CLCS) problem is to find the LCS of A and B which contains C as the subsequence of the answer. The dynamic programming algorithm for solving the CLCS problem, proposed by Chin et al., calculates the three-dimensional lattice layer by layer. We find that the values of most corresponding CLCS lattice cells are identical in two consecutive layers when the alphabet set is small. In this paper, we clarify whether lattice cells need to be calculated or not for reducing the computational redundancy in two consecutive layers. Accordingly, our algorithm calculates only some special boundary cells instead of the whole three-dimensional lattice in most cases, although our algorithm still requires O(mnr) time and O(mn) space in the worst case. In 2010, Deorowicz and Obstój showed that the algorithm of Chin et al. has good performance when the alphabet size $|\Sigma| \leq 20$. As our experimental results show, our algorithm is faster than Chin's algorithm when $|\Sigma| \leq 20$. So our algorithm is better than most of the previous CLCS algorithms when $|\Sigma|$ is small.*

1 Introduction

Needleman and Wunsch [15] first proposed the concept of the *longest common subsequence* (LCS) in order to deal with the relationships between biological sequences, including DNA, RNA, and protein sequences. By measuring the similarity of two sequences, there are a lot of applications in various areas, such as bioinformatics, file plagiarism, and voice recognition [13].

A *subsequence* is obtained by removing zero or more symbols from the original sequence or string. Given two sequences (or strings) $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$, where $|A| = m$ and $|B| = n$, the LCS problem is defined to find the common subsequence S in both A and B such that S is the longest one. A lot of algorithms have been proposed for solving the LCS problem [11, 14, 22]. In addition, many variations of the LCS problem have also been presented, such as the *constrained longest common subsequence* (CLCS) problem [4, 21] and the *merged longest common subsequence* (MLCS) [10, 17, 20] problem.

Given two sequences $A = a_1a_2 \cdots a_m$ and $B = b_1b_2 \cdots b_n$, and an extra constrained sequence $C = c_1c_2 \cdots c_r$, where $|A| = m$, $|B| = n$, $|C| = r$, and $m \leq n$, the *constrained longest common subsequence* (CLCS) problem, a variant LCS problem defined by Tsai [21], is to find the LCS S of A and B such that S contains C as its subsequence. Tsai proposed a *dynamic programming* (DP) algorithm in $O(m^2n^2r)$ time and space. Peng proposed a DP algorithm in $O(mnr)$ time and space for the CLCS problem [16]. Chin *et al.* showed that the CLCS problem is a special case of the *constrained sequence alignment* (CSA) problem and proposed a simpler dynamic programming algorithm with $O(mnr)$ time and space [4]. Peng and Ting [19] proposed a recursive algorithm based on the linear space algorithm with the divide-and-conquer approach [9] in $O(mnr)$ time and $O(nr)$ space.

Furthermore, Arslan and Eğecioğlu [2] proposed an improved algorithm in $O(mnr)$ time by changing its calculation order. Deorowicz [5] rewrote the formula of the dynamic programing by Chin *et al.* and employed the skill mentioned by Hunt and Szymanski [11], thus proposed an algorithm in $O(r(ml + R) + n)$ time, where $l =$

*Corresponding author.

$|LCS(A,B)|$ and R is the number of total matching pairs in A and B. Deorowicz's algorithm is more efficient when the alphabet size grows up. Iliopoulos and Rahman [12] proposed an algorithm in $O(rR \log\log n + n)$ time by using the DP formula proposed by Arslan and Eğecioğlu [2] and building the *bounded heap* (BH) data structure. Both algorithms of Deorowicz [5] and Iliopoulos & Rahman [12] only considered the matching pairs in A and B. Deorowicz and Obstój [7] proposed an algorithm based on an entry-exit point approach [8] in $O(r(ml + R) + n)$ time. Besides, they [7] implemented the existing algorithms before 2010 to compare the performance of these algorithms in practice. In 2010, Becerra *et al.* [3] proposed a space-efficient algorithm in $O(R+D)$ space, where D is the size of domination set, dependent on the length of the constrained sequence. In 2010, Deorowicz [6] also proposed an algorithm based on the *bit-parallelism* (BP) technique with a computer of word size w in $O(mnr/|\Sigma| + nr\lceil m/w \rceil)$ time under a fixed alphabet Σ.

In 2010, Peng *et al.* [18] proposed an algorithm in $O(mnr)$ time and $O(nr)$ space for a generalized version of the CLCS problem, called weighted CPSA (WCPSA) problem. In 2012, Ann *et al.* proposed an efficient algorithm for run-length encoded (RLE) sequences, in $O(mn\bar{r} + m\bar{n}r + \bar{m}nr)$ time, where $\bar{m}$, $\bar{n}$ and $\bar{r}$ are the numbers of runs in A, B and C, respectively [1].

The DP algorithm of Chin *et al.* calculates the whole two-dimensional lattices layer by layer, decomposed from the three-dimensional CLCS lattice [4]. We find that the values of most corresponding CLCS lattice cells are identical in two consecutive layers in case that the alphabet set is small. For reducing the computational redundancy in two consecutive layers, we clarify which lattices need not be calculated. Our algorithm calculates only some special boundary cells, instead of the whole 3-dimensional lattice, in most cases. Our algorithm still requires $O(mnr)$ time and $O(mn)$ space to solve the CLCS problem in the worst case. In 2010, Deorowicz and Obstój showed that the algorithm of Chin *et al.* has outstanding performance when alphabet size $|\Sigma| \leq 20$. Our experimental results show that our algorithm is faster than Chin's algorithm when $|\Sigma| \leq 20$. Therefore, our algorithm is better than most of the previous CLCS algorithms when Σ is small.

The organization of this paper is given as follows. In Section 2, the preliminaries of the CLCS problem are introduced. Section 3 presents the properties for reducing computational redundancy and then proposes our algorithm. Section 4 shows the experimental results. And finally, we derive the conclusion in Section 5.

2 The Constrained Longest Common Subsequence Problem

Let $S = s_1 s_2 \cdots s_i \cdots s_{|S|}$ be a sequence of characters over a finite alphabet Σ, where s_i and $|S|$ represent the ith character and the length of S, respectively. The notation $i..j$ indicates the index range from indexes i to j, so $S_{i..j}$ represents the substring of S from position i to position j. Note that $S_{i..j} = \emptyset$ if $i > j$.

Given two sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, and an extra constrained sequence $C = c_1 c_2 c_3 \cdots c_r$, where $|A| = m$, $|B| = n$, $|C| = r$, and $m \leq n$, the CLCS problem is to find an LCS of A and B such that C is contained in the answer as a subsequence. The answer of the problem is denoted by $CLCS(A,B,C)$. For example, assume that $A =$ `abcde` and $B =$ `acdbe` and $C =$ `ab`. The LCS of A and B is `acde`, and the $CLCS(A,B,C)$ becomes `abe` when the extra constrained sequence C is considered. Let $M(i,j,k) = |CLCS(A_{1..i}, B_{1..j}, C_{1..k})|$. Chin *et al.* [4] proposed a DP algorithm in $O(mnr)$ time and space to solve the CLCS problem as follows.

$M(i,j,k) =$

$$\begin{cases} M(i-1,j-1,k-1)+1 & \text{if } a_i = b_j \text{ and } a_i = c_k, \\ M(i-1,j-1,k)+1 & \text{if } a_i = b_j \text{ and } a_i \neq c_k, \\ \max \begin{cases} M(i-1,j,k) \\ M(i,j-1,k) \end{cases} & \text{if } a_i \neq b_j, \end{cases} \tag{1}$$

with the boundary conditions:
$M(i,0,0) = M(0,j,0) = 0$,
$M(i,0,k) = M(0,j,k) = -\infty$,
$M(i,j,0) = |LCS(A_{1..i}, B_{1..j})|$.

3 Our Constrained LCS Algorithm

By observing the three-dimensional DP lattice, it is worthy to notice that the values of most corresponding CLCS lattice cells are identical in two consecutive layers k and $k-1$. Based on this observation, the paper proposes a new algorithm for solving the CLCS problem by identifying *unchanged points*, instead of considering *match points*. The main strategy of our algorithm is to find some *boundary unchanged points* (BUP) such that each $M(i,j,k)$ in the lower right corner of those BUP's

is identical to $M(i,j,k-1)$. One cannot precisely predict the *changed region* (CR) from layer $k-1$ to layer k, because $M(i,j,k) \neq M(i,j,k-1)$ may happen when (i,j,k) is a *mismatch point* $(a_i \neq b_j)$, a *simple match point* $(a_i = b_j \neq c_k)$ or a *strong match point* $(a_i = b_j = c_k)$. Thus, some rules are proposed for identifying BUP in the following.

For easy presentation, let $(i, j_1..j_2, k)$ denote a collection of contiguous points (i, j', k) where $j_1 \leq j' \leq j_2$, that is, (i, j_1, k), (i, j_1+1, k), $\cdots$, (i, j_2-1, k), (i, j_2, k). In the following, an example is provided for explaining each term with A = `bddbcbaadbc`, B = `aacdadbdbabdadcbaadcc` and C = `cb`. The calculated lattice in layers $k=0$, $k=1$ and $k=2$ are shown in Figures 1, 2 and 3, respectively.

Definition 1. (Strong match point [5]) *For each layer $k \geq 1$, a point (i,j,k) is a* **strong match point** *if $a_i = b_j = c_k$.*

Definition 2. (Simple match point) *For each layer $k \geq 0$, a point (i,j,k) is a* **simple match point** *if $a_i = b_j \neq c_k$. Note that c_0 is set to empty.*

Definition 3. (Previous strong match point) *For each layer $k \geq 2$, a point (i,j,k) is a* **previous strong match point** *if $a_i = b_j = c_{k-1}$.*

Definition 4. (Layer start-point) *For each layer $k \geq 1$, a point (i,j,k) is a layer start-point, denoted as LS_k, if (i,j,k) is a strong match point and there is no other strong match point (i',j',k) such that $i' \leq i$ and $j' \leq j$. Specifically, we set $LS_0 = (0,0,0)$.*

For example, $LS_1 = (5,3,1)$ in Figure 2, and $LS_2 = (6,7,2)$ in Figure 3. In order to efficiently locate the next match position of a specific symbol at each position in a sequence, we build the $NextMatch$ tables of A and B. Then, we can easily locate LS_{k+1} if LS_k is known.

Definition 5. (Changed region (CR)) *A point (i,j,k)* $\in$ **changed region (CR)** *if $M(i,j,k) \neq M(i,j,k-1)$. Let $LS_{k'} = (i',j',k')$. We stipulate that $(i'', j'-1, k')$ and $(i'-1, j'', k') \in CR$, where $i'-1 \leq i'' \leq |A|$ and $j'-1 \leq j'' \leq |B|$, respectively.*

Definition 6. (Unchanged region (UR)) *A point (i,j,k)* $\in$ **unchanged region (UR)** *if $M(i,j,k) = M(i,j,k-1)$.*

Figure 2 shows examples of CR and UR.

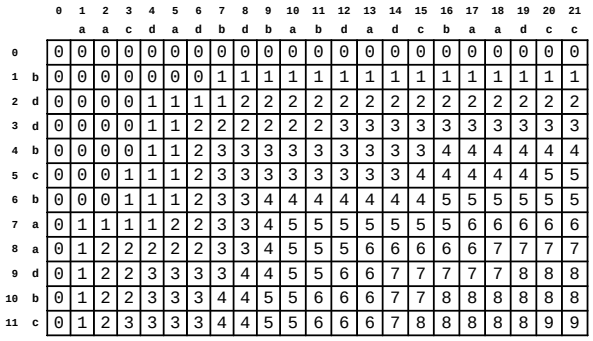

Figure 1: The CLCS lattice in layer $k=0$ for A=`bddbcbaadbc`, B=`aacdadbdbabdadcbaadcc` and C = `cb`.

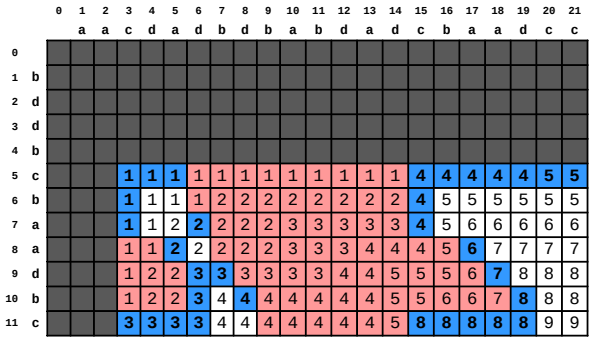

Figure 2: The CLCS lattice in layer $k=1$ for A=`bddbcbaadbc`, B=`aacdadbdbabdadcbaadcc` and C = `cb`, where the red (gray) cells form the changed regions (CR), the blue (dark) cells with bold font form the unchanged region (UR) which need to be checked in our algorithm, and the white region represents the cell in UR which needs not be checked.

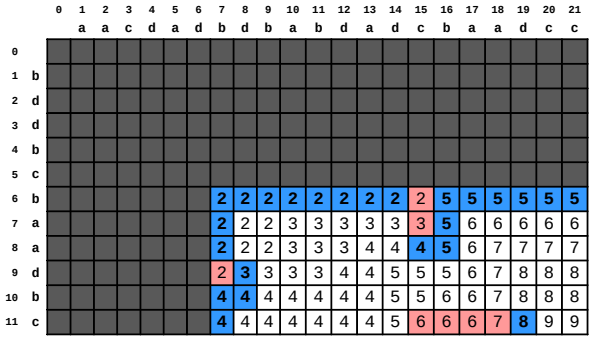

Figure 3: The CLCS lattice in layer $k=2$ for A=`bddbcbaadbc`, B=`aacdadbdbabdadcbaadcc` and C = `cb`.

Lemma 1. [5] $M(i,j,k_1) \leq M(i,j,k_2)$ *if* $k_1 \geq k_2$.

Lemma 2. $M(i-1,j-1,k) = M(i-1,j-1,k-1) = M(i-1,j-1,k-2)$ *if* $M(i,j,k) = M(i,j,k-1)$, $(i,j,k-1)$ *is a strong match point and* (i,j,k) *is a simple match point.*

Proof. By the definition of CLCS, we have $M(i,j,k) = M(i-1,j-1,k)+1$ and $M(i,j,k-1) = M(i-1,j-1,k-2)+1$. If $M(i,j,k) = M(i,j,k-1)$, we get

$$M(i-1,j-1,k) = M(i-1,j-1,k-2). \quad (2)$$

By Lemma 1, we have

$$M(i-1,j-1,k) \leq M(i-1,j-1,k-1) \leq M(i-1,j-1,k-2). \quad (3)$$

By Equations 2 and 3, we have $M(i-1,j-1,k) = M(i-1,j-1,k-1) = M(i-1,j-1,k-2)$. □

Theorem 1. *If* (i,j,k) *is not a previous strong match point,* $(i-1,j-1,k)$, $(i-1,j,k)$ *and* $(i,j-1,k) \in$ *UR, then* $(i,j,k) \in$ *UR. If* (i,j,k) *is a previous strong match point,* $(i-1,j-1,k-1)$ *and* $(i-1,j-1,k) \in$ *UR, then* $(i,j,k) \in$ *UR.*

Proof. For the first condition, we consider the following three cases.

Case 1: Suppose (i,j,k) is a mismatch point ($a_i \neq b_j$). We have $M(i,j,k-1) = \max\{M(i-1,j,k-1), M(i,j-1,k-1)\}$ and $M(i,j,k) = \max\{M(i-1,j,k), M(i,j-1,k)\}$. In addition, $(i-1,j,k)$ and $(i,j-1,k) \in$ UR, then $M(i-1,j,k-1) = M(i-1,j,k)$ and $M(i,j-1,k-1) = M(i,j-1,k)$. Thus, $M(i,j,k-1) = M(i,j,k)$, and then $(i,j,k) \in$ UR.

Case 2: Suppose (i,j,k) and $(i,j,k-1)$ are a strong match point and a simple match point, respectively ($a_i = b_j = c_k \neq c_{k-1}$). It is always true that $M(i,j,k) = M(i,j,k-1)$ because both $M(i,j,k)$ and $M(i,j,k-1)$ are equal to $M(i-1,j-1,k-1)+1$.

Case 3: Suppose $(i,j,k-1)$ and (i,j,k) are both simple match points ($c_k \neq a_i = b_j \neq c_{k-1}$). We have $M(i,j,k-1) = M(i-1,j-1,k-1)+1 = M(i-1,j-1,k)+1 = M(i,j,k)$.

For the second condition, two cases have to be discussed as follows.

Case 4: If (i,j,k) is a simple match point ($a_i = b_j = c_{k-1} \neq c_k$), we have $M(i,j,k) = M(i-1,j-1,k)+1$, and $M(i,j,k-1) = M(i-1,j-1,k-2)+1$. Thus, with $M(i-1,j-1,k) = M(i-1,j-1,k-2)$, we get $M(i,j,k) = M(i,j,k-1)$.

Case 5: If (i,j,k) is a strong match point ($a_i = b_j = c_{k-1} = c_k$), we have $M(i,j,k) = M(i-1,j-1,k-1)+1$, and $M(i,j,k-1) = M(i-1,j-1,k-2)+1$. Thus, with $M(i-1,j-1,k-1) = M(i-1,j-1,k-2)$, we get $M(i,j,k) = M(i,j,k-1)$.

□

Theorem 1 states the cases that $(i,j,k) \in$ UR and we need not compute it. The examples of Theorem 1 are shown in Table 1.

In contrast to Theorem 1, the following corollary describes those points that need to be calculated, since we do not know whether those $M(i,j,k)$'s will be changed or not.

Corollary 1. *We need to calculate* $M(i,j,k)$ *if one of the following two conditions is satisfied.*

1. $(i-1,j-1,k)$, $(i-1,j,k)$ *or* $(i,j-1,k) \in$ *CR when* (i,j,k) *is not a previous strong match point.*
2. $(i-1,j-1,k-1)$ *or* $(i-1,j-1,k) \in$ *CR when* (i,j,k) *is a previous strong match point.*

For easy implementation of our algorithm, we further simplify the above corollary as follows.

Corollary 2. $M(i,j,k)$ *needs to be calculated if one of the following two conditions is satisfied.*

1. $(i-1,j-1,k)$, $(i-1,j,k)$ *or* $(i,j-1,k) \in$ *CR.*
2. $(i-1,j-1,k-1) \in$ *CR when* (i,j,k) *is a previous strong match point.*

Based on Corollary 2, we propose our algorithm for calculating the CLCS length in Algorithm 1. The initial L of size mn is constructed from the traditional LCS problem for layer $k = 0$. Since some positions may have the same value between two consecutive layers, L is used to repeatedly record the CLCS length of (i,j,k) in each layer k. Whenever $L[i][j]$ is to be changed on layer k, we record the old value for layer $k-1$ in $L'[i][j]$. For example, when the computation is performed on layer $k = 1$, $M(4,2,1) = -\infty \neq M(4,2,0) = 0$, so $L'[4][2]$ records 0 and $L[4][2]$ is updated to $-\infty$. $M(i,j,k)$ is usually calculated from L, but $M(i,j,k)$ is calculated from L' when (i,j,k) is a strong match point and $(i-1,j-1,k) \in$ CR, since $M(i-1,j-1,k-1)$ is now stored in $L'[i][j]$.

CR_i records all positions j of the changed region in each current row i, while CR_i is utilized in the next row $i+1$. $PCR_k[i]$ records all positions $j+1$ of the changed region if $(i+1,j+1,k+1)$ is a previous strong match point ($a_{i+1} = b_{j+1} = c_k$) in each row i of layer k. Note that $PCR_k[i]$ is utilized in position $j+1$ of row $i+1$ in layer $k+1$, so its position value is $j+1$, not j.

Lines 7 and 8 initialize the boundary cells of L in layer k to $-\infty$ and line 9 initializes the changed region of the starting row in layer k. $Len = M(i,j,k)$ is calculated by Equation 1 from L or L' (line 13) and compared with $L[i][j]$. If $(i,j,k) \in$

Table 1: The examples of Theorem 1 for A=bddbcbaadbc, B=aacdadbdbabdadcbaadcc and C = cb in Figures 2 and 3.

Case	(i, j, k)∈UR	Type	UR for making decision
Case 1	(7, 8, 2)	$a_i \neq b_j$	(6, 7, 2), (6, 8, 2), (7, 7, 2)
Case 2	(10, 16, 2)	$a_i = b_j = c_k \neq c_{k-1}$	(9, 15, 2), (9, 16, 2), (10, 15, 2)
Case 3	(7, 13, 2)	$c_k \neq a_i = b_j \neq c_{k-1}$	(6, 12, 2), (6, 13, 2), (7, 12, 2)
Case 4	(11, 20, 2)	$a_i = b_j = c_{k-1} \neq c_k$	(10, 19, 2), (10, 19, 1)

Algorithm 1 Computing the length of CLCS.

Input: $A = a_1a_2\cdots a_m$, $B = b_1b_2\cdots b_n$ and $C = c_1c_2\cdots c_r$
Output: Length of **CLCS**(A, B, C)

1: Construct arrays $NextMatch_A$ and $NextMatch_B$
2: Construct lattice L of LCS(A, B)
3: **for** $k := 1 \rightarrow r$ **do**
4: $LS_k \leftarrow (NextMatch_A[c_k][LS_{k-1}.i],$ $NextMatch_B[c_k][LS_{k-1}.j])$
5: **if** LS_k is invalid **then**
6: **return** $-\infty$
7: $L[LS_k.i-1][LS_k.j-1..n] \leftarrow -\infty$
8: $L[LS_k.i..m][LS_k.j-1] \leftarrow -\infty$
9: $CR_{LS_k.i-1} \leftarrow \{LS_k.j, LS_k.j+1, \cdots, n\}$
10: **for** $i := LS_k.i \rightarrow m$ **do**
11: $j \leftarrow LS_k.j$ // Starting column of layer k
12: **while** not end of CR_{i-1} **and** not end of $PCR_{k-1}[i-1]$ **do**
13: $Len \leftarrow$ Calculate $M(i, j, k)$ from L and L' based on a_i, b_j and c_k
14: **if** $L[i][j] \neq Len$ **then** //$(i, j, k) \in$ CR
15: $L'[i][j] \leftarrow L[i][j]$
16: $L[i][j] \leftarrow Len$
17: Append j into CR_i
18: **if** $a_{i+1} = b_{j+1} = c_k$ **then**
19: Append $j+1$ into $PCR_k[i]$
20: j++ // Continue to right position
21: **else** // $(i, j, k) \in$ UR
22: $j \leftarrow$ min{next available positions of CR_{i-1} and $PCR_{k-1}[i-1]$}
23: **return** $L[m][n]$

CR (lines 14-19), $L[i][j]$ needs to be updated. Position j is appended into CR_i. If $(i+1, j+1, k+1)$ is a previous strong match point, position $j+1$ is appended into $PCR_k[i]$ for layer $k+1$. Then, $(i, j+1, k)$ is calculated continuously. If $(i, j, k) \in$ UR, then we go to the next available position of CR_{i-1} and $PCR_{k-1}[i-1]$ (lines 20-21).

Take A=cddbdbdbdbc, B=dbacbdcbbcbbcb and C=cb as an example. Figures 1, 2 and 3 show the content of lattices L in layers 0, 1 and 2, respectively.

In the first iteration ($k = 0$), we initialize $M(i, j, 0) = L[i][j]$, so the content of lattice L' is empty. The detailed contents of CR, UR and checked points in layers $k = 1$ and $k = 2$ are shown in Table 2. A checked point (i, j, k) needs to be calculated in our algorithm, but (i, j, k) may be in UR or CR. In the second iteration ($k = 1$), the layer start-point (5, 3, 1) is obtained, so L[4][2..21] and L[5..11][2] are copied to L'[4][2..21] and L'[5..11][2], respectively, and then L[4][2..21] and L[5..11][2] are set to $-\infty$.

Previous strong match points never occur when $k \leq 1$, so $PCR_0[i]$ are empty in each row i. Before the starting row ($i = 5$) of layer 1, all positions from 3 to 21 are put into CR_4. After calculating, the changed positions of row 5 are recorded in CR_5, which are 6 through 14. In the following rows, we calculate the position from starting column ($j = 3$). If the point $(i, j, 1) \in$ CR, then we calculate $(i, j+1, 1)$ continuously. Otherwise, we go to the next available position of CR_{i-1} and $PCR_0[i-1]$. It is worth noting that position 15 is put into $PCR_1[10]$ in row 10 because (10, 14, 1) ∈ CR and (11, 15, 2) is a previous strong match point ($a_{11} = b_{15} = c_1 =$ c).

It is obvious that Algorithm 1 requires only O(mn) space, since L and L' are reused in every layer, and PCR can be reused every two layers alternately. In the worst case, we have to calculate the whole lattice in each layer when all points $(i, j, k) \in$ CR. So Algorithm 1 requires O(mnr) time.

4 Experimental Results

This section shows the comparisons of time efficiency between our CLCS algorithm and the algorithm of Chin *et al.* [4]. The testing environment is a computer running 64-bit Windows 7 with 3.40GHZ CPU (Intel Core i7-2600) and 4GB RAM. The algorithms are implemented by Microsoft Visual C++ 2015.

The published algorithms can be classified roughly into the match-point based algorithm and the DP based algorithm. The match-point based algorithms only compute the positions of match points, which are not suitable for small $|\Sigma|$, such as

Table 2: The detailed contents of changed region (CR), unchanged region (UR) and checked points for each row in layers $k = 1$ and $k = 2$. A checked point (i, j, k) needs to be calculated in our algorithm, but (i, j, k) may be in UR or CR. Symbol $-$ denotes that no point belongs to the type in the row.

k	i	CR_i	$(i,j,k) \in$ CR	$PCR_k[i]$	Checked point (i,j,k)	$(i,j,k) \in$ UR
1	4	3..21	(4, 3..21, 1)	–	–	–
1	5	6..14	(5, 6..14, 1)	–	(5, 3..5, 1), (5, 15..21, 1)	–
1	6	6..14	(6, 6..14, 1)	–	(6, 3, 1), (6, 15, 1)	(6, 4..5, 1), (6, 16..21, 1)
1	7	7..14	(7, 7..14, 1)	–	(7, 3, 1), (7, 6, 1), (7, 15, 1)	(7, 4..5, 1), (7, 16..21, 1)
1	8	3..4, 7..16	(8, 3..4, 1), (8, 7..16, 1)	–	(8, 5, 1), (8, 17, 1)	(8, 6, 1), (8, 18..21, 1)
1	9	3..5, 8..17	(9, 3..5, 1), (9, 8..17, 1)	–	(9, 6..7, 1), (9, 18, 1)	(9, 19..21, 1)
1	10	3..5, 9..18	(10, 3..5, 1), (10, 9..18, 1)	15	(10, 6, 1), (10, 8, 1), (10, 19, 1)	(10, 7, 1), (10, 20..21, 1)
1	11	9..14	(11, 9..14, 1)	–	(11, 3..6, 1), (11, 15..19, 1)	(11, 7..8, 1), (11, 20..21, 1)
2	5	7..21	(5, 7..21, 1)	–	–	–
2	6	15	(6, 15, 2)	–	(6, 7..14, 2), (6, 16..21, 2)	–
2	7	15	(7, 15, 2)	–	(7, 7, 2), (7, 16, 2)	(7, 8..14, 2), (7, 17..21, 2)
2	8	–	–	–	(8, 7, 2), (8, 15..16, 2)	(8, 8..14, 2), (8, 17..21, 2)
2	9	7	(9, 7, 2)	–	(9, 8, 2)	(9, 9..21, 2)
2	10	–	–	–	(10, 7..8, 2)	(10, 9..21, 2)
2	11	15..18	(11, 15..18, 2)	–	(11, 7, 2), (11, 19, 2)	(11, 8..14, 2), (11, 20..21, 2)

the algorithms proposed by Deorowicz [5], Iliopoulos and Rahman [12] and Becerra *et al.* [3]. These algorithms become more efficient when $|\Sigma|$ grows up, due to decreasing match points. The DP based algorithms include Chin *et al.* [4], Peng and Ting [19], and Arslan and Eğecioğlu [2]. In DP based algorithms, they compute the whole 3-dimensional lattice and do not consider additional characteristics of input sequences, so the required time depends on only the lengths of input sequences, not affected by $|\Sigma|$. On the other hand, our algorithm is more efficient algorithm for sequences with high similarity or small alphabet. The *similarity* of two sequences A and B is defined as follows.

$$similarity(A, B) = \frac{LCS(A, B)}{\min\{|A|, |B|\}}. \quad (4)$$

In the following experiments, we focus on some influence factors of our algorithm and the algorithm of Chin *et al.* [4], including sequence similarities, sequence lengths and alphabet sizes. Each experiment is performed 100 times. For consistent presentation of our algorithm, we define the 6-tuple parameter, $T(|A|, |B|, |C|, |\Sigma|, algo, similarity)$, where $|C| \in \{2, 4, 8, 16\}$, $|\Sigma| \in \{2, 4, 20, 64, 256, 1000\}$, and *algo* is the implemented algorithm. The *similarity* is defined in Equation 4 and symbol R means that the sequences are generated randomly.

The first experiment studies the influence of various similarities on our algorithm, where $|A| = |B| = 1000$, $|C| \in \{2, 4, 8, 16\}$ and $|\Sigma| \in \{4, 20\}$. Figures 4 and 5 show that the execution time decreases when the similarity increases, and the downtrend is steeper when $|C|$ and $|\Sigma|$ increase. In summary, our algorithm has better efficiency when the input sequences are similar.

The second experiment discusses the influence of various alphabet sizes, where the sequences are randomly generated. The execution time comparison with various alphabet sizes between our algorithm and the algorithm of Chin *et al.* [4] is shown in Figure 6. Because Chin's algorithm is not affected by $|\Sigma|$ obviously, we just take Chin's algorithm with $|C| = 4$ and 16 as benchmarks. According to the experimental results, the algorithm by Chin *et al.* is stable for different sizes of Σ, and the required time of our algorithm usually increases when $|\Sigma|$ becomes large. For the middle size of $|\Sigma|$, such as $|\Sigma| = 64$ or 256, the match points decrease, and then the changed region is enlarged. Thus, the required time increases. However, it is

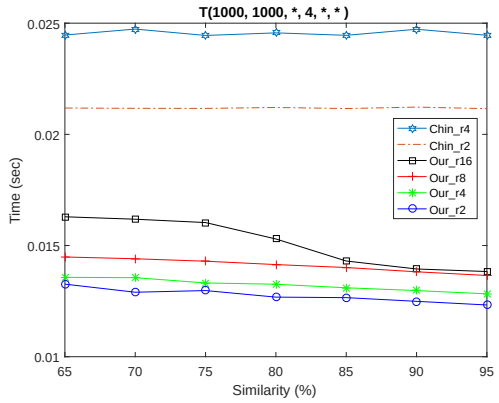

Figure 4: The execution time comparison (in seconds) of various *similarity* with $|A| = |B| = 1000$, $|\Sigma| = 4$ and $|C| \in \{2, 4, 8, 16\}$. `Our_r2` and `Chin_r2` denote ours and Chin's algorithm, respectively, with $|C| = 2$.

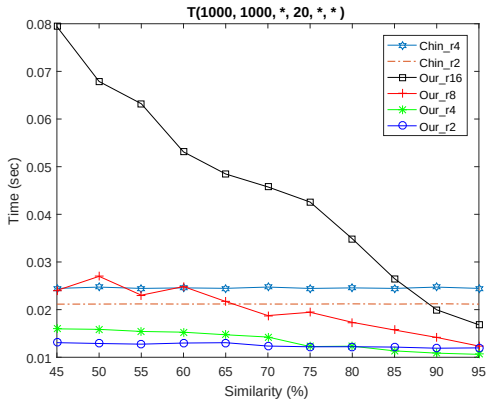

Figure 5: The execution time comparison (in seconds) of various *similarity* with $|A| = |B| = 1000$, $|\Sigma| = 20$ and $|C| \in \{2, 4, 8, 16\}$. `Our_r2` and `Chin_r2` denote ours and Chin's algorithm, respectively, with $|C| = 2$.

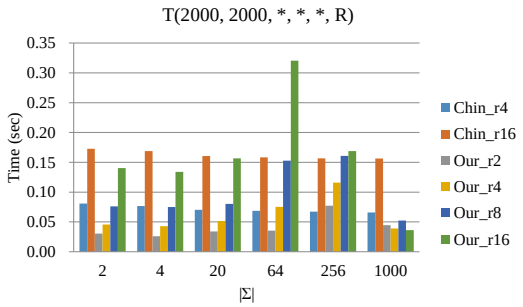

Figure 6: The execution time comparison (in seconds) of various $|\Sigma|$ with $|A| = |B| = 2000$ and $|C| \in \{2, 4, 8, 16\}$. `Our_r4` and `Chin_r4` denote ours and Chin's algorithm, respectively, with $|C| = 4$.

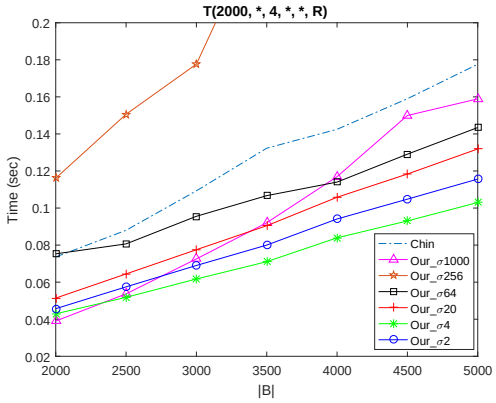

Figure 7: The execution time comparison (in seconds) of various $|\Sigma|$ with $|A| = 2000$, $|B| \in \{2000, 2500, 3000, 3500, 4000, 4500, 5000\}$ and $|C| = 4$. `Chin` denotes Chin's algorithm, and `Our_`σ`20` denotes ours with $|\Sigma| = 20$.

hard to find feasible solutions for large $|C|$ and very big $|\Sigma|$, such as $|\Sigma| = 1000$. In this situation, the algorithm usually terminates for layer with larger k. So the execution time decreases suddenly with $|C| = 16$ and $|\Sigma| = 1000$.

The third experiment tries to fix the length of $A = 2000$ with $|C| \in \{4, 16\}$ to observe the trend of execution time between $|\Sigma| \in \{2, 4, 20, 64, 256, 1000\}$ and $|B| \in \{2000, 2500, 3000, 3500, 4000, 4500, 5000\}$. The experimental results are shown in Figures 7 and 8. The similarity is thought to increase because the CLCS length increases with longer $|B|$ by Equation 4. However, the similarity increase is small when $|B|$ is enlarged, since the sequences are generated randomly. Thus, the slope of execution time is almost same with different $|B|$.

Then, in the fourth experiment, it is worthy to discuss the influence of $|C|$ with fixed $|\Sigma| = 4$ and $|\Sigma| = 20$ in our algorithm, whose experimental results are shown in Figures 9 and 10. We can see that the growth trend of execution time becomes steeper while $|C|$ increases because large $|C|$ leads to large changed region.

The match-point based algorithms are efficient in big alphabet size because of decreased match points, but the efficiency is worse than DP based algorithms when alphabet size is small. According to the experimental results, our algorithm has better efficiency than DP based algorithms with a small alphabet size or highly similar sequences.

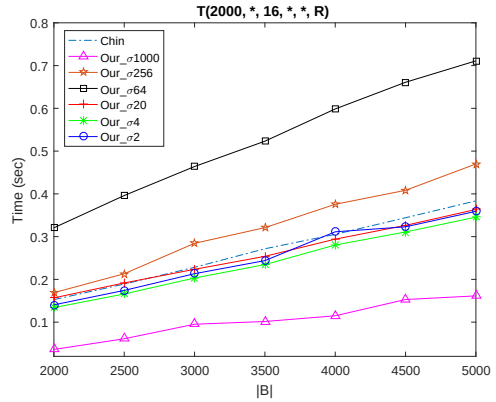

Figure 8: The execution time comparison (in seconds) of various $|\Sigma|$ with $|A| = 2000$, $|B| \in \{2000, 2500, 3000, 3500, 4000, 4500, 5000\}$ and $|C| = 16$.

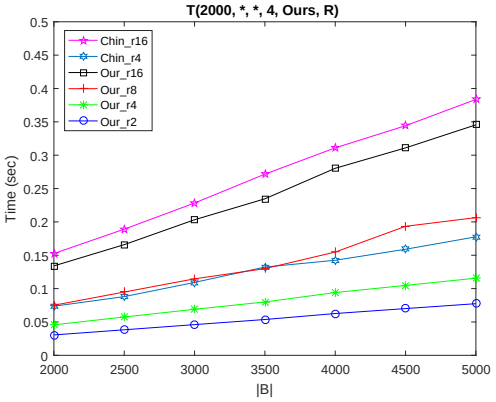

Figure 9: The execution time (in seconds) of various $|C|$ with $|A| = 2000$, $|\Sigma| = 4$ and $|B| \in \{2000, 2500, 3000, 3500, 4000, 4500 , 5000\}$. `Our_r2` denotes our algorithm with $|C| = 2$.

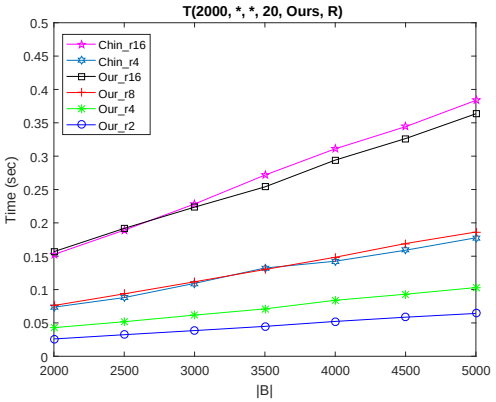

Figure 10: The execution time (in seconds) of various $|C|$ with $|A| = 2000$, $|\Sigma| = 20$ and $|B| \in \{2000, 2500, 3000, 3500, 4000, 4500, 5000\}$.

5 Conclusion

In this paper, we propose an efficient algorithm for solving the CLCS problem in $O(mnr)$ time and $O(mn)$ space, where m, n, and r are the lengths of A, B, and C, respectively. We observed that the values of most CLCS lattice cells are identical in two consecutive layers k and $k-1$, so the values in layer $k-1$ can be used directly in layer k. That is, the calculation of most lattice cells can be omitted. According to the experiment described by Deorowicz and Obstój [7], Chin *et al.* is the fastest algorithm when $|\Sigma| \leq 5$. Besides, the performance of Chin's algorithm is better than most of algorithms. By the experiment results, we observe that the execution time of our algorithm is about 25% to 50% of Chin *et al.* [4] when $|\Sigma| \leq 20$. So, our algorithm has better performance than most of algorithms when $|\Sigma|$ is small. However, our algorithm requires more time than Chin's algorithm when $|\Sigma|$ is large. Besides, the influence of alphabet size is larger than sequence similarity in our algorithm.

In our algorithm, the execution time increases when $|\Sigma|$ increases because fewer match points may result in larger changed region. Although the match points decrease when $|\Sigma|$ increases, there is no evident relationship between match points and alphabet size. So, studying the relationship between match points and alphabet size is worthy in the future. In addition, we still have to maintain the information of mismatch points in our algorithm. It reduces the execution efficiency. It is worth to apply more suitable data structures on the calculations of match points, thus the algorithm may be further improved in the future.

Acknowledgement

This research work was partially supported by the National Science Council of Taiwan under contract MOST 104-2221-E-110-018-MY3.

References

[1] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "Fast algorithms for computing the constrained LCS of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, 2012.

[2] A. N. Arslan and O. Ömer Eğecioğlu, "Algorithms for the constrained longest common

subsequence problems," *International Journal of Foundations of Computer Science*, Vol. 16, No. 6, pp. 1099–1109, 2005.

[3] D. Becerra, W. Soto, L. Nino, and Y. Pinzón, "An algorithm for constrained LCS," *ACS/IEEE International Conference on Computer Systems and Applications, Hammamet, Tunisia*, pp. 237–246, May, 2010.

[4] F. Y. L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, and S. K. Kim, "A simple algorithm for the constrained sequence problems," *Information Processing Letters*, Vol. 90(4), pp. 175–179, 2004.

[5] S. Deorowicz, "Fast algorithm for constrained longest common subsequence problem.," *Theoretical and Applied Informatics*, Vol. 19, No. 2, pp. 91–102, 2007.

[6] S. Deorowicz, "Bit-parallel algorithm for the constrained longest common subsequence problem," *Fundamenta Informaticae*, Vol. 99, pp. 409–433, 2010.

[7] S. Deorowicz and J. Obstój, "Constrained longest common subsequence computing algorithms in practice," *Computing and Informatics*, Vol. 29, pp. 427–445, 2010.

[8] D. He and A. N. Arslan, "A space-efficient algorithm for the constrained pairwise sequence alignment problem," *Genome Informatics*, Vol. 16, No. 2, pp. 237–246, 2005.

[9] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, No. 6, pp. 341–343, 1975.

[10] 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, pp. 188–193, 2008.

[11] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[12] C. S. Iliopoulos and M. S. Rahman, "New efficient algorithms for the LCS and constrained LCS problems," *Information Processing Letters*, Vol. 106, No. 1, pp. 13–18, 2008.

[13] J. B. Kruskal, "An overview of sequence comparison: the warps, string edits, and macromolecules," *SIAM Review*, Vol. 25, No. 2, pp. 201–237, 1977.

[14] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar text strings," *Acta Informatica*, Vol. 18, pp. 171–179, 1982.

[15] D. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, Vol. 48, pp. 443–453, 1970.

[16] C.-L. Peng, "An approach for solving the constrained longest common subsequence problem," Master's Thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan, 2003.

[17] 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, pp. 1935–1947, 2010.

[18] Y.-H. Peng, C.-B. Yang, K.-S. Huang, and K.-T. Tseng, "An algorithm and applications to sequence alignment with weighted constraints," *International Journal of Foundations of Computer Science*, Vol. 21, pp. 51–59, 2010.

[19] Z. S. Peng and H. F. Ting, "Time and space efficient algorithms for constrained sequence alignment," *Implementation and Application of Automata. CIAA 2004. Lecture Notes in Computer Science*, Vol. 3317, pp. 237–246, 2005.

[20] A. M. Rahman and M. S. Rahman, "Effective sparse dynamic programming algorithms for merged and block merged LCS problems," *Journal of Computers*, Vol. 9, No. 8, pp. 1743–1754, 2014.

[21] Y. T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, Vol. 88, pp. 173–176, 2003.

[22] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.