Efficient Algorithms for the Flexible Longest Common Subsequence Problem *

Yi-Pu Guo[a], Yung-Hsing Peng[b] and Chang-Biau Yang[a†]

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

[b]Innovative DigiTech-Enabled Applications & Services Institute
Institute for Information Industry, Kaohsiung, Taiwan

Abstract

Given two sequences, the traditional longest common subsequence (LCS) problem is to obtain the common subsequence with the maximum number of matches, without considering the continuity of the matched characters. However, in many applications, the matching results with higher continuity are more meaningful than the sparse ones, even if the number of matched characters is a little lower. Accordingly, we define a new variant of the LCS problem, called the flexible longest common subsequence (FLCS) problem. In this paper, we design a scoring function to estimate the continuity of a matching result between two strings. We show that the optimal solution of FLCS can be determined in $O(n^2)$ time, where n denotes the longer length of the two input sequences. Therefore, the results in this paper offer a new efficient tool for sequence analysis.

1 Introduction

The topic of evaluating how similar one string or one sequence to another is, such as the *longest common subsequence* [3, 4, 7–10, 15, 17] or the *edit distance* [2, 14], has been studied for several decades. Based on these algorithms, methods for solving verification [6], classification [1, 11], and clustering [13] problems are further developed.

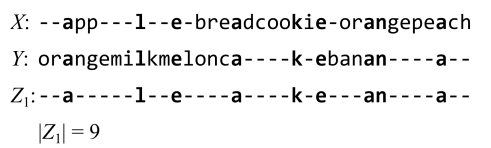
Given two sequences X and Y, the *longest common subsequence* (LCS) problem asks for the longest sequence that is a subsequence of both X and Y. Meanwhile, the *edit distance* problem is to find the minimum number of operations (to insert, delete or substitute one character each time) required for modifying one string to another. However, to our knowledge, the continuity of matched characters has not yet been emphasized in previous methods. In Figure 1, there are two alignment results Z_1 and Z_2 of string $X =$ `applebreadcookieorangepeach` and string $Y =$ `orangemilkmeloncakebanana`, where Z_1 has more matched characters (which is 9) than Z_2 (which is 8) does. However, Z_1 does not seem to be quite meaningful because of its scattered matches.

In bioinformatics, longer continuous matches in amino acid sequences mean that these sequence segments are likely to have the same function. For plagiary detection, long and continuous matches infer that many sentences were copied from one article to the other. Therefore, it is interesting and necessary to devise a comparison scheme that considers the continuity of matches. For this purpose, we propose the *flexible longest common subsequence* (FLCS) problem.

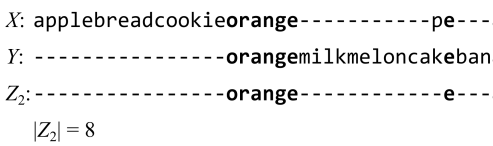
The organization of this paper is as follows. In Section 2, we give a formal definition to the FLCS problem, and propose a straightforward algorithm with $O(n^3)$ time. In Section 3, we further propose improved $O(n^2 \times min\{r \log r, n\})$-time, $O(n^2 r)$-time, and $O(n^2)$-time algorithms, where r denotes the size of the dominant list in our algorithm, which is usually much less than n. Finally, the conclusions are given in Section 4.

*This research was partially supported by the National Science Council of Taiwan under contract NSC 102-2221-E-110-033-MY2. This research was also partially supported by the "Online and Offline Integrated Smart Commerce Platform (1/4)" of the Institute for Information Industry, which is subsidized by the Ministry of Economy Affairs of Taiwan.

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



(a)



(b)

Figure 1: Two alignment results with different continuity.

2 The Flexible Longest Common Subsequence Problem

2.1 Problem Definition

For a given sequence of elements $S = s_1 s_2 s_3 \dots s_{|S|}$, where $|S|$ represents the length of S, we use s_i to denote the ith element of S. Also, let $S_{i..j} = s_i s_{i+1} s_{i+2} \dots s_j$ denote a substring from the ith element to the jth element of S. For two given sequences X and Y, a pair of two characters from X and Y is defined as follows.

Definition 1. *A pair (i, j) is called a* matched pair *iff $x_i = y_j$. Otherwise, (i, j) is a* mismatched pair*. A character x_i (y_j) can form a mismatched pair with a gap, represented as (i, ∞) or (∞, j), where x_∞ and y_∞ denote symbols that does not exist in X and Y.*

Thus, an alignment result (such as Z_1 in Figure 1) consists of a sequence P of pairs. However, for ease of reading, the alignment results are represented in characters, rather than pairs of indices.

Definition 2. *Given an alignment result $P = p_1 p_2 p_3 \dots p_{|P|}$, a matched segment of P is a substring $p_i p_{i+1} p_{i+2} \dots p_j$ of P that contains only matched pairs. Similarly, a mismatched segment of P is a substring of P that contains only mismatched pairs.*

Suppose there are $|Z|$ segments $z_1 z_2 z_3 \dots z_{|Z|}$ in P. Let Z_M be the set of matched segments. To emphasize continuous matches, each matched segment z_i in P is measured by a polynomial function $f(x) = x^\gamma$ referring to the length of z_i, where γ is a constant. That is, the score of the whole alignment is the summation of the scores of all matched segments, denoted as $F(P)$, as follows.

$$F(P) = \sum_{z_i \in Z_M} |z_i|^\gamma. \quad (1)$$

In the above equation, γ is a user-specified parameter. For clarity, the length of each segment is supposed to be locally maximized. That is, for each matched (mismatched) segment $p_i p_{i+1} p_{i+2} \dots p_j$, the pairs p_{i-1} and p_{j+1} can only be mismatched (matched) pairs.

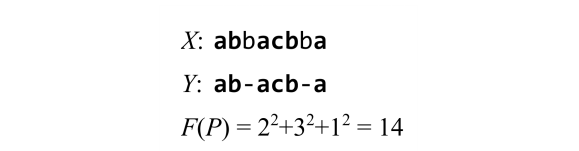


Figure 2: An example for computing the scoring function $F(P)$.

Figure 2 shows a scoring example for sequences $X =$ `abbacbba` and $Y =$ `abacba`. There are three matched segments, which are $z_1 =$ `ab`, $z_3 =$ `acb`, and $z_5 =$ `a`. The lengths of z_1, z_3 and z_5 are 1, 3 and 2, respectively. Therefore, when $\gamma = 2$, we have $F(P) = 1^2 + 3^2 + 2^2 = 14$. By adjusting the parameter γ in the scoring function, we can control the degree of emphasis on continuity. The higher γ is, the higher score the function gains on continuous matches. If $\gamma = 1$, the obtained score of this function is the same as the length of the traditional LCS. The formal definition of the *flexible longest common subsequence* (FLCS) problem is given as follows.

Definition 3. The Flexible Longest Common Subsequence Problem: *Given two sequences $X = x_1 x_2 x_3 \dots x_m$, $Y = y_1 y_2 y_3 \dots y_n$, and a scoring function $F(P)$ with γ as the continuity coefficient, the flexible longest common subsequence problem is to obtain the sequence P of matching pairs for X and Y which corresponds to a common subsequence of X and Y with maximal $F(P)$.*

2.2 The Straightforward Dynamic Programming

In this section, we first propose a simple algorithm with $O(n^3)$ time. Let $L(X_i, Y_j)$ denote the optimal score of the FLCS between prefixes

$X_{1..i}$ and $Y_{1..j}$. The straightforward dynamic programming for computing $L(X_i, Y_j)$ is presented in Equation 2.

$$L(X_i, Y_j) = \max \begin{cases} L(X_{i-1}, Y_j) \\ L(X_i, Y_{j-1}) \\ \{L(X_{i-c}, Y_{j-c}) + c^\gamma \mid c \in \mathbb{N} \\ \quad \text{and } X_{(i-c+1)..i} = Y_{(j-c+1)..j}\} \end{cases} \quad (2)$$

The correctness of the above algorithm can be easily verified, since it examines each possible length for the last matched segment that ends with $x_i = y_j$. Clearly, the time complexity of the algorithm is $O(n^3)$, where it is assumed that $m \leq n$.

3 Improved Algorithms for the FLCS problem

In this section, we first propose an algorithm with the dominant strategy, which solves the FLCS problem in $O(n^2 \times min\{r \log r, n\})$ time, where r denotes the maximal size of dominant lists. After that, we improve the algorithm by predicting the dominance, which gives an $O(n^2 r)$-time algorithm. Finally, we show that the computation time can be further reduced to $O(n^2)$.

3.1 The Algorithm with the Dominant Strategy

To reduce the time spent on checking the length, the concept of *dominant points* [5] is applied. If there is always one solution better than another, we say that this one *dominates* another, and the dominated solutions can be removed from the dominant list. Algorithm 1 is a general implementation for solving the FLCS with dominant lists, whose notations are explained as follows. We use $D_{i,j} = \{e_1, e_2, e_3, \ldots, e_{|D_{i,j}|}\}$ to denote the dominant list that preserves the candidate solutions for $L(X_i, Y_j)$ in lattice cell (i, j). Each candidate solution is represented by a vector $e_k = (s_k, v_k)$, where s_k and v_k record its score and the length of the last matched segment ending at $x_i = y_j$, respectively.

By updating these values, one solution can be extended from one list to the next. When $x_i = y_j$, the function EX_1 extends each solution $e_k = (s_k, v_k) \in D_{i-1,j-1}$ to $e' = (s', v') \in D_{i,j}$, as shown in Equation 3.

$$\text{EX}_1(e_k) = (s_k - v_k^\gamma + (v_k + 1)^\gamma, v_k + 1). \quad (3)$$

On the other hand, when $x_i \neq y_j$, the function EX_2 extends each solution $e_k = (s_k, v_k) \in D_{i-1,j} \cup D_{i,j-1}$ to $e' = (s', v') \in D_{i,j}$, as follows.

$$\text{EX}_2(e_k) = (s_k, 0). \quad (4)$$

For $\gamma \geq 1$, we say that $e_{k_1} = (s_{k_1}, v_{k_1})$ *dominates* $e_{k_2} = (s_{k_2}, v_{k_2})$ if $s_{k_1} \geq s_{k_2}$ and $v_{k_1} \geq v_{k_2}$. That is, no matter how these two solutions are extended, the score of the extended solution $e'_{k_1} = (s'_{k_1}, v'_{k_1})$ will not be worse than the extended solution $e'_{k_2} = (s'_{k_2}, v'_{k_2})$. Once a solution is dominated by any other solution, it can be removed from the dominant list. In Algorithm 1, the function $\text{DOM}(E)$ removes all dominated solutions in the given list E. By applying the algorithm for finding 2D maximal on bounded integer points [15], it takes $O(n)$ time for computing each $D_{i,j}$ because $0 \leq v_k \leq n$. Therefore, the time complexity of this algorithm is bounded by $O(n^3)$, or $O(n^2 r \log r)$, where the maximal size of $D_{i,j}$ is assumed to be bounded by r. By our observation, r is usually much less than n.

3.2 The Algorithm with Dominance Prediction

Based on Equation 3, we can predict the score $f_k(t)$ of e_k for the next t extensions with EX_1, which can be written as Equation 5.

$$f_k(t) = s_k - v_k^\gamma + (v_k + t)^\gamma \quad (5)$$

For two solutions e_{k_1} and e_{k_2}, the intersection point t of $f_{k_1}(t)$ and $f_{k_2}(t)$ can be computed by Equation 6. By computing such t, which we call the *survival time*, the timing for the future dominance of e_{k_1} and e_{k_2} can be predicted.

$$f_{k_1}(t) = f_{k_2}(t), \\ s_{k_1} - v_{k_1}^\gamma + (v_{k_1} + t)^\gamma = s_{k_2} - v_{k_2}^\gamma + (v_{k_2} + t)^\gamma \quad (6)$$

For conciseness, in this paper we consider the special case with $\gamma = 2$, whose intersection point can be calculated by the formula shown in Equation 7.

$$t = \frac{s_{k_1} - s_{k_2}}{2(v_{k_2} - v_{k_1})}. \quad (7)$$

For other positive integer $\gamma = 3$, 4, or 5, one can refer to Galois theory [16]. For other values of γ, we have to approximate the integer-precise answer by numerical analysis, like the Newton-Raphson

Algorithm 1 The dynamic programming algorithm with the dominant concept.

$D_{i,0} = D_{0,j} =$ the list of the initial solution.
for $i = 1 \to m$ **do**
  for $j = 1 \to n$ **do**
    if $x_i = y_j$ **then**
      $E = \text{EX}_1(D_{i-1,j-1}) \cup \text{EX}_2(D_{i-1,j}) \cup \text{EX}_2(D_{i,j-1})$
    else
      $E = \text{EX}_2(D_{i-1,j}) \cup \text{EX}_2(D_{i,j-1})$
    end if
    $D_{i,j} = \text{DOM}(E)$
  end for
end for

method [12], which can be achieved in a few iterations for nonnegative integer t.

Note that only the nonnegative t is meaningful. If the obtained t is negative, it means that solutions e_{k_1} and e_{k_2} cannot dominate each other in the future.

From hereafter, we include an additional survival time t_k in the vector of each solution, i.e., $e_k = (s_k, v_k, t_k)$, where t_k is a nonnegative integer. The survival time t_k means that e_k is alive from time 0 (now) to time t_k, and it will be expired (dominated by some other solutions) on time $t_k + 1$, where the current $D_{i,j}$ is marked as time 0. During the process, we order the dominant list decreasingly by s_k, thus the first solution $e_1 \in D_{i,j}$, denoted as $D_{i,j,1}$, is always with the highest score. For $D_{i,j,1}$, we use $t_{i,j,1}$ to denote its survival time. The function for computing the survival time for two solutions e_{k_1} and e_{k_2} is denoted as $\text{INTERSECT}(e_{k_1}, e_{k_2})$, which is assumed to take constant time.

The extending functions with survival time are denoted as EXT_1 and EXT_2. When $x_i = y_j$, EXT_1 extends one solution $e_k = (s_k, v_k, t_k) \in D_{i-1,j-1}$ to $e'_k = (s'_k, v'_k, t'_k) \in D_{i,j}$, shown as follows.

$$\text{EXT}_1(e_k) = (s_k - v_k^\gamma + (v_k+1)^\gamma, v_k+1, t_k-1). \quad (8)$$

Similarly, when $x_i \neq y_j$, EXT_2 is defined as follows.

$$\text{EXT}_2(e_k) = (s_k, 0, \infty). \quad (9)$$

The new algorithm for constructing the dominant list $D_{i,j}$ with survival time is shown in Algorithm 2, and the function $\text{INSERT}(D_{i,j}, e')$ is described by Algorithm 3.

Here we provide some explanations for Algorithms 2 and 3. To construct the dominant list $D_{i,j}$, extended solutions are collected from $D_{i-1,j}$, $D_{i,j-1}$ and $D_{i-1,j-1}$. For extended solutions from $D_{i-1,j}$ or $D_{i,j-1}$, the solution with the highest

Algorithm 2 Construction of $D_{i,j}$ with predicted survival time.

$D_{i,0} = D_{0,j} = \{(0, 0, \infty)\}$
for $i = 1 \to m$ **do**
  for $j = 1 \to n$ **do**
    $E = \text{EXT}_2(D_{i-1,j,1}) \cup \text{EXT}_2(D_{i,j-1,1})$
    $e' = \text{DOM}(E)$
    if $x_i = y_j$ **then**
      $D_{i,j} = \text{EXT}_1(D_{i-1,j-1})$
      if $t_{i,j,1} < 0$, remove $D_{i,j,1}$ from $D_{i,j}$
      $\text{INSERT}(D_{i,j}, e')$
    else
      $D_{i,j} = \{e'\}$
    end if
  end for
end for

Algorithm 3 Insert e' into $D_{i,j}$ and maintain elements in the list.

function $\text{INSERT}(D_{i,j},\ e' = (s', v', t'))$
  if $s' \leq s_1$, **then** discard e' and **return**
  $k \leftarrow 1$
  $t' = \text{INTERSECT}(e', e_1)$
  while $t' \geq t_k$ **do**
    remove e_k from $D_{i,j}$
    $k \leftarrow k + 1$
    $t' = \text{INTERSECT}(e', e_k)$
  end while
  push e' on top of $D_{i,j}$
end function

score would dominate all other solutions, because $v' = 0$. That is, what we have to compare is the top solutions in $D_{i-1,j}$ and $D_{i,j-1}$, which are $D_{i-1,j,1}$ and $D_{i,j-1,1}$. For $x_i = y_j$, $D_{i,j}$ can be constructed by extending $D_{i-1,j-1}$, removing expired solutions, and inserting the solution $e' = (s', v', t')$ (extended from $D_{i-1,j}$ or $D_{i,j-1}$).

To insert $e' = (s', v', t')$ into $D_{i,j}$, and to maintain the dominant solutions, the function INSERT is presented in Algorithm 3. First, to ensure that e' is not dominated by any other solution in $D_{i,j}$, since for each $e_k \in D_{i,j}$ we have $v_k \geq v' = 0$, $s' > s_1$ should be satisfied. Otherwise e' is dominated and it must be discarded. Next, t' should be updated by intersecting e' with existing solutions in $D_{i,j}$. For some solution $e_k = (s_k, v_k, t_k)$, if $t' \geq t_k$, then e_k is dominated by e', which means e_k should be removed. Note that even when $t' = t_k$, e_k should be removed because $s' > s_k$. There are up to $|D_{i,j}|$ intersections to be computed, and finally, e' should be placed on the top of $D_{i,j}$.

Figure 3 illustrates an example for $X =$ `aabbbacbba`, $Y =$ `aabbacba`, and $\gamma = 2$, where the situation $x_i = y_j$ is specified by grey color. For demonstration, to construct the dominant list of the cell $(5, 4)$, solutions from $D_{4,4}$, $D_{5,3}$ and $D_{4,3}$ should be collected. For the solutions from $D_{4,4}$ and $D_{5,3}$, since the v' will become 0, the solution with the highest s' will dominate all the others in $D_{4,4}$ and $D_{5,3}$. By comparing the extended score of $(16, 4, \infty)$ (the best one in $D_{4,4}$) and $(9, 0, 2)$ (the best one in $D_{5,3}$), the solution $e' = (16, 0, \infty)$ (extended from $(16, 4, \infty)$) is retrieved. For the solutions from $D_{4,3}$, since the situation has been predicted, the list $\{(9, 0, 2), (5, 1, \infty)\}$ is extended as $\{(10, 1, 1), (8, 2, \infty)\}$. Since t values are in increasing order , the first element (with the smallest t value) in the list should be checked for expiration. The expired solution will be removed. In this example, no solution is expired.

Following Algorithm 3, the detailed steps for inserting $e' = (16, 0, \infty)$ into $\{(10, 1, 1), (8, 2, \infty)\}$ are described as follows.

- At first, because $s' = 16 > s_1 = 10$, we know that e' is not dominated by any other solutions in the list, and it should be inserted.
- To update t' of e', the intersection of e' and $D_{5,4,1} = (10, 1, 1)$ is first computed, which is $t' = 3$. By $t' = 3 \geq t_{5,4,1} = 1$, $D_{5,4,1}$ is dominated and removed.
- Moving on to the new $D_{5,4,1} = (8, 2, \infty)$, the intersection of e' and $D_{5,4,1}$ is $t' = 2$. Because $t' = 2 < t_{5,4,1} = \infty$ is satisfied, no more domination will occur, and the loop terminates.
- To keep the list in order, e' is placed on the top of $D_{5,4}$.
- Finally, the process of $D_{5,4}$ finishes. We have $D_{5,4} = \{(16, 0, 2), (8, 2, \infty)\}$.

For $D_{i,j}$ with $x_i \neq y_j$, the dominant list can be constructed in constant time. However, for the case $x_i = y_j$, both the extension from $D_{i-1,j-1}$ and the function INSERT require $O(r)$ time, where r is the maximal size of the dominant list. Thus, the time complexity of this improved algorithm is $O(n^2 r)$, which is bounded by $O(n^3)$.

3.3 The $O(n^2)$-time Algorithm

After analyzing the time complexity of the algorithm in Section 3.2, the bottlenecks are the task for extending solutions from $D_{i-1,j-1}$ to $D_{i,j}$, and the function INSERT for inserting e' into $D_{i,j}$. In this section, we define a new solution representation, by which the one-by-one extension from $D_{i-1,j-1}$ to $D_{i,j}$ can be avoided, and the information required by the algorithm is still retrievable. After that, by amortizing the time required for checking and deleting elements in INSERT, we show that the optimal FLCS can be determined in $O(n^2)$ time.

Instead of the representation $e_k = (s_k, v_k, t_k)$ used in Section 3.2, a new representation of a solution $\epsilon_k = (\sigma_k, \nu_k, \tau_k)$ is defined as follows. First, we use σ_k to represent the score that ϵ_k has accumulated from the beginning till the last mismatched pair, but without the last matched segment. In other words, $s_k = \sigma_k + v_k^\gamma$. In addition, ν_k denotes the row index of the lattice cell where the last mismatched pair of ϵ_k is. Hence, for a solution ϵ_k in $D_{i,j}$, we have $v_k = i - \nu_k$. Finally, let τ_k be the largest row index of the lattice cell where ϵ_k still survives. That is, we have $t_k = \tau_k - i$ for each $\epsilon_k \in D_{i,j}$.

These two kinds of representations e_k and ϵ_k can be converted to each other in constant time. Clearly, $e_k = (s_k, v_k, t_k) = (\sigma_k + (i - \nu_k)^\gamma, i - \nu_k, \tau_k - i)$, and $\epsilon_k = (\sigma_k, \nu_k, \tau_k) = (s_k - v_k^\gamma, i - v_k, t_k + i)$. As $e_k = (0, 0, \infty)$ in $D_{0,j}$ or $D_{i,0}$ for an initial solution, the corresponding initial solution is now set as $\epsilon_k = (0 - 0^\gamma, i - 0, \infty + i) = (0, i, \infty)$. Therefore, with this new representation, a solution ϵ_k can be extended to ϵ'_k by EXTD$_1$ or EXTD$_2$ with Equation 10.

		0	1	2	3	4	5	6	7	8
		-	a	a	b	b	a	c	b	a
0	-	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)
1	a	(0,0,∞)	(1,1,∞)	(1,1,∞)	(1,0,∞)	(1,0,∞)	(1,1,∞)	(1,0,∞)	(1,0,∞)	(1,1,∞)
2	a	(0,0,∞)	(1,1,∞)	(4,2,∞)	(4,0,∞)	(4,0,∞)	(4,0,1) (2,1,∞)	(4,0,∞)	(4,0,∞)	(4,0,1) (2,1,∞)
3	b	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,3,∞)	(9,0,2) (5,1,∞)	(9,0,∞)	(9,0,∞)	(9,0,2) (5,1,∞)	(9,0,∞)
4	b	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,0,2) (5,1,∞)	(16,4,∞)	(16,0,∞)	(16,0,∞)	(16,0,3) (10,1,∞)	(16,0,∞)
5	b	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,0,2) (5,1,∞)	(16,0,2) (8,2,∞)	(16,0,∞)	(16,0,∞)	(17,1,∞)	(17,0,∞)
6	a	(0,0,∞)	(1,1,∞)	(4,0,1) (2,1,∞)	(9,0,∞)	(16,0,∞)	(17,1,1) (13,3,∞)	(17,0,∞)	(17,0,∞)	(20,2,∞)
7	c	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,0,∞)	(16,0,∞)	(17,0,∞)	(20,2,0) (20,4,∞)	(20,0,∞)	(20,0,∞)
8	b	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,0,2) (5,1,∞)	(16,0,3) (10,1,∞)	(17,0,∞)	(20,0,∞)	(29,5,∞)	(29,0,∞)
9	b	(0,0,∞)	(1,0,∞)	(4,0,∞)	(9,0,2) (5,1,∞)	(16,0,2) (8,2,∞)	(17,0,∞)	(20,0,∞)	(29,0,4) (21,1,∞)	(29,0,∞)
10	a	(0,0,∞)	(1,1,∞)	(4,0,1) (2,1,∞)	(9,0,∞)	(16,0,∞)	(17,1,1) (13,3,∞)	(20,0,∞)	(29,0,∞)	(30,1,3) (24,2,∞)

Figure 3: The 2D lattice of the dynamic programming with the efficient dominant method.

$$\begin{aligned} \text{EXTD}_1(\epsilon_k) &= (\sigma_k, \nu_k, \tau_k) \quad \text{for } x_i = y_j \\ \text{EXTD}_2(\epsilon_k) &= (\sigma_k + v_k^\gamma, i, \infty) \quad \text{for } x_i \neq y_j. \end{aligned} \tag{10}$$

The computation for t_k and v_k is transformed to the row indexes of lattice cells, which need not be updated cell by cell. The score σ_k is updated only when the last matched segment terminates with $x_i \neq y_j$. Note that $v_k = i - \nu_k$, where i is the row index of the cell that ϵ_k comes from. Since only one solution is concerned for a mismatched pair, we update variables only when a mismatch occurs. Hence, we have a representation that reduces the one-by-one extension from $D_{i-1,j-1}$.

We apply the new representation ϵ_k based the algorithm in Section 3.2, obtaining a new modified algorithm. By $e_k = (s_k, v_k, t_k) = (\sigma_k + (i-\nu_k)^\gamma, i - \nu_k, \tau_k - i)$, we have the following transformation. First, the score predicting function $f_k(\tau)$ is shown as Equation 11.

$$\begin{aligned} f_k(t) &= s_k - v_k^\gamma + (v_k + t)^\gamma \\ &= \sigma_k + (i + t - \nu_k)^\gamma \\ f_k(\tau) &= \sigma_k + (\tau - \nu_k)^\gamma \end{aligned} \tag{11}$$

Next, the function $\text{INTERSECT}(\epsilon_{k_1}, \epsilon_{k_2})$ returns the integer-precise point τ that makes $f_{k_1}(\tau) = f_{k_2}(\tau)$. The modified algorithm to construct dominant lists is shown in Algorithm 4, where $\tau_{i,j,1}$ is the counterpart of $t_{i,j,1}$, and the modified insertion function is shown in Algorithm 5.

Note that the original bottleneck of extending solutions from $D_{i-1,j-1}$ to $D_{i,j}$ is now drastically reduced with the simple operation $D_{i,j} =$

Algorithm 4 Construction of $D_{i,j}$ for each lattice cell (i, j).

```
D_{i,0} = D_{0,j} = {(0, i, ∞)}
for i = 1 → m do
    for j = 1 → n do
        E = EXTD_2(D_{i-1,j,1}) ∪ EXTD_2(D_{i,j-1,1})
        ε' = DOM(E)
        if x_i = y_j then
            D_{i,j} = D_{i-1,j-1}
            if τ_{i,j,1} < i, remove D_{i,j,1} from D_{i,j}
            INST(D_{i,j}, ε')
        else
            D_{i,j} = {ε'}
        end if
    end for
end for
```

Algorithm 5 Insert ϵ' into $D_{i,j}$ and maintain elements in the list.

```
function INST(D_{i,j}, ε' = (σ', ν', τ'))
    if σ' ≤ σ_1 + (i − ν_1)^γ, then discard ε' and return
    k ← 1
    τ' = INTERSECT(ε', ε_1)
    while τ' ≥ τ_k do
        remove ε_k from D_{i,j}
        k ← k + 1
        τ' = INTERSECT(ε', ε_k)
    end while
    push ε' on top of D_{i,j}
end function
```

$D_{i-1,j-1}$. It can be done in constant time by shallow copying from $D_{i-1,j-1}$ to $D_{i,j}$. In other words, the address of $D_{i-1,j-1}$ is copied as the address of $D_{i,j}$, and only a minor part of data are updated.

Here we again take the same example $X =$ `aabbbacbba`, $Y =$ `aabbacba`, and $\gamma = 2$ for comparison. The computation of the previous algorithm is shown in Figure 3, and the computation with the new representation is shown in Figure 4. The dominant lists are constructed row by row as the progressing of dynamic programming. Since the address of $D_{i-1,j-1}$ is copied to $D_{i,j}$, the memory of $D_{i-1,j-1}$ is reused in $D_{i,j}$. Thus, only the dominant lists in the current row are recorded, and the lists in previous rows are simply moved and slightly modified. In Figure 4, one can find that lists $D_{i-1,j-1}$ and $D_{i,j}$ are almost the same if $x_i = y_j$.

For demonstration, to construct the dominant list $D_{5,4}$, solutions from $D_{4,4}$, $D_{5,3}$ and $D_{4,3}$ are collected. For the solutions from $D_{4,4}$ and $D_{5,3}$, the solution ϵ' with the highest score $\sigma_k + (i - \nu_k)^\gamma$ will dominate all the others in $D_{4,4}$ and $D_{5,3}$. By comparing $(0, 0, \infty)$ (the best one in $D_{4,4}$ with extended score $0 + (4-0)^2 = 16$) and $(9, 5, 7)$ (the best one in $D_{5,3}$ with extended score $9 + (5-5)^2 = 9$), the solution $\epsilon' = (16, 5, \infty)$ (extended from $(0, 0, \infty)$ in $D_{4,4}$) is retrieved. For the solutions from $D_{4,3}$, the whole list $\{(9, 4, 6),\ (4, 3, \infty)\}$ is kept for $D_{5,4}$. However, the first element in the list should be checked for expiration. Because $\tau_{5,4,1} = 6 \geq i = 5$, no solution is expired. Following Algorithm 5, the detailed steps for inserting $\epsilon' = (16, 5, \infty)$ into $\{(9, 4, 6),\ (4, 3, \infty)\}$ are described as follows.

- At first, because $\sigma' = 16 > \sigma_1 + (i - \nu_1)^\gamma = 10$, we know that ϵ' is not dominated by any other solutions in the list, and it should be inserted.
- To update τ' of ϵ', the intersection of ϵ' and $D_{5,4,1} = (9, 4, 6)$ is first computed, which is 8. By $\tau' = 8 \geq \tau_{5,4,1} = 6$, $D_{5,4,1}$ is dominated and removed.
- Moving on to the new $D_{5,4,1} = (4, 3, \infty)$, the intersection of ϵ' and $D_{5,4,1}$ is 7. Because $\tau' = 7 < \tau_{5,4,1} = \infty$ is satisfied, no more domination will occur, and the loop stops.
- To keep the list in order, ϵ' is placed on the top of $D_{5,4}$.
- Finally, the maintenance process of $D_{5,4}$ finishes. We have $D_{5,4} = \{(16, 5, 7),\ (4, 3, \infty)\}$.

To end this section, we give the analysis of the amortized time complexity. By Algorithm 4, at most one solution is newly generated and inserted into the list of each $D_{i,j}$. The number of all new solutions is therefore bounded by $O(n^2)$, which is also the maximal number of times to remove a solution. Though the function INST may take linear time for processing some $D_{i,j}$, the overall time is still bounded by $O(n^2)$. Note that by applying Hirschberg's divide-and-conquer strategy [7], the optimal solution can also be obtained in $O(n^2)$ time, which completes our algorithm.

4 Conclusion

In this paper, we first define the flexible longest common subsequence (FLCS) problem with the concept of matching continuity. To find the optimal solution of FLCS, an $O(n^2 r \log r)$-time algorithm is first developed, which is further improved to $O(n^2 r)$ and $O(n^2)$, where n denote the longer length of the two input sequences, and r is the size of dominant list. In real applications, the FLCS score is useful for estimating the similarity with emphasized continuity. In the future, we hope to design a more general function that simultaneously considers the penalty of gaps, providing a more flexible tool for sequence analysis.

References

[1] A. Abdulla-Al-Maruf, H.-H. Huang, and K. Kawagoe, "Time series classification method based on longest common subsequence and textual approximation," *Proceeding of 2012 Seventh International Conference on Digital Information Management (ICDIM)*, pp. 130–137, 2012.

[2] H.-Y. Ann, C.-B. Yang, Y.-H. Peng, and B.-C. Liaw, "Efficient algorithms for the block edit problems," *Information and Computation*, Vol. 208(3), pp. 221–229, 2010.

[3] 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, pp. 360–364, 2008.

[4] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "Fast algorithms for comput-

		0	1	2	3	4	5	6	7	8
		-	a	a	b	b	a	c	b	a
0	-	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)	(0,0,∞)
1	a	(0,1,∞)	(0,0,∞)	(0,0,∞)	(1,1,∞)	(1,1,∞)	(0,0,∞)	(1,1,∞)	(1,1,∞)	(0,0,∞)
2	a	(0,2,∞)	(0,1,∞)	(0,0,∞)	(4,2,∞)	(4,2,∞)	(4,2,3) (1,1,∞)	(4,2,∞)	(4,2,∞)	(4,2,3) (1,1,∞)
3	b	(0,3,∞)	(1,3,∞)	(4,3,∞)	(0,0,∞)	(9,3,5) (4,2,∞)	(9,3,∞)	(9,3,∞)	(9,3,5) (4,2,∞)	(9,3,∞)
4	b	(0,4,∞)	(1,4,∞)	(4,4,∞)	(9,4,6) (4,3,∞)	(0,0,∞)	(16,4,∞)	(16,4,∞)	(16,4,7) (9,3,∞)	(16,4,∞)
5	b	(0,5,∞)	(1,5,∞)	(4,5,∞)	(9,5,7) (4,4,∞)	(16,5,7) (4,3,∞)	(16,5,∞)	(16,5,∞)	(16,4,∞)	(17,5,∞)
6	a	(0,6,∞)	(0,5,∞)	(4,6,7) (1,5,∞)	(9,6,∞)	(16,6,∞)	(16,5,7) (4,3,∞)	(17,6,∞)	(17,6,∞)	(16,4,∞)
7	c	(0,7,∞)	(1,7,∞)	(4,7,∞)	(9,7,∞)	(16,7,∞)	(17,7,∞)	(16,5,7) (4,3,∞)	(20,7,∞)	(20,7,∞)
8	b	(0,8,∞)	(1,8,∞)	(4,8,∞)	(9,8,10) (4,7,∞)	(16,8,11) (9,7,∞)	(17,8,∞)	(20,8,∞)	(4,3,∞)	(29,8,∞)
9	b	(0,9,∞)	(1,9,∞)	(4,9,∞)	(9,9,11) (4,8,∞)	(16,9,11) (4,7,∞)	(17,9,∞)	(20,9,∞)	(29,9,13) (20,8,∞)	(29,9,∞)
10	a	(0,10,∞)	(0,9,∞)	(4,10,11) (1,9,∞)	(9,10,∞)	(16,10,∞)	(16,9,11) (4,7,∞)	(20,10,∞)	(29,10,∞)	(29,9,13) (20,8,∞)

Figure 4: The 2D lattice for the dynamic programming with the new representation.

ing the constrained lcs of run-length encoded strings," *Theoretical Computer Science*, Vol. 432, pp. 1–9, 2012.

[5] K.-Y. Cheng, K.-S. Huang, and C.-B. Yang, "The longest common subsequence problem with the gapped constraint," *Proc. of the 30th Workshop on Combinatorial Mathematics and Computation Theory*, pp. 80–85, 2013.

[6] A. Flores-Mendez and M. Bernal-Urbina, "Dynamic signature verification through the longest common subsequence problem and genetic algorithms," *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–6, 2010.

[7] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, pp. 341–343, 1975.

[8] 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(5), pp. 188–193, 2008.

[9] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20(5), pp. 350–353, 1977.

[10] C. S. Iliopoulos and M. S. Rahman, "Algorithms for computing variants of the longest common subsequence problem," *Theoretical Computer Science*, Vol. 395, pp. 255–267, 2008.

[11] Y.-S. Jeong, M. K. Jeong, and O. A. Omitaomu, "Weighted dynamic time warping for time series classification," *Pattern Recognition*, Vol. 44, No. 9, pp. 2231 – 2240, 2011.

[12] D. R. Kincaid and E. W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*. American Mathematical Soc., third ed., 2002.

[13] Y. Namiki, T. Ishida, and Y. Akiyama, "Acceleration of sequence clustering using longest common subsequence filtering," *BMC Bioinformatics*, Vol. 14, No. Suppl 8, p. S7, 2013.

[14] M. Pawlik and N. Augsten, "RTED: a robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, Vol. 5, No. 4, pp. 334–345, 2011.

[15] 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.

[16] I. Stewart, *Galois theory*. Chapman Hall/CRC Mathematics, third ed., 2003.

[17] 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, pp. 44–52, 2013.