 ISBN/978-957-437-895-1(PDF)

Diagonal Algorithms for the Longest Common Subsequence Problems with t-length and at Least t-length Substrings *

Guo-Fong Huang[a], Chang-Biau Yang[a]†, Kuo-Tsung Tseng[b] and Kuo-Si Huang[c]
[a]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan

[b]Department of Shipping and Transportation Management
[c]Department of Business Computing
[b,c]National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan

Abstract

The LCS_t *(longest common subsequence problem with t-length substrings) problem a variant of the longest common subsequence* (LCS) *problem, in which the LCS answer consists of t-length substrings. For the* LCS_{t+} *(LCS with at least t-length substrings) problem, the LCS answer is composed of substrings with at least t-length. In this paper, we propose an efficient algorithm with three implementations for solving the* LCS_t*, based on the concept of the diagonal method, with time complexities* $\mathrm{O}(n(m-L))$, $\mathrm{O}(n+l(m-L)+R)$, *and* $\mathrm{O}(n+l(m-L)\log n)$, *where* m *and* n *denote the lengths of the two input sequences* A *and* B*, respectively, and* $m \leq n$*;* l *denotes the number of the common* t*-length substrings in the* LCS_t *answer;* L *denotes the* LCS_t *length;* R *is the number of* t*-match pairs (with* t*-length substrings) between* A *and* B*. We also propose an algorithm for solving the* LCS_{t+} *problem in* $\mathrm{O}(n+L(m-L)+R)$ *time. The experimental results show that our algorithms are efficient when the two input sequences are highly similar.*

Keywords: longest common subsequence, longest common subsequence with t-length substrings, diagonal method, match pair

1 Introduction

The *longest common subsequence problem with t-length substrings* (LCS_t) and the *longest common subsequence problem with at least t-length substrings* (LCS_{t+}) are two variants of the *longest common subsequence* (LCS) problem [11]. The LCS problem has been well-studied for several decades and it has many variants [4, 5, 7, 13, 15].

The biosequence alignment is one of valuable applications of the LCS problem, but the answer of the traditional LCS problem might not be so meaningful for RNA sequences, since each codon consists of three nucleotides. In 2013, Beson *et al.* [2] defined a new similarity measure, the *longest common subsequence problem with t-length substrings* (LCS_t), in which the LCS answer consists of t-length common substrings. For example, suppose that $A = \texttt{actaacg}$, $B = \texttt{tacttacacg}$, and $t = 3$. We have $LCS_3(A, B) = \texttt{actacg} = A_{1..3}A_{5..7} = B_{2..4}B_{8..10}$. In this problem, the length of each common substring must be exactly $t = 3$. In 2014, Pavetić *et al.* [12] proposed a variant of the LCS_t problem, the *longest common subsequence problem with at least t-length substrings* (LCS_{t+}), where the length of each common substring in the answer is greater than or equal to t.

After LCS_t and LCS_{t+} problems have been presented, many researchers proposed the algorithms for solving these problem with dynamic programming, van Emde Boas(vEB) tree or match pair, and so on. The summary of these related algorithms, including our algorithms, are presented in Table 1.

In this paper, we propose a diagonal algorithm for solving the LCS_t problem with three different searching approach and one diagonal algorithm for solving the LCS_{t+} problem. Our diagonal algorithms for solving these problems are inspired by the LCS algorithm proposed by Nakatsu *et al.*

*This research work was partially supported by the Ministry of Science and Technology of Taiwan under contract MOST 108-2221-E-110-031.
†Corresponding author: cbyang@cse.nsysu.edu.tw.

Table 1: The time and space complexities of the LCS_t and LCS_{t+} algorithms. Here, t denotes the length of each common substring; L denotes the length of the answer; l denotes the number of the common t-length substrings in the answer, where $L = l \times t$ in the LCS_t problem; R denotes the number of t-match pairs between A and B.

LCS_t

Year	Author(s)	Time complexity	Space complexity	Note
2013	Benson *et al.*	$O(tmn)$	$O(tm)$	Dynamic programming
2014	Deorowicz and Grabowski	DP: $O(mn)$ Sparse: $O(n + R \log l)$ Dense: $O(mn/t + n(t \log n)^{2/3})$ Dense-vEB: $O(mn \log\log m/t)$ DP-4R: $O(mn/\log n)$	DP: $O(tm)$ Sparse: $O(n + min\{R, ml\})$ Dense: $O(m)$ Dense-vEB: $O(m \log\log m)$ DP-4R: $O(n + nt/\log n)$	Sparse dynamic programming, van Emde Boas tree
2016	Benson *et al.*	$O(mn)$	$O(tm)$	Dynamic programming
2017	Zhu *et al.*	$O(mn)$	$O(tm)$	Divide-and-conquer
2018	Pavetić *et al.*	$O(m + n + R + min\{R \log l, R + ml\})$	$O(m + n)$	Match pair
2020	This paper	$O(n(m - L))$ $O(n + l(m - L) \log n)$ $O(n + l(m - L) + R)$	$O(n)$	Diagonal

LCS_{t+}

Year	Author(s)	Time complexity	Space complexity	Note
2014	Pavetić *et al.*	$O(m + n + R \log R)$	$O(n + m + R)$	Sparse dynamic programming
2016	Benson *et al.*	$O(tmn)$	$O(tm)$	Dynamic programming
2017	Ueki *et al.*	$O(mn)$	$O(tm)$, $n > m$	Dynamic programming
2017	Zhu *et al.*	$O(mn)$	$O(tm)$	Divide-and-conquer
2018	Pavetić *et al.*	$O(m + n + R + min\{R(\log L + t), R + mL\})$	$O(m + n)$	Match pair
2020	This paper	$O(n + L(m - L) + R)$	$O(R)$	Diagonal

[10]. Its time complexity is close to linear if the two input sequences are highly similar. The diagonal algorithm has achieved the success for solving various algorithms efficiently, such as the merged LCS problem [15], the constrained LCS problem [7], and the longest common increasing subsequence problem [9]. The very theoretical study on the 2-dimensional LCS problem was also presented by Chan *et al.* [3].

The rest of this paper is organized as follows. In Section 2, we present the definitions of the LCS, LCS_t and LCS_{t+} problems. We propose our diagonal algorithms for solving the LCS_t and LCS_{t+} problems in Sections 3 and 4, respectively. Some experimental results are provided to compare the efficiencies of our algorithms with some previously published algorithms in Section 5. Finally, we give our conclusions in Section 6.

2 The Longest Common Subsequence Problems with t-length and at least t-length Substrings (LCS_t/LCS_{t+})

The longest common subsequence problem with t-length substrings (LCS_t) problem is formally defined as follows.

Definition 1. [2] Given two sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, if there is a common subsequence $C = A_{i_1-t+1..i_1}\ A_{i_2-t+1..i_2}\ \cdots\ A_{i_l-t+1..i_l} = B_{j_1-t+1..j_1}\ B_{j_2-t+1..j_2} \cdots\ B_{j_l-t+1..j_l}$, where $t \le i_{g-1} \le i_g - t$ and $t \le j_{g-1} \le j_g - t$ for $2 \le g \le l$, we say that C is a common subsequence with t-length substrings of A and B. The LCS_t problem is to find a common subsequence, consisting of t-length substrings of A and B, with the maximal length.

For example, suppose that $A =$ `actaacg`, $B =$ `ctgacactcg` and $t = 2$. The LCS_2 answer is $A_{2..3}A_{5..6} = B_{1..2}B_{4..5} =$ `ctac`, with length 4.

The longest common subsequence problem with at least t-length substrings (LCS_{t+}) is formally defined as follows.

Definition 2. [12] Given two sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, if there is a common subsequence $C = A_{i_1-s_1+1..i_1}\ A_{i_2-s_2+1..i_2} \cdots A_{i_T-s_T+1..i_T} = B_{j_1-s_1+1..j_1}\ B_{j_2-s_2+1..j_2}\ \cdots\ B_{j_T-s_T+1..j_T}$, where $s_{g-1} \le i_{g-1} \le i_g - s_g \le m - s_g$ and $s_{g-1} \le j_{g-1} \le j_g - s_g \le n - s_g$ for $2 \le g \le T$, and $t \le s_c$ for $1 \le c \le T$, we say that C is a common subsequence with at least t-length substrings of A and B. The LCS_{t+} problem is to find a common subsequence, consisting of at least t-length substrings of A and B, with the maximal length.

For example, suppose that $A =$ `actaacg`, $B =$ `ctgacactcg` and $t = 2$. The LCS_{2+} answer of this example is $A_{1..3}A_{6..7} = B_{6..8}B_{9..10} =$ `actcg`, with length 5.

In 2013, Benson *et al.* [2] first defined the LCS_t problem and proposed a dynamic programming algorithm with the t-match pair concept. The t-match pair $tMatch(i, j)$ is defined in Equation 1. Equation 2 shows the dynamic programming algorithm of Benson *et al.* The time complexity is $O(tmn)$ since it needs $O(t)$ time for determining each $tMatch(i, j)$, and the space complexity is $O(n^2)$.

$$tMatch(i,j) = \begin{cases} 1 & \text{if } A_{i-t+1..i} = B_{j-t+1..j}, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

$$BS_t(i,j) = \max \begin{cases} 0 & \text{if } 1 \le i, j \le t-1 \\ BS_t(i, j-1) & \\ BS_t(i-1, j) & \\ BS_t(i-t, j-t) + & \\ \quad tMatch(i,j) \times t & \end{cases} \tag{2}$$

In 2014, Pavetić *et al.* [12] defined the LCS_{t+} problem, a variant of the LCS_t problem They also proposed a sparse dynamic algorithm for solving

Table 2: An example of constructing $d_{i,s}$ for the LCS_t algorithm with A = `actaacg`, B = `ctgacactcg` and $t = 2$. The LCS_2 answer is $A_{2..3}A_{5..6} = B_{1..2}B_{4..5}$ = `ctac`, with length 4.

round r \ s (length)	**0**	**2**	**4**
1	$d_{0,0}$	$d_{2,2}$	$d_{4,4}$
	0	5	∞
2	$d_{1,0}$	$d_{3,2}$	$d_{5,4}$
	0	2	∞
3	$d_{2,0}$	$d_{4,2}$	$d_{6,4}$
	0	2	5

it. Their algorithm is more efficient if given sequences are dissimilar. The complexities of the related algorithms are shown in Table 1.

3 The Diagonal Algorithms for LCS_t

Some published algorithms for LCS_t are based on the t-match pairs. The number of t-match pairs may be close to $O(mn)$ when the size of the alphabet set is very small. These algorithms need more time if the two input sequences are similar. On the contrary, our algorithm is very efficient for solving the LCS_t problem with highly similar sequences.

Let $d_{i,s} = j$ denote the smallest position index j of B such that $LCS_t(A_{1..i}, B_{1..j}) = s$. An example of constructing $d_{i,s}$ with $t = 2$ (i.e. LCS_2) for A = `actaacg` and B = `ctgacactcg` is shown in Table 2. The LCS_2 answer is $A_{2..3}A_{5..6} = B_{1..2}B_{4..5}$ = `ctac`, with length 4.

We explain our algorithm with the example round by round. In the first round, the 1st and 2nd characters of A are a_1a_2 = `ac`. One can find that b_4b_5 = `ac`. Then we get $d_{2,2} = 5$, which means that $\text{LCS}_2(A_{1..2}, B_{1..5})$=2. By the same way, the 3rd and 4th characters of A are a_3a_4 = `ta`, but there is no `ta` in $B_{6..10}$ (after b_5). So the first round stops.

In the second round, we start from the 2nd character of A. We have a_2a_3 = `ct`, and one can find that b_1b_2 = `ct`. So $d_{2,2} = 5$ is replaced by $d_{3,2} = 2$ since the latter uses fewer characters of B to obtain an answer of length 2. Next, a_4a_5 = `aa`, but there is no `aa` in $B_{3..10}$. So the second round stops.

In the third round, we start from a_3. We have a_3a_4 = `ta`, but there is no `ta` in B. So we set $d_{4,2} = 2$ to the same as $d_{3,2} = 2$. Next, a_5a_6 = `ac`. One can find that b_4b_5 = `ac` after b_2, so we get $d_{6,4} = 5$. It means that $\text{LCS}_2(A_{1..6}, B_{1..5})$=4. The third round stops here. The algorithm terminates since there does not exist any solution better than length 4, since there are only four characters remained in $A_{4..7}$ for the next round.

It is obvious that the time complexity may be various if we use different searching approaches for finding t-match pairs. The first searching approach is to scan sequence B sequentially. The second and third searching approaches find all t-match pairs in the preprocessing phase. For comparing two t-length substrings (shortened as t-substrings) in constant time, we need to classify t-substrings into several groups by constructing the *longest common prefix* (LCP) table [8], which was also used in the preprocessing of Deorowicz and Grabowski [6]. The second and third searching approaches can be easily done with the LCP table. The details of modifying the searching approaches to improve our algorithm are omitted here.

The dynamic programming formula of our diagonal algorithm for solving LCS_t problem is given in Equation 3. The function $Match(A_{i-t+1..i}, j')$ returns the smallest j such that $j \geq j'$ and $A_{i-t+1..i} = B_{j-t+1..j}$. If there is no such j, it returns ∞.

$$d_{i,s} = \min \begin{cases} 0 & \text{if } s = 0, \\ \infty & \text{if } 1 \leq s < t \text{ or } i < s, \\ d_{i-1,s} & \\ Match(A_{i-t+1..i}, & \text{if } s \geq t. \\ \quad d_{i-t,s-t} + t) & \end{cases} \tag{3}$$

Note that practically, we need calculate only $d_{i,s}$ for s that is a multiple of t, since there is no solution for other values of s.

Lemma 1. *If $d_{r+l\times t-1,l\times t}$ can be extended to $d_{r+g+(l+1)\times t-1,(l+1)\times t}$, $0 \leq g \leq t-1$, $d_{r+g+(l+1)\times t-1,(l+1)\times t}$ will be checked in round $(r+g)$.*

Proof. Suppose that $d_{r+l\times t-1,l\times t}$ can be extended to $d_{r+g+(l+1)\times t-1,(l+1)\times t}$, $0 \leq g < t$. If $d_{r+l\times t-1,l\times t} \neq \infty$, it means that we have found l t-matches in round r or before. Since l t-matches have been already found, $d_{r+g+t-1,t}, d_{r+g+2t-1,2t}, \cdots, d_{r+g+l\times t-1,l\times t}$ and $d_{r+g+(l+1)\times t-1,(l+1)\times t}$ will be checked in round $(r+g)$.

□

The difference of our LCS_t algorithms and the traditional LCS algorithm is that we treat a t-length substring as a character. In our diagonal

 ISBN/978-957-437-895-1(PDF)

LCS_t algorithms, if a t-match pair in the rest part of A and B is extendable, then it will be checked in a certain future round by Lemma 1. The pseudo code of our LCS_t algorithm is shown in Algorithm 1. The three searching approaches differ only in line 8.

Algorithm 1 A diagonal algorithm for LCS_t problem.

Input: Two sequences $A = a_1a_2a_3\cdots a_m$ and $B = b_1b_2b_3\cdots b_n$, where $m \le n$, and the length t of common substrings.
Output: The length of $LCS_t(A, B)$

1: $L \leftarrow 0$ ▷ initial length of $LCS_t(A, B)$
2: **for** $r = 1 \to m$ **do** ▷ round r
3: **if** $L > m - r$ **then**
4: break ▷ cannot obtain longer LCS_t
5: $d_{r-1,0} \leftarrow 0$, $s \leftarrow t$ ▷ initialization
6: **while** true **do**
7: $i \leftarrow r + s - 1$
8: $j \leftarrow Match(A_{i-t+1..i}, d_{i-t,s-t}+t)$ ▷ j is the smallest position index such that $d_{i-t,s-t}+t \le j \le n$ and $A_{i-t+1..i} = B_{j-t+1..j}$
9: $d_{i,s} \leftarrow \min(d_{i-1,s}, j)$ ▷ $d_{i-1,s} = \infty$ in round 0
10: **if** $d_{i,s} < \infty$ **then**
11: $L \leftarrow \max(L, s)$
12: **else**
13: break
14: $s \leftarrow s + t$
15: **return** L

4 The Diagonal Algorithm for LCS_{t+}

In the diagonal LCS_t algorithm, we can treat a t-length substring as a character in the LCS algorithm. However, we cannot do this simple treatment for the LCS_{t+} problem. In the LCS_{t+} problem, the solution of $d_{i,s}$ can be extended from each $d_{i-g,s-g}$ with a common g-length substring of A and B, $t \le g \le 2t-1$. For example, suppose that $A =$ `actgca` and $B =$ `acxtgcxactxgc`, $t = 2$. The calculation result of $d_{i,s}$ is shown in Table 3. $d_{5,5} = 6$ is extended from $d_{2,2} = 2$ with $A_{3..5} = B_{4..6}$. The solution of $d_{5,5}$ may be extended with a common substring of length 2 or 3.

Equation 4 presents the dynamic programming of our diagonal algorithm for solving the LCS_{t+} problem.

Table 3: An example for the LCS_{t+} algorithm with $A =$ `actgca`, $B =$ `acxtgcxactxgc` and $t = 2$. s denotes the LCS_{t+} length of A and B, and r denotes the number of the round.

r \ s	**0**	**1**	**2**	**3**	**4**	**5**
1	$d_{0,0}$	$d_{1,1}$	$d_{2,2}$	$d_{3,3}$	$d_{4,4}$	$d_{5,5}$
	0	∞	2	10	5	6

$$d_{i,s} = \min \begin{cases} 0 & \text{if } s = 0, \\ \infty & \text{if } 1 \le s < t \text{ or } i < s, \\ d_{i-1,s} & \\ Match(A_{i-t+1..i}, & \text{if } s \ge t, \\ \quad d_{i-t,s-t} + t) & \\ Match(A_{i-t..i}, & \text{if } s \ge t+1, \\ \quad d_{i-t-1,s-t-1} + t + 1) & \\ \vdots & \\ Match(A_{i-2t+2..i}, & \text{if } s \ge 2t-1. \\ d_{i-2t+1,s-2t+1} + 2t - 1) & \end{cases} \tag{4}$$

In Equation 4, consideration of each common g-length substring for $t \le g \le 2t-1$ is enough for this problem.

Lemma 2. *$Match(A_{i+1..i+2t}, d_{i,s} + 2t) \ge Match(A_{i+t+1..i+2t}, j)$, where $j = Match(A_{i+1..i+t}, d_{i,s} + t) + t$.*

Proof. Suppose $Match(A_{i+1..i+2t}, d_{i,s} + 2t) = j_2$, which means that $A_{i+1..i+2t} = B_{j_2-2t+1..j_2}$ is a $2t$-match. It also implies that $A_{i+1..i+t} = B_{j_2-2t+1..j_2-t}$ and $Match(A_{i+1..i+t}, d_{i,s}+t) \le j_2 - t$ because there is possibly a substring $B_{j_1-t+1..j_1} = A_{i+1..i+t}$ and $j_1 < j_2 - t$. There is also possibly a substring $B_{j_3-t+1..j_3}$, $B_{j_3-t+1..j_3} = A_{i+t+1..i+2t}$ and $j_1 + t \le j_3 \le j_2$, such that $Match(A_{i+t+1..i+2t}, Match(A_{i+1..i+t}, d_{i,s}+t)+t) = Match(A_{i+t+1..i+2t}, j_1 + t) = j_3 \le j_2$. □

The pseudo code of our LCS_{t+} algorithm is presented in Algorithm 2. The time complexity is $\mathrm{O}(t(n+m) + tL(m-L)\log n)$ if we use binary search in lines 10 and 11. We need to find the common substrings of lengths $t, t+1, \cdots, 2t-1$, and it requires $\mathrm{O}(t(n+m))$ time to accomplish the job with their group names. It needs t times of searching in lines 9 through 12 and the binary search needs $\mathrm{O}(\log n)$ time, so the time complexity of each round is $\mathrm{O}(tL\log n)$.

To avoid the redundant matches in lines 9 through 12, we use the concept of match pairs proposed by Pavetić *et al.* [12]. We scan all t-match pairs which can be extended and record their extensions. In the following, we illustrate our algorithm with $A =$ `actaacg`, $B =$ `ctgacactcg` and t

 ISBN/978-957-437-895-1(PDF)

Algorithm 2 A diagonal algorithm for computing the LCS_{t+} length.

Input: Two sequences $A = a_1a_2a_3\cdots a_m$ and $B = b_1b_2b_3\cdots b_n$, where $m \le n$, and the length t of the substring.
Output: The length of $LCS_{t+}(A, B)$

```
L ← 0                                   ▷ initial length of LCS_{t+}(A, B)
for r = 1 → m do                        ▷ round r
    if L > m − r then
        break ▷ no longer LCS_{t+} can be found
    d_{r−1,0} ← 0, s ← t                ▷ initialization
    while true do
        i ← r + s − 1
        for c = t → 2t − 1 do
            if s ≥ c then
                Find the smallest j such that d_{i−c,s−c} + c ≤ j ≤ n and A_{i−c+1..i} = B_{j−c+1..j}
                d_{i,s} ← min(d_{i−1,s}, j)
        if d_{i,s} < ∞ then
            L ← max(L, s)
        else
            break
        s ← s + t
return L
```

= 2. The result is shown in Table 4. The LCS_{t+} answer of this example is $A_{1..3}A_{6..7} = B_{6..8}B_{9..10}$ = actcg, with length 5.

Definition 3. *$MatchPairs[i]$ is a set consisting of all ending position indices of B that match $A_{i-t+1..i}$ with length t. That is, $MatchPairs[i] = \{j|\ A_{i-t+1..i} = B_{j-t+1..j},\ t \le j \le n.\}$*

Definition 4. *$Extend[i]$ is a set consisting of all possibly extendable position indices of B that extend from $A_{i-t..i-1}$ with length t in the same round. That is, $Extend[i] = \{j|\ A_{i-t..i-1} = B_{j-t..j-1},\ t+1 \le j \le n.\}$*

Table 4: An example for the LCS_{t+} algorithm with A = actaacg, B = ctgacactcg and $t = 2$. s denotes the LCS_{t+} length of A and B, and r denotes the number of the round.

r \ s	**0**	**1**	**2**	**3**	**4**	**5**
1	$d_{0,0}$	$d_{1,1}$	$d_{2,2}$	$d_{3,3}$	$d_{4,4}$	$d_{5,5}$
	0	∞	5	8	∞	∞
2	$d_{1,0}$	$d_{2,1}$	$d_{3,2}$	$d_{4,3}$	$d_{5,4}$	$d_{6,5}$
	0	∞	2	8	∞	∞
3	$d_{2,0}$	$d_{3,1}$	$d_{4,2}$	$d_{5,3}$	$d_{6,4}$	$d_{7,5}$
	0	∞	2	8	5	10

In the first round, we start from $A_{1..2}$ = ac. We have $MatchPairs[2] = \{5, 7\}$. In other words, $A_{1..2} = B_{4..5} = B_{6..7}$. We also have $d_{0,0} = 0$. Then, we get $d_{2,2} = 5$ since it is more extendable by using fewer characters of B, and we remove 5 and 7 from $MatchPairs[2]$. Then, $MatchPairs[3] = \{8\}$, which can be extended from $A_{1..2} = B_{6..7}$, so 8 is added into $Extend[3]$. Next, $A_{2..3}$ = ct, $MatchPairs[3] = \{8\}$, and $d_{1,1} = \infty$, so $d_{3,3}$ cannot be extended from $d_{1,1}$ with $A_{2..3} = B_{7..8}$. $Extend[3] = 8$ means that $A_{1..3} = B_{6..8}$ can be extended from $A_{1..2} = B_{6..7}$, so $d_{3,3} = 8$. No extension can be added into $Extend[4]$. For $A_{3..4}$ = ta, there is no extendable common substring in $MatchPairs[4] = \emptyset$ from $d_{2,2} = 5$, so $d_{4,4} = \infty$. For $A_{4..5}$ = aa, there is no extendable common substring in $MatchPairs[5] = \emptyset$ from $d_{3,3} = 8$, so $d_{5,5} = \infty$. Thus, the first round stops with the answer length 3.

In the second round, we start from $A_{2..3}$ = ct, $MatchPairs[3] = \{2\}$, and $d_{1,0} = 0$, so $d_{3,2} = 2$ since it uses fewer characters of B than $d_{2,2} = 5$. Then we remove 2 from $MatchPairs[3]$. Next $MatchPairs[4] = \emptyset$ and $MatchPairs[5] = \emptyset$, so $d_{4,3} = 8$ copied from $d_{3,3}$ and $d_{5,4} = \infty$ copied from $d_{4,4}$. For $A_{5..6}$, we find that $MatchPairs[6] = \{5, 7\}$, but they cannot be extended from $d_{4,3} = 8$, so $d_{6,5} = \infty$ copied from $d_{5,5}$.Thus, this round stops.

In the third round, we start from $A_{3..4}$ = ta. We have $MatchPairs[4] = \emptyset$ and $MatchPairs[5 = \emptyset]$, so $d_{4,2} = 2$ copied from $d_{3,2} = 2$ and $d_{5,3} = 8$ copied from $d_{4,3} = 8$. Next, $A_{5..6}$ = ac, $MatchPairs[6] = \{5, 7\}$, and $d_{4,2} = 2$, so $d_{6,4} = 5$ since it is better than $d_{5,4} = \infty$. Then, we remove 5 and 7 from $MatchPairs[6]$. And we find that there is no extension for 5 and 7 in $MatchPairs[7]$, so no element is added into $Extend[7]$. For $A_{6..7}$ = cg, we find that $MatchPairs[7] = \{10\}$ and $d_{5,3} = 8$, so $d_{7,5} = 10$ and remove 10 from $MatchPairs[7]$. This round stops. The algorithm terminates since we cannot find a solution better than length 5 in $A_{4..7}$ for the next round.

The pseudo code of our LCS_{t+} algorithm with match pairs is presented in Algorithm 3. In each round r, we need to find the extendable common t-length substrings for $\mathrm{O}(L)$ times, and there are at most $m-L$ rounds, so it needs $\mathrm{O}(L(m-L))$ time. Each t-match pair is removed after it has been extended, and there are at most R t-match pairs are removed, where R is the number of t-match pairs of A and B, so the total time complexity is $\mathrm{O}(L(m-L)+R)$.

Algorithm 3 Computing the LCS_{t+} length with match pairs.

Input: Substrings length parameter t and sequences $A = a_1a_2a_3\cdots a_m$ and $B = b_1b_2b_3\cdots b_n$, where $m \le n$.
Output: Length of $LCS_{t+}(A, B)$

```
L ← 0
for r = 1 → m do                                   ▷ round r
    if r > m − L then
        break
    s ← t
    while s ≤ L + t do
        i ← r + s − 1
        d_{i,s} ← d_{i−1,s}
        for each j with j ≥ d_{i−t,s−t} + t and j ∈ MatchPairs[i] do
            d_{i,s} ← min(d_{i,s}, j)
            if j + 1 ∈ MatchPairs[i + 1] then
                Insert j + 1 into Extend[i + 1]
            Remove j from MatchPairs[i]
        for each j ∈ Extend[i] do
            d_{i,s} ← min(d_{i,s}, j)
            if j + 1 ∈ MatchPairs[i + 1] then
                Insert j + 1 into Extend[i + 1]
            Remove j from MatchPairs[i] and Extend[i]
        if d_{i,s} < ∞ then
            L ← max(L, s)
        s ← s + 1
return L
```

5 Experimental Results

In this Section, some simulation experiments of our algorithms and previously published algorithms are performed on pseudorandom sequences with different lengths, alphabet set sizes and t. The execution time is the average of 100 times in each experiment. These algorithms are implemented by Python 3.8, and they are tested on several computers with 64-bit Windows 10 OS, CPU clock rate of 3.70GHZ (Intel(R) Core(TM) i7-8700K CPU) and 8 GB of RAM.

5.1 The Experimental Results of LCS_t

The pseudorandom datasets are generated with $|A| \in \{1000, 2000, 5000\}$, $|B| \in \{1000, 2000, 5000\}$, $|\Sigma| \in \{2, 4, 8, 20\}$, $t \in \{2, 3, 5\}$, and various similarities $SI \in \{0.1, 0.2, \ldots, 0.9\}$, where $SI = \frac{L}{\min(|A|,|B|)}$.

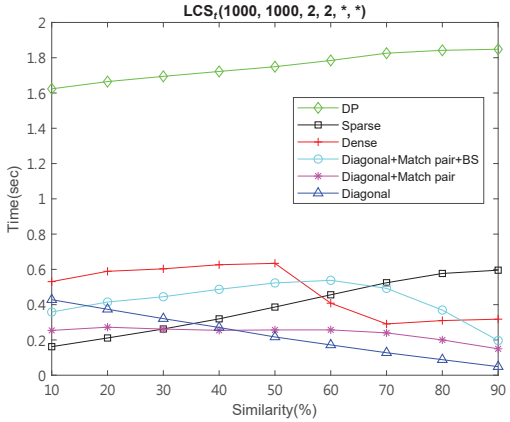

Figure 1: The average execution time for the LCS_t algorithm, where $|A| = 1000$, $|B| = 1000$, $t = 2$ and $|\Sigma| = 2$.

In our experiments, we compare the performances of the following LCS_t algorithms:

- **DP**: The DP algorithm of Benson *et al.* [1],
- **Dense**: The dense DP algorithm of Deorowicz and Grabowski [6],
- **Sparse**: The sparse DP algorithm proposed by Pavetić *et al.* [12],
- **Diagonal**: Our diagonal algorithm presented in this paper,
- **Diagonal+Match Pair**: Our diagonal algorithm with match pairs presented in this paper,
- **Diagonal+Match Pair+BS**: Our diagonal algorithm with match pairs and binary search presented in this paper.

We do not implement their sparse algorithm of Deorowicz and Grabowski [6] since the time complexity is worse than the sparse algorithm of Pavetić *et al.* [12]. Due to the implementation hardness of the vEB tree and the method of Four Russians, we do not implement those algorithms of Deorowicz and Grabowski [6] to avoid the wrong experimental results.

Figure 1 shows the average execution time of various algorithms. We use a 6-tuple ($|A|$, $|B|$, t, $|\Sigma|$, *similarity*, *algo*) to represent the parameters in each performance chart. For example, in Figure 1, $(1000, 1000, 2, 2, *, *)$ means that $|A| = m = 1000$, $|B| = n = 1000$, $t = 2$, alphabet set size $|\Sigma| = 2$, the first "$*$" is a wildcard representing all

 ISBN/978-957-437-895-1(PDF)

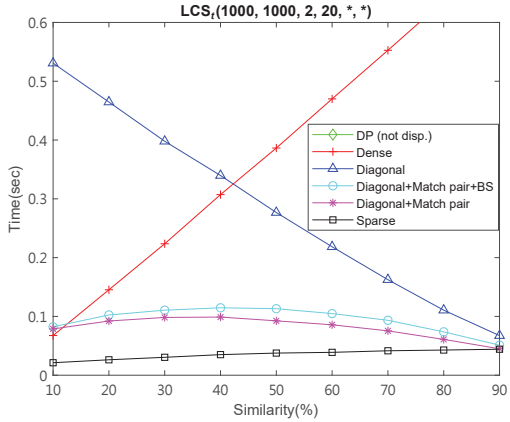

Figure 2: The average execution time for the LCS_t algorithms, where $|A| = 1000$, $|B| = 1000$, $t = 2$ and $|\Sigma| = 20$.

similarities of LCS_t and the second "$*$" is a wild-card representing all possible algorithms. Figures 2, 3 and 4 illustrate the average execution times for various lengths of A and B, t and alphabet set size $|\Sigma|$.

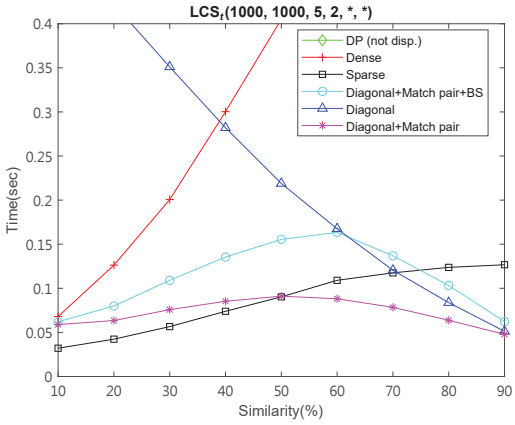

Figure 3: The average execution time for the LCS_t algorithms, where $|A| = 1000$, $|B| = 1000$, $t = 5$ and $|\Sigma| = 2$.

The DP algorithm of Benson *et al.* takes much more time than other algorithms in all cases. The sparse algorithm of Pavetić *et al.* takes less time when the number of t-match pairs decreases, so its performance is efficient when t and $|\Sigma|$ are large. The dense algorithm of Deorowicz and Grabowski uses a special data structure to handle various numbers of t-match pairs, so it takes a variety of time with different similarities. It is obvious that our diagonal algorithm has better performance with higher similarity. When the number

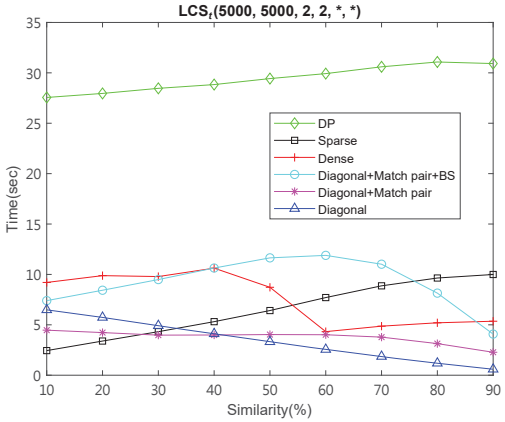

Figure 4: The average execution time for the LCS_t algorithms, where $|A| = 5000$, $|B| = 5000$, $t = 2$ and $|\Sigma| = 2$.

of t-match pairs decreases, the diagonal algorithm with match pairs has better performance than diagonal LCS_t algorithm without match pairs and binary search.

5.2 The Experimental Results of LCS_{t+}

Since we need to consider the extension of each match pair, our LCS_{t+} algorithm is more complicated than our LCS_t algorithms, and it takes more time to handle match pairs. There is a benefit for the method based on t-match pairs that the number R of t-match pairs decreases quickly by the increase of Σ and t since $R = \frac{mn}{|\Sigma|^t}$ in random strings.

The pseudorandom datasets are generated with $|A| \in \{1000, 2000, 5000\}$, $|B| \in \{1000, 2000, 5000\}$, $|\Sigma| \in \{2, 4, 8, 20\}$, $t \in \{2, 3, 5\}$, and various similarities $SI \in \{0.1, 0.2, \ldots, 0.9\}$, where $SI = \frac{L}{\min(|A|,|B|)}$.

In our experiments, we compare the performances of the following LCS_{t+} algorithms:

- **DP wtih O**(tmn): The DP algorithm of Benson *et al.* [1],
- **DP with O**(mn): The DP algorithm of Ueki *et al.* [16],
- **Sparse**: The sparse DP algorithm proposed by Pavetić *et al.* [12],
- **Diagonal**: Our algorithm presented in this paper.

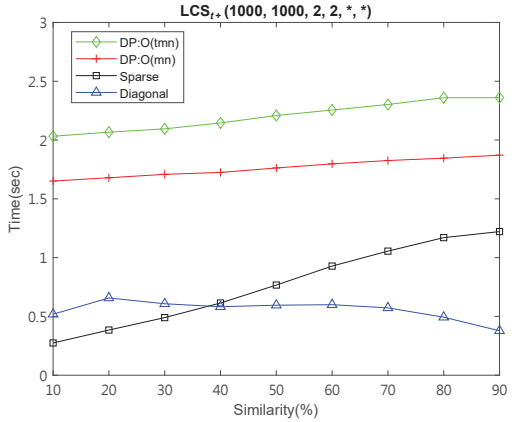

Figure 5: The average execution time for the LCS_{t+} algorithms, where $|A| = 1000$, $|B| = 1000$, $t = 2$ and $|\Sigma| = 2$.

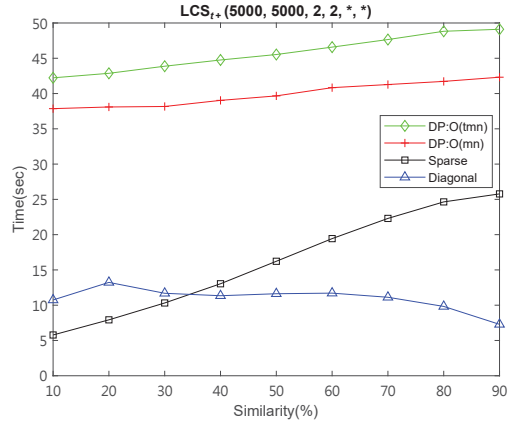

Figure 6: The average execution time for the LCS_{t+} algorithms, where $|A| = 5000$, $|B| = 5000$, $t = 2$ and $|\Sigma| = 2$.

Figures 5 through 8 shows the average execution time of these algorithms for various lengths of A and B, t and alphabet set size $|\Sigma|$. Our diagonal LCS_{t+} algorithm has better performance with higher similarity and larger number of t-match pairs. When the number of t-match pairs and the similarity decrease, the sparse algorithm of Pavetić *et al.* has better performance.

6 Conclusion

In this paper, we solve the LCS_t problem by the diagonal method with three different implementation ways for searching matching substrings. The time complexities are $\mathrm{O}(n(m-L))$, $\mathrm{O}(n+l(m-L)+R)$, and $\mathrm{O}(n+l(m-L)\log n)$, where m, n, l

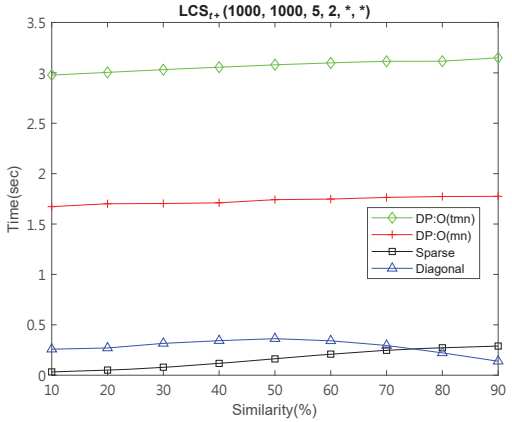

Figure 7: The average execution time for the LCS_{t+} algorithms, where $|A| = 1000$, $|B| = 1000$, $t = 5$ and $|\Sigma| = 2$.

and L denote the lengths of the input sequences A, B, the number of common t-length substrings in the answer, and the length of the LCS_t answer, respectively; R denotes the number of t-match pairs between A and B. We also propose a diagonal algorithm for solving the LCS_{t+} problem in $\mathrm{O}(n+L(m-L)+R)$ time. We generate some pseudorandom datasets to test our algorithms. Experimental results show that our algorithms are faster if A and B are very similar and the number of t-match pairs are large.

Our goal in the future is to design a better method to search the extendable common substrings of the diagonal algorithm for solving the LCS_t and LCS_{t+} problems. We hope the time complexity could be reduced to $\mathrm{O}(n+l(m-L))$. We also consider the upper bound of the length of common substrings could be added to as a continuity constraint, such as the motif finding problem [14].

References

[1] G. Benson, A. Levy, S. Maimoni, D. Noifeld, and B. R. Shalom, "LCSk a refined similarity measure," *Theoretical Computer Science*, Vol. 368, pp. 11–26, 2016.

[2] G. Benson, A. Levy, and B. R. Shalom, "Longest common subsequence in k length substrings," *Proceedings of the 6th International Conference on Similarity Search and Applications*, Vol. 8199, pp. 257–265, 2013.

[3] H. T. Chan, H. T. Chiu, C. B. Yang, and Y. H. Peng, "The generalized definitions of

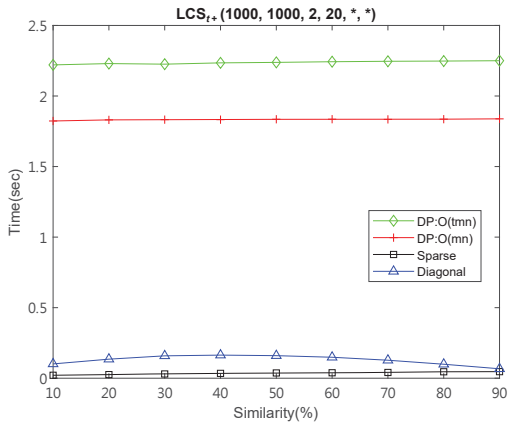

Figure 8: The average execution time for the LCS$_{t+}$ algorithms, where $|A| = 1000$, $|B| = 1000$, $t = 2$ and $|\Sigma| = 20$.

the two-dimensional largest common substructure problems," *Algorithmica*, Vol. 82, pp. 2039–2062, 2020.

[4] A. Danek and S. Deorowicz, "Bit-parallel algorithm for the block variant of the merged longest common subsequence problem," *Advances in Intelligent Systems and Computing*, Vol. 242, pp. 173–181, 2014.

[5] S. Deorowicz and A. Danek, "Bit-parallel algorithms for the merged longest common subsequence problem," *International Journal of Foundations of Computer Science*, Vol. 24, pp. 1281–1298, 2013.

[6] S. Deorowicz and S. Grabowski, "Efficient algorithms for the longest common subsequence in k-length substrings," *Information Processing Letters*, Vol. 114, pp. 634–638, 2014.

[7] S. H. Hung, C. B. Yang, and K. S. Huang, "A diagonal-based algorithm for the constrained longest common subsequence problem," *Proceedings of the 23rd International Computer Symposium 2018 (ICS 2018)*, Vol. 1013, Yunlin, Taiwan, pp. 425–432, 2018.

[8] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)* (A. Amir, ed.), Berlin, Germany, pp. 181–192, Springer Berlin Heidelberg, 2001.

[9] S. F. Lo, K. T. Tseng, C. B. Yang, and K. S. Huang, "A diagonal-based algorithm for the longest common increasing subsequence problem," *Theoretical Computer Science*, Vol. 815, pp. 69–78, 2020.

[10] 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.

[11] S. 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.

[12] F. Pavetić, I. Katanić, G. Matula, G. Žužić, and M. Šikić, "Fast and simple algorithms for computing both LCSk and LCSk+," *CoRR, abs/1705.07279*, 2018.

[13] A. M. Rahman and M. S. Rahman, "Effective sparse dynamic programming algorithms for merged and block merged LCS problems," *Journal of Computers*, Vol. 9(8), pp. 1743–1754, 2014.

[14] S. D. Sayeed, M. S. Rahman, and A. Rahman, "On multiple longest common subsequence and common motifs with gaps (extended abstract)," *Proceedings of the 12th Internatioanl Workshop on Algorithms and Computation (WALCOM 2018). Lecture Notes in Computer Science*, Vol. 10755, pp. 207–215, 2018.

[15] 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.

[16] Y. Ueki, Diptarama, M. Kurihara, Y. Matsuoka, K. Narisawa, R. Yoshinaka, H. Bannai, S. Inenaga, and A. Shinohara, "Longest common subsequence in at least k length order-isomorphic substrings," *Proceedings of the 43rd International Conference on Current Trends in Theory and Practice of Computer Science. Lecture Notes in Computer Science*, Vol. 10139, pp. 364–374, 2017.