Theoretical Computer Science 708 (2018) 75–90

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Efficient merged longest common subsequence algorithms for similar sequences[☆,☆☆]

Kuo-Tsung Tseng[a], De-Sheng Chan[b], Chang-Biau Yang[b,*], Shou-Fu Lo[b]

[a] *Department of Shipping and Transportation Management, National Kaohsiung Marine University, Kaohsiung, Taiwan*
[b] *Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan*

A R T I C L E   I N F O

Article history:
Received 7 March 2017
Received in revised form 16 October 2017
Accepted 27 October 2017
Available online 9 November 2017
Communicated by T. Calamoneri

Keywords:
Longest Common Subsequence (LCS)
Merged Longest Common Subsequence (MLCS)
Sequence similarity

A B S T R A C T

Given a pair of merging sequences A, B and a target sequence T, the merged longest common subsequence (MLCS) problem is to find out the longest common subsequence (LCS) between sequences $E(A, B)$ and T, where $E(A, B)$ is obtained from merging two subsequences of A and B. In this paper, we first propose an algorithm for solving the MLCS problem in $O(n|\Sigma| + (r - L + 1)Lm)$ time and $O(n|\Sigma| + Lm)$ space, where r and L denote the lengths of T and MLCS, respectively, and m and n denote the shorter and longer lengths of A and B, respectively. From the time complexity, it is clear that our algorithm is very efficient when T and $E(A, B)$ are very similar. With slight modification, our algorithm can also solve another merged LCS problem variant, the block-merged LCS (BMLCS) problem, in $O(n|\Sigma|+(r-L+1)L\delta)$ time and $O(n|\Sigma|+L\delta)$ space, where δ denotes the larger number of blocks of A and B. Experimental results show that our algorithms are faster than other previously published MLCS and BMLCS algorithms for sequences with high similarities. The source codes and datasets for experiments can be found on our web site http://par.cse.nsysu.edu.tw/~mlcs/ [20].

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

It is essential in many applications to measure the similarity of two sequences, such as computational biology, pattern matching, plagiarism detection, voice recognition, and so on. The most well-known methods for measuring the sequence similarity in computer science are the algorithms for the longest common subsequence (LCS) problem [1–4,6,7,12,19,32–35] and the edit distance problem [22–25,28].

Given two sequences $A = a_1a_2a_3 \dots a_m$ and $B = b_1b_2b_3 \dots b_n$, the *longest common subsequence* (LCS) problem is to find the longest sequence which is a subsequence of both A and B. Without loss of generality, it is assumed that $m \le n$. Here, a subsequence of A or B can be obtained by deleting an arbitrary number of characters at arbitrary positions of A or B. The LCS of A and B is denoted by $LCS(A, B)$. For example, consider $A = accgt$ and $B = tagct$. $LCS(A, B)$ is agt or act.

The LCS problem has been extensively studied since 1970. In 1974, Wagner and Fischer proposed a dynamic programming (DP) algorithm with $O(mn)$ time and $O(mn)$ space [38]. After that, Hirschberg improved the space complexity from

[☆] This research work was partially supported by the Ministry of Science and Technology, Taiwan under contract MOST 104-2221-E-110-018-MY3.
[☆☆] A preliminary version of this paper was presented at the 20th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2016), 5–8 July 2016, Orlando, Florida, USA [34].
[*] Corresponding author.
E-mail address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

https://doi.org/10.1016/j.tcs.2017.10.027
0304-3975/© 2017 Elsevier B.V. All rights reserved.

Table 1
Time and space complexities of the MLCS and BMLCS algorithms. Notations: $|A| = m, |B| = n$, $m \le n, |T| = r$; α, β: number of blocks in A and B, respectively; $\delta = \max\{\alpha, \beta\}$; In Grabowski [11], $r = \Omega(n^c)$ for some constant $c > 0$; w: word size of a computer; L: length of the MLCS or BMLCS answer; $\mathcal{R}$, $\mathcal{P}$: numbers of matching pairs of A, T and B, T respectively.

Authors	MLCS problem	BMLCS problem
Time complexities		
Huang et al. [15]	$O(rnm)$	$O(rn\delta)$
		$O(r^2 + r\delta^2)$
Peng et al. [29]	$O(Lrm)$	$O(Lr\delta)$
Deorowicz and Danek [9,8]	$O(\lceil r/w \rceil mn \log w)$	$O((m\beta + n\alpha)\lceil r/w \rceil + \alpha\beta\lceil r/w \rceil)$
Rahman and Rahman [31]	$O((\mathcal{R}m + \mathcal{P}n)\log\log m)$	$O((\mathcal{R}\beta m + \mathcal{P}\alpha n)\log\log m)$
Grabowski [11]	$O(mnr/\log^{1.5} n)$	
This paper	$O(n\|\Sigma\| + (r - L + 1)Lm)$	$O(n\|\Sigma\| + (r - L + 1)L\delta)$
Space complexities		
Huang et al. [15]	$O(mn)$	$O(r^2 + r\delta^2)$
Peng et al. [29]	$O(n + Lm)$	$O(n + L\delta)$
Rahman and Rahman [31]	$\theta(\max\{rn, m\})$	$\theta(\max\{\mathcal{R} + \mathcal{P}, m\})$
This paper	$O(n\|\Sigma\| + Lm)$	$O(n\|\Sigma\| + L\delta)$

$O(mn)$ to linear with the divide-and-conquer approach [13]. In 1977, Hunt and Szymanski proposed an algorithm considering only the matching characters of the two sequences with $O(n + (R + m)\log m)$ time [17], where R denotes the total number of matching pairs of the two sequences. R should be small if $|\Sigma|$, the size of the alphabet set, is large, and thus the algorithm is efficient. On the contrary, if $|\Sigma|$ is small, the efficiency of their algorithm is lowered dramatically and the time complexity of the worst case becomes $O(mn \log m)$. Theoretically, their algorithm can be improved to $O(n + (R + m)\log\log m)$ time with the van Emde Boas tree [17]. In 1982, Nakatsu et al. utilized the diagonal concept to solve the LCS problem with $O(n(m - L))$ time [26], where L denotes the length of $LCS(A, B)$, and it is more efficient for similar sequences.

The *merged LCS* (MLCS) problem is a variant of the LCS problem. Given a pair of merging sequences $A = a_1a_2a_3 \dots a_m$ and $B = b_1b_2b_3 \dots b_n$, and a target sequence $T = t_1t_2t_3 \dots t_r$, the MLCS problem is to find the LCS of the merged sequence $E(A, B)$ and the target sequence T. $E(A, B)$ is obtained by merging two subsequences of A and B arbitrarily with preserving their orders. Formally, we have $E(A, B) = e_1e_2e_3 \dots e_{|E(A,B)|}$ such that its subsequence $P = e_{i_1}e_{i_2} \dots e_{i_k}$, where $1 \le i_1 < i_2 < \dots < i_k \le |E(A, B)|$, is a subsequence of A, and the remaining subsequence of $E(A, B)$ after removing P is a subsequence of B. The answer of the MLCS problem is denoted by $MLCS(A, B, T)$. Without loss of generality, we assume that $m \le n$.

For example, consider sequences $A = a_1a_2a_3 = acg$, $B = b_1b_2b_3b_4 = ccca$ and a target sequence $T = t_1t_2t_3t_4t_5t_6 = actcgc$. There are many ways to construct the merged sequence $E(A, B)$, such as $E_1(A, B) = a_1b_1b_2b_3a_2a_3b_4 = accccga$, $E_2(A, B) = b_1b_2b_3b_4a_1a_2a_3 = cccaacg$ and $E_3(A, B) = a_1b_1b_2a_2a_3b_3b_4 = acccgca$. In this example, $LCS(E_3(A, B), T) = accgc$ is the answer of the MLCS problem.

The MLCS problem was first defined by Huang et al. in 2008 [15] and they proposed a dynamic programming algorithm with $O(mnr)$ time algorithm to solve it. In 2010, Peng et al. proposed an algorithm for solving the MLCS problem in $O(Lnr)$ time [29], where L denotes the length of $MLCS(A, B, T)$. If L is small, their algorithm performs well. Based on Huang's [15] algorithm, Deorowicz and Danek proposed the bit-parallel algorithm for solving MLCS problem in $O(\lceil r/w \rceil mn \log w)$ time [9], where w denotes the word size of a computer and $w \ge \log(\max(n, r))$. In 2014, Rahman and Rahman proposed an algorithm with $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$ time, which uses the *bounded heap* data structure to reduce the time complexity [31], where $\mathcal{R}$ and $\mathcal{P}$ denote the total numbers of matching pairs in A, T and B, T respectively. In 2016, Grabowski [11] proved that the MLCS problem can be computed with $O(mnr/\log^{1.5} n)$ time in the worst case, when $r = |T| = \Omega(n^c)$ for some constant $c > 0$.

The *block-merged LCS* (BMLCS) problem [15] is a variant of the MLCS problem with additional block constrains for merging sequences A and B. In the BMLCS problem, the input sequences A and B are divided into α and β blocks, respectively, where $\delta = \max\{\alpha, \beta\}$. The subsequence extracted from one block should be contained in BMLCS as a substring. Several algorithms have been proposed for solving the BMLCS problem [8,15,29,31]. In general, these algorithms were modified and extended from the MLCS algorithms with the same concept. Worth to be mentioned, Huang et al. [15] invoked the S-table technique [21] to speed up the computational time in solving the BMLCS problem. Table 1 shows the time and space complexities of known algorithms [8,9,15,29,31] for solving these problems. Please note that our notations may be different from those in the original papers.

There are some applications for the MLCS and BMLCS problems. For example, the voice recognition, we have a complete voice data stream without noise and two or more incomplete voice data streams with different noises. We may recognize the voice command if we are able to compute the similarities among those voice streams. Another example is the gene loss during genome duplication and long-time evolution, coming from biology. Two species may come from the same ancestor and they may own different parts of their ancestor. Such an example can be found between two yeast species *Saccharomyces cerevisiae* and *Kluyveromyces waltii* [18]. Kellis et al. [18] obtained the support for the whole-genome duplication (WGD) or two rounds of gene duplication (2R) hypothesis [14,27] after detecting the doubly conserved synteny (DCS) blocks of the above two yeast species.

From the time complexities of the algorithms proposed by Peng et al. [29], it is obvious that the shorter MLCS or BMLCS length, the better the performance is. Both their MLCS and BMLCS algorithms are efficient if the merged sequence and the target sequence are dissimilar, which means that L is small. On the contrary, if the given sequences are similar, their algorithms cannot perform so well. Therefore, in this paper, we focus on the situation for similar sequences. Inspired by the LCS algorithm proposed by Nakatsu et al. [26], we propose efficient algorithms to solve the *merged LCS* (MLCS) and *block-merged LCS* (BMLCS) problems with $O(n|\Sigma|+(r-L+1)Lm)$ time and $O(n|\Sigma|+(r-L+1)L\delta)$ time, respectively. When L is large (i.e., the input sequences are similar), our algorithms are very efficient.

The rest of this paper is organized as follows. In Section 2, some preliminaries and definitions of the MLCS and BMLCS problems are introduced. In Sections 3 and 4, we present our MLCS and BMLCS algorithms, respectively. Then, in Section 5, the comparisons of the execution time with previously published algorithms are given. Finally, we give our conclusions and future works in Section 6.

2. Preliminaries

In this section, we present some notations and describe the dynamic programming (DP) methods for solving the *merged LCS* (MLCS) and *block-merged LCS* (BMLCS) problems [15].

Let a sequence $S=s_1s_2s_3\dots s_{|S|}$ be a sequence of characters over a finite alphabet set Σ. The notations are listed as follows:

- $|S|$: the length of S.
- s_i: the ith character of S.
- $S_{i..j}$: the substring of S from position i to position j, where $S_{i..j}=\varepsilon$ if $j>i$, and ε denotes an empty string.

In the MLCS problem, let $H(i,j,k)$ denote the length of $MLCS$ $(A_{1..i}, B_{1..j}, T_{1..k})$. The DP formula for solving the MLCS problem proposed by Huang et al. [15] is given as follows.

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

with the boundary conditions:

$$H(i,0,k)=\text{length of } LCS(A_{1..i},T_{1..k}) \text{ and } H(0,j,k)=\text{length of } LCS(B_{1..j},T_{1..k}).$$

In 2013, Deorowicz and Danek [9] found that there is a bit of error in the original DP formula of Huang et al. [15], which was originally written as “if $a_i\neq t_k$ and $b_j\neq t_k$” in the third condition. They gave a counterexample with $A=dda$, $B=bac$ and $T=aba$. They also suggested the correction with keeping the last three functions, but removing the third condition. Here, we give another correction, presented in Eq. (1), which corrects the error from “$a_i\neq t_k$ and $b_j\neq t_k$” to “$a_i\neq t_k$ or $b_j\neq t_k$”. For $a_i=t_k$ and $b_j=t_k$, the solution must come from the first or the second function, since we cannot ignore the length increase from t_k when t_k is identical to both a_i and b_j. On the other hand, the solution may come from the last three functions (third condition) when $a_i\neq t_k$ or $b_j\neq t_k$ (i.e., negation of “a_i and b_j”).

As for the BMLCS problem, we are given a pair of merging block sequences $A=a_1a_2a_3\dots a_m=A_1A_2A_3\dots A_\alpha$ and $B=b_1b_2b_3\dots b_n=B_1B_2B_3\dots B_\beta$ and a target sequence $T=t_1t_2t_3\dots t_r$, where α and β denote the numbers of blocks in sequences A and B, respectively. The BMLCS problem is to find the LCS between the blocked merged sequence $E^b(A,B)$ and a target sequence T, denoted by $BMLCS(A,B,T)$. Here, the blocked merged sequence $E^b(A,B)$ can be obtained by merging block sequences A and B in their original orders.

For example, suppose $A=A_1A_2=a\#cg\#$, $B=B_1B_2=ccc\#a\#$, and $T=actcgc$, where the character # denotes the *end of a block* (EOB). EOB is nothing but a virtual dividing character, not a real character. In this example, both A and B are divided into two blocks. The blocked merged sequence $E^b(A,B)$ can be obtained by examining all possible permutations between block sequences A and B, such as $E^b_1(A,B)=A_1A_2B_1B_2=acgccca$, $E^b_2(A,B)=A_1B_1A_2B_2=accccga$, $E^b_3(A,B)=A_1B_1B_2A_2=acccacg$, $E^b_4(A,B)=B_1B_2A_1A_2=cccaacg$, $E^b_5(A,B)=B_1A_1B_2A_2=cccaacg$ and $E^b_6(A,B)=B_1A_1A_2B_2=cccacga$. The answer of $BMLCS(A,B,T)$ is $LCS(E^b_2(A,B),T)$ or $LCS(E^b_3(A,B),T)$, whose content is $accg$.

Let $H_B(i,j,k)$ denote the length of $BMLCS(A_{1..i},B_{1..j},T_{1..k})$. The DP formula for solving the BMLCS problem proposed by Huang et al. [15] is given as follows.

$$H_B(i,j,k) = \max \begin{cases} \max \begin{cases} H_B(i-1,j,k-1)+1 & \text{if } a_i = t_k \\ H_B(i-1,j,k) & \text{if } a_i \neq t_k \\ H_B(i,j-1,k-1)+1 & \text{if } b_j = t_k \\ H_B(i,j-1,k) & \text{if } b_j \neq t_k \\ H_B(i,j,k-1) & \text{if } a_i \neq t_k \text{ or } b_j \neq t_k \end{cases} & \begin{array}{l}\text{if } a_i = EOB \\ \text{or } b_j = EOB\end{array} \\ \max \begin{cases} H_B(i-1,j,k-1)+1 & \text{if } a_i = t_k \\ H_B(i-1,j,k) & \text{if } a_i \neq t_k \\ H_B(i,j,k-1) & \text{if } a_i \neq t_k \end{cases} & \begin{array}{l}\text{if } a_i \neq EOB \\ \text{and } b_j = EOB\end{array} \\ \max \begin{cases} H_B(i,j-1,k-1)+1 & \text{if } b_j = t_k \\ H_B(i,j-1,k) & \text{if } b_j \neq t_k \\ H_B(i,j,k-1) & \text{if } b_j \neq t_k \end{cases} & \begin{array}{l}\text{if } a_i = EOB \\ \text{and } b_j \neq EOB\end{array} \end{cases} \tag{2}$$

with the boundary conditions:

$$H_B(i,0,k) = \text{length of } LCS(A_{1..i}, T_{1..k}) \text{ and } H_B(0,j,k) = \text{length of } LCS(B_{1..j}, T_{1..k}).$$

The time complexity of the above algorithm is $O(mn\delta)$. Furthermore, Huang et al. [15] utilized the S-table technique [21] to reduce the required time to $O(n^2 + n\delta^2)$ for the block-merged LCS problem.

Nakatsu et al. proposed an $O(n(m-L))$-time algorithm for solving the LCS problem [26]. It can be seen that their algorithm is very efficient when L is large, i.e. the two input sequences are similar. In fact, their algorithm is a method based on the diagonal scheme, instead of the domination concept.

Given $A_{1..i}$, let $d_{i,s}$ denote the smallest index j of B such that $|LCS(A_{1..i}, B_{1..j})| = s$. Nakatsu et al. solved the LCS problem with $d_{i,s}$ as follows [26].

$$d_{i,s} = \begin{cases} \min(j, d_{i-1,s}) & \text{if there exists the smallest } j \text{ such that } a_i = b_j, \\ & \text{where } j > d_{i-1,s-1} \text{ when } s \geq 2. \\ d_{i-1,s} & \text{if there is no such } j. \end{cases} \tag{3}$$

3. Our merged LCS algorithm

Our MLCS algorithm is inspired by the LCS algorithm of Nakatsu et al. [26], which is more efficient with highly similar sequences. Our algorithm is a diagonal-based method, and utilizes the domination concept to reduce the unnecessary computation.

First, we give the definition of the *minimum dominating set* $D_{k,s}$ and then present three lemmas and a theorem as a basis for solving the MLCS problem. In the following, we use $MLCS(A_{1..i}, B_{1..j}, T_{1..k})$ to denote the length of the MLCS answer when there is no ambiguity. If it represents the sequence content, there will be explicit explanation.

Definition 1. For any two 2-tuples $\langle i_1, j_1\rangle$ and $\langle i_2, j_2\rangle$, $\langle i_1, j_1\rangle \neq \langle i_2, j_2\rangle$, we say that $\langle i_1, j_1\rangle$ dominates $\langle i_2, j_2\rangle$ if $i_1 \leq i_2$ and $j_1 \leq j_2$.

Definition 2. Let $D_{k,s}$, where $k, s \geq 0$, denote the dominating set $\{\langle i_2, j_2\rangle \mid MLCS(A_{1..i_2}, B_{1..j_2}, T_{1..k}) = s$, and for any $\langle i_1, j_1\rangle$ that dominates $\langle i_2, j_2\rangle$, $MLCS(A_{1..i_1}, B_{1..j_1}, T_{1..k}) \leq s-1\}$. In other words, any two distinct elements in $D_{k,s}$ do not dominate each other.

By Definitions 1 and 2, and the definition of the MLCS problem, there are some facts presented as follows.

Fact 1. *$D_{k,s} = \emptyset$ if $k < s$, $k < 0$ or $s < 0$.*

Fact 2. *$MLCS()$ is a monotonically increasing function. In other words, $MLCS(A_{1..i}, B_{1..j}, T_{1..k}) \leq MLCS(A_{1..x}, B_{1..y}, T_{1..z})$ if $i \leq x$, $j \leq y$ and $k \leq z$.*

Then, we have the following lemmas.

Lemma 1. *If $MLCS(A_{1..i}, B_{1..j}, T_{1..k}) = s$, there exists $\langle x, y\rangle$ which dominates $\langle i, j\rangle$ such that $MLCS(A_{1..x}, B_{1..y}, T_{1..k}) = s-1$, where $k, s \geq 1$.*

Proof. By the definition of the MLCS problem, $E(A_{1..i}, B_{1..j})$ and $T_{1..k}$ have a common subsequence of length s. Let merged sequence $E(A_{1..i}, B_{1..j}) = e_1e_2e_3\dots e_{i+j}$. There must exist s matching index pairs in order $\langle e_{x_1}, t_{y_1}\rangle$, $\langle e_{x_2}, t_{y_2}\rangle$, $\cdots$,

$\langle e_{x_s}, t_{y_s}\rangle$ to be a common subsequence with length s, where $1 \le x_1 < x_2 < \cdots \le (i+j)$ and $1 \le y_1 < y_2 < \cdots \le k$. Therefore, the common subsequence with length $s-1$ can be obtained by deleting the last matching index pair $\langle e_{x_s}, t_{y_s}\rangle$. Thus, lemma holds. □

Lemma 2. *For $\langle i_1, j_1\rangle \in D_{k,s-1}$ and $\langle i_2, j_2\rangle \in D_{k,s}$, $k, s \ge 1$, it is true that $i_1 < i_2$ or $j_1 < j_2$.*

Proof. Assume that $i_2 \le i_1$ and $j_2 \le j_1$. By Fact 2, $s = MLCS(A_{1..i_2}, B_{1..j_2}, T_{1..k}) \le MLCS\ (A_{1..i_1}, B_{1..j_1}, T_{1..k}) = s-1$. It is obviously not true that $s \le s-1$. Therefore, the assumption is not true and the lemma holds. □

Lemma 3. *For $\langle i_1, j_1\rangle \in D_{k-1,s}$ and $\langle i_2, j_2\rangle \in D_{k,s}$, $k, s \ge 1$, it is true that $i_2 \le i_1$ or $j_2 \le j_1$.*

Proof. We prove this lemma by contradiction. Assume that $i_1 < i_2$ and $j_1 < j_2$. By Fact 2, we have

$$MLCS(A_{1..i_1}, B_{1..j_1}, T_{1..k-1}) = s \le MLCS(A_{1..i_1}, B_{1..j_1}, T_{1..k}).$$

Then, we have

$$MLCS(A_{1..i_1}, B_{1..j_1}, T_{1..k}) = s + \lambda,\ \lambda \ge 0.$$

Case 1: If $\lambda = 0$, then $\langle i_1, j_1\rangle$ can be put into $D_{k,s}$. Thus, $\langle i_1, j_1\rangle$ dominates $\langle i_2, j_2\rangle$ in $D_{k,s}$.
Case 2: If $\lambda \ge 1$, then by applying Lemma 1 λ times, we can finally find $\langle x, y\rangle$ that $x \le i_1 < i_2$, $y \le j_1 < j_2$, and $MLCS(A_{1..x}, B_{1..y}, T_{1..k}) = s$. It implies $\langle x, y\rangle$ dominates $\langle i_2, j_2\rangle$ in $D_{k,s}$.
By Definition 1, both cases are not true. Therefore, the assumption is not true and the lemma holds. □

For solving the MLCS problem efficiently, we first define three functions ExtA, ExtB and Dominate. Let $\langle i_1, j_1\rangle \in D_{k-1,s-1}$. Note that $MLCS(A_{1..i_1}, B_{1..j_1}, T_{1..k-1}) = s-1$. By observing one more character t_k in T, ExtA($D_{k-1,s-1}$) is to extend each element $\langle i_1, j_1\rangle \in D_{k-1,s-1}$ to another 2-tuple $\langle i_2, j_1\rangle$, $i_1 < i_2 \le m$, such that i_2 is the smallest index for $a_{i_2} = t_k$. If there exists no such $\langle i_2, j_1\rangle$, the extension result is defined to be empty. Similarly, ExtB($D_{k-1,s-1}$) extends each element $\langle i_1, j_1\rangle \in D_{k-1,s-1}$ to $\langle i_1, j_2\rangle$, $j_1 < j_2 \le n$, such that j_2 is the smallest index for $b_{j_2} = t_k$. After this extension, we get $MLCS(A_{1..i_2}, B_{1..j_1}, T_{1..k}) \ge s$ and $MLCS(A_{1..i_1}, B_{1..j_2}, T_{1..k}) \ge s$. In other words, we extend one more character in T to find one more common character in A or B, and thus the length of the solution is extended from $s-1$ to s or more. Note that in some cases, the length of the extended result may be increased by more than 1.

For example, suppose $A = acg$, $B = ccca$, $T = actcgc$. We have $MLCS(\varepsilon, B_{1..4}, T_{1..1}) = 1$, where ε denotes an empty string, and $MLCS(A_{1..1}, \varepsilon, T_{1..1}) = 1$, thus $D_{1,1} = \{\langle 0, 4\rangle, \langle 1, 0\rangle\}$. Next, $\langle 0, 4\rangle$ is extended to $\langle 2, 4\rangle$ with ExtA by finding the index of the first matching character of c (t_2) in A. Similarly, ExtA($\langle 1, 0\rangle$) = $\langle 2, 0\rangle$. So, ExtA($D_{1,1}$) = $\{\langle 2, 0\rangle, \langle 2, 4\rangle\}$. We can also get ExtB($D_{1,1}$) = $\{\langle 1, 1\rangle\}$. Thus, we get an extended set W = ExtA($D_{1,1}$) ∪ ExtB($D_{1,1}$) = $\{\langle 1, 1\rangle, \langle 2, 0\rangle, \langle 2, 4\rangle\}$. As another example, consider $A = g$, $B = accct$, $T = cccat$. $\langle 0, 1\rangle \in D_{4,1}$. After the extension, we get ExtB($\langle 0, 1\rangle$) = $\langle 0, 5\rangle$. As one can see, $MLCS(\varepsilon, accct, cccat) = 4$, which is not 2. Thus, after the extension, the MLCS length is increased by 3, not 1.

Dominate is to merge two sets of 2-tuples and to remove all 2-tuples that are dominated by others. In other words, the input set for Dominate may not be a dominating set, while its output is a dominating set. For example, suppose $W = \{\langle 1, 1\rangle, \langle 2, 0\rangle, \langle 2, 4\rangle\}$. We get Dominate($W$) = $\{\langle 1, 1\rangle, \langle 2, 0\rangle\}$, since $\langle 2, 4\rangle$ is dominated by $\langle 2, 0\rangle$.

Theorem 1. $D_{k,s} = \text{Dominate}(D_{k-1,s} \cup \text{ExtA}(D_{k-1,s-1}) \cup \text{ExtB}(D_{k-1,s-1}))$, $1 \le k,\ s \le r$.

Proof. For each $\langle x', y'\rangle \in D_{k-1,s} \cup$ ExtA($D_{k-1,s-1}$) ∪ ExtB($D_{k-1,s-1}$), if $MLCS(A_{1..x'}, B_{1..y'}, T_{1..k}) = s$ and it is dominated by others, it can be obviously done. So, our proof focuses on that $\langle x', y'\rangle$ will be dominated by another 2-tuple in $D_{k,s}$ if $MLCS(A_{1..x'}, B_{1..y'}, T_{1..k}) \ge s+1$.

- Consider the case for ExtA($D_{k-1,s-1}$) ∪ ExtB($D_{k-1,s-1}$).
  Let $\langle i_1, j_1\rangle \in D_{k-1,s-1}$. Suppose ExtA($\{\langle i_1, j_1\rangle\}$) ∪ ExtB($\{\langle i_1, j_1\rangle\}$) = $\{\langle x, j_1\rangle, \langle i_1, y\rangle\}$. By Fact 2, we have

  Case 1: $MLCS(A_{1..x}, B_{1..j_1}, T_{1..k}) = s + \lambda,\ \lambda \ge 0,$

  Case 2: $MLCS(A_{1..i_1}, B_{1..y}, T_{1..k}) = s + \mu,\ \mu \ge 0.$

  For case 1, we consider the following cases:
  Case (a): If $\lambda = 0$, then $MLCS(A_{1..x}, B_{1..j_1}, T_{1..k}) = s$.
  Case (b): If $\lambda > 0$, then by applying Lemma 1 λ times, we can find $\langle x_1, y_1\rangle$, $x_1 \le x$ and $y_1 \le j_1$, such that

  $$MLCS(A_{1..x_1}, B_{1..y_1}, T_{1..k}) = s.$$

  It implies that $\langle x_1, y_1\rangle$ dominates $\langle x, j_1\rangle$. $\langle x, j_1\rangle$ will be removed after the Dominate function is applied. Thus, $\langle x, j_1\rangle$ cannot be in $D_{k,s}$. The proof for case 2 is similar.

Table 2
The construction of $D_{k,s}$ in our MLCS algorithm with $A = acg$, $B = ccca$ and $T = actcgc$.

Round i	s (length)					
	0	1	2	3	4	5
1	$D_{0,0}$	$D_{1,1}$	$D_{2,2}$			
	$\langle 0,0\rangle$	$\langle 0,4\rangle$	$\langle 1,1\rangle$			
		$\langle 1,0\rangle$	$\langle 2,0\rangle$			
			~~$\langle 2,4\rangle$~~			
2	$D_{1,0}$	$D_{2,1}$	$D_{3,2}$	$D_{4,3}$	$D_{5,4}$	$D_{6,5}$
	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,1\rangle$	$\langle 1,2\rangle$	$\langle 3,1\rangle$	$\langle 3,2\rangle$
		~~$\langle 0,4\rangle$~~	$\langle 2,0\rangle$	$\langle 2,1\rangle$	~~$\langle 3,2\rangle$~~	
		$\langle 1,0\rangle$		~~$\langle 2,1\rangle$~~		
		~~$\langle 2,0\rangle$~~				

- Consider the case for $D_{k-1,s}$.
  Suppose that there exists $\langle i_2, j_2\rangle \in D_{k-1,s}$ such that
  $$MLCS(A_{1..i_2}, B_{1..j_2}, T_{1..k}) = s + \lambda,\ \lambda \geq 1.$$
  Then by applying Lemma 1 λ times, we find $\langle x_2, y_2\rangle$, $x_2 \leq i_2$ and $y_2 \leq j_2$, such that
  $$MLCS(A_{1..x_2}, B_{1..y_2}, T_{1..k}) = s.$$
  It implies that $\langle x_2, y_2\rangle$ dominates $\langle i_2, j_2\rangle$. $\langle i_2, j_2\rangle$ will be removed after the Dominate function is applied. Thus, $\langle i_2, j_2\rangle$ cannot be in $D_{k,s}$.

According to the above, for $\langle x', y'\rangle \in D_{k-1,s} \cup \text{ExtA}(D_{k-1,s-1}) \cup \text{ExtB}(D_{k-1,s-1})$, where $MLCS(A_{1..x'}, B_{1..y'}, T_{1..k}) > s$, $\langle x', y'\rangle$ will be dominated by another 2-tuple in $D_{k,s}$. And after the Dominate function is applied, all dominated elements will be removed. Therefore, the theorem holds. □

With Theorem 1, the MLCS problem can be solved by calculating $D_{k,s}$. For easy implementation of Dominate, we should keep the data set of the 2-tuples by increasing order in the first dimension and decreasing order in the second dimension. With this arrangement, we can merge two data sets and eliminate those dominated 2-tuples by the linear merging scheme. The pseudo code of our MLCS algorithm is shown in Algorithm 1. Theorem 1 is implemented by function Generate (described in Function 1).

Algorithm 1 Computing the MLCS length and $D_{k,s}$.

```
Input: Sequences $A = a_1a_2a_3 \dots a_m$, $B = b_1b_2b_3 \dots b_n$ and $T = t_1t_2t_3 \dots t_r$
Output: Length of $MLCS(A, B, T)$
 1: Construct the arrays of $next_A$ and $next_B$ // next character position
 2: $L \leftarrow 0$
 3: for $i = 1 \to r$ do
 4:     Set $D_{i-1,0} = \{\langle 0,0\rangle\}$
 5:     for $s = 1 \to r - i + 1$ do
 6:         $k \leftarrow i + s - 1$
 7:         $D_{k,s} \leftarrow$ Generate($D_{k-1,s}$, $D_{k-1,s-1}$)
 8:         if $D_{k,s} = \emptyset$ then
 9:             break
10:     if $s > L$ then
11:         $L \leftarrow s$
12:     if $i > r - L$ then
13:         break
14: return $L$
```

The main loop (containing lines 3 through 13) constructs $D_{k,s}$ with the row-major scheme as the example shown in Table 2. In the ith round (lines 5 through 9), it scans T sequentially starting from t_i to construct sets $D_{i,1}$, $D_{i+1,2}$, $D_{i+2,3}$, $\cdots$, $D_{i+s-1,s}$ until the set cannot be constructed, where $s = MLCS(A_{1..m}, B_{1..n}, T_{1..(i+s-1)})$, the maximum length that can be obtained in this round. Generate in Line 7 implements Theorem 1. Line 8 decides whether the termination condition is satisfied or not in this round. In lines 10 and 11, L stores the maximum length s in this round. We quit our MLCS algorithm when $i > r - L$ in lines 12 and 13, because the maximum length of the ith round is $r - i + 1$. Thus, after round $i + 1$, the maximum length will be no more than the current L. Eventually, the final L represents the length of $MLCS(A_{1..m}, B_{1..n}, T_{1..r})$.

Now, we explain the details of the example shown in Table 2, where $A = acg$, $B = ccca$ and $T = actcgc$. In the first round, we first set $D_{0,0} = \{\langle 0,0\rangle\}$, and then $D_{1,1}$ is constructed from $\text{Dominate}(D_{0,1} \cup \text{ExtA}(D_{0,0}) \cup \text{ExtB}(D_{0,0}))$ according to

Function 1 Generating $D_{k,s}$ by extending $D_{k-1,s-1}$ and uniting $D_{k-1,s}$.

```
Input: $D_{k-1,s}$ and $D_{k-1,s-1}$
Output: $D_{k,s}$
1: function GENERATE($D_{k-1,s}$, $D_{k-1,s-1}$)
2:     $W_a, W_b \leftarrow \emptyset$
3:     for $\langle i_c, j_c\rangle \in D_{k-1,s-1}$ ($c = 1$ to $|D_{k-1,s-1}|$) do        // $i_c$ ($j_c$) is strictly increasing (decreasing)
4:         if ($i'_c \leftarrow next_A(t_k, i_c)) \le m$ then Append $\langle i'_c, j_c\rangle$ into $W_a$        // ExtA
5:         if ($j'_c \leftarrow next_B(t_k, j_c)) \le n$ then Append $\langle i_c, j'_c\rangle$ into $W_b$        // ExtB
6:     $D \leftarrow$ DOMINATE($D_{k-1,s}$, $W_a$)        // Dominate($D_{k-1,s} \cup$ ExtA)
7:     $D \leftarrow$ DOMINATE($D$, $W_b$)        // Dominate(Dominate($D_{k-1,s} \cup$ ExtA) $\cup$ ExtB)
8:     Return $D$
9: end function
```

Function 2 Merging two 2-tuple lists and eliminating dominated 2-tuples.

```
Input: 2-tuple lists $L_1$ and $L_2$, which are nondecreasing and nonincreasing in the first and second dimensions, respectively
Output: Merged and non-dominated 2-tuple list $R$
1: function DOMINATE($L_1$, $L_2$)
2:     $R \leftarrow \emptyset$
3:     while $L_1 \ne \emptyset$ or $L_2 \ne \emptyset$ do
4:         $\langle i_s, j_s\rangle \leftarrow$ the non-null smaller one of the first 2-tuples of $L_1$ and $L_2$
5:         $\langle i_r, j_r\rangle \leftarrow$ the last 2-tuple of $R$
6:         if ($\langle i_s, j_s\rangle$ dominates $\langle i_r, j_r\rangle$) then Replace $\langle i_r, j_r\rangle$ with $\langle i_s, j_s\rangle$
7:         else if ( $\langle i_r, j_r\rangle$ dominates $\langle i_s, j_s\rangle$) then Discard $\langle i_s, j_s\rangle$
8:         else Append $\langle i_s, j_s\rangle$ into $R$
9:     Return $R$
10: end function
```

Theorem 1. $D_{0,1}$ is an empty set by Fact 1. Thus, $D_{1,1} = \{\langle 0,4\rangle, \langle 1,0\rangle\}$. Then $D_{2,2}$ can be constructed from DOMINATE($D_{1,2} \cup$ EXTA($D_{1,1}$) $\cup$ EXTB($D_{1,1}$)). Note that $\langle 2,4\rangle$ is removed, since $\langle 2,4\rangle$ is dominated by $\langle 2,0\rangle$. Thus, $D_{2,2} = \{\langle 1,1\rangle, \langle 2,0\rangle\}$. $D_{3,3}$ is an empty set, because we cannot extend any 2-tuple from set $D_{2,2}$. Accordingly, the first round stops. In the second round, $D_{2,1}$ can be constructed from $D_{1,0}$ and $D_{1,1}$. Repeating the above steps, we get $D_{2,1}$, $D_{3,2}$, $D_{4,3}$, $D_{5,4}$ and $D_{6,5}$ in this round. Our MLCS algorithm terminates since the optimal length is got in the second round, which is $r - i + 1$. Therefore, the length of $MLCS(A_{1..m}, B_{1..n}, T_{1..r}) = 5$.

Now, we go into the details of the two functions, GENERATE and DOMINATE. The main action of GENERATE is to extend one more character in T and it considers the matching characters only, such as $a_i = t_k$ or $b_j = t_k$. To find out the matching pairs efficiently, we build a character table with Peng's algorithm [29]. In the character table, the function $next_A(\alpha, i) = i'$ denotes the next position of character α in A after position i. $i' = \infty$ if no such i' can be found. It takes $O(m|\Sigma|)$ time to construct the character table of A in the preprocessing stage, where $|\Sigma|$ denotes the number of distinct characters in A. It requires only constant time to get the answer of $next_A(\alpha, i)$. Similarly, the character table of B can also be constructed.

The GENERATE function is shown in Function 1. We use $next_A$ and $next_B$ to find the index of the first matching character t_k after a_{i_c} and b_{j_c}, respectively. If we extend successfully $\langle i_c, j_c\rangle$ to another 2-tuple $\langle i'_c, j'_c\rangle$, then it is true that $i'_c \le m$ and $j'_c \le n$. Obviously, Theorem 1 can be rewritten as $D_{k,s} =$ DOMINATE(DOMINATE($D_{k-1,s} \cup$ EXTA($D_{k-1,s-1}$)) $\cup$ EXTB($D_{k-1,s-1}$)), which is implemented in lines 6 and 7.

We check the domination in DOMINATE (Function 2). Since the two 2-tuple lists input to the function are arranged as that they are nondecreasing and nonincreasing in the first and second dimensions, respectively, the job for removing the dominated 2-tuples can be easily done with the linear merging scheme.

Theorem 2. *Algorithm 1 solves the MLCS problem in $O(n|\Sigma| + (r - L + 1)Lm)$ time and $O(n|\Sigma| + Lm)$ space.*

Proof. By Theorem 1, Algorithm 1 is correct. The preprocessing stage (line 1) takes $O((m + n)|\Sigma|) = O(n|\Sigma|)$ time to construct the arrays of $next_A$ and $next_B$. The outer loop in lines 3 through 13 is executed exactly $(r - L + 1)$ times. (The loop breaks when $i > r - L$.) The inner loop in lines 5 through 9 is executed at most L times, and each GENERATE and DOMINATE can be done in $O(|D|)$ time, where $|D| \le m$. Thus, the time complexity is $O(n|\Sigma| + (r - L + 1)Lm)$.

For the space complexity, there are $(r - L + 1)$ iterations in the outer loop, the length of each iteration (lines 5 through 9) is at most L, and the number of tuples in each $D_{k,s}$ can be no more than m. For practical implementation of Algorithm 1, the space of $D_{k-1,s}$ can be reused as the storage of $D_{k,s}$. In other words, all iterations in the outer loop can share the same storage space. Thus, the space complexity is $O(n|\Sigma| + Lm)$. □

4. Our block-merged LCS algorithm

With slight modification of our MLCS algorithm, we can also solve the BMLCS problem by properly dealing with EOB (end of a block). In our MLCS algorithm, the function GENERATE is to extend $D_{k-1,s-1}$ to $D_{k,s}$, which finds the next match of t_k from A or B. As for the BMLCS problem, we need to modify it to be suitable for blocked merged sequences. Here,

Table 3
An example of our BMLCS algorithm for $A = a\#cg\#$, $B = ccc\#a\#$ and $T = actcgc$.

Round i	s (length)				
	0	1	2	3	4
1	$D_{0,0}$	$D_{1,1}$	$D_{2,2}$		
	$\langle 0,0\rangle$	$\langle 0,4\rangle$ $\langle 1,0\rangle$	$\langle 1,1\rangle$ $\langle 2,0\rangle$ ~~$\langle 2,4\rangle$~~		
2	$D_{1,0}$	$D_{2,1}$	$D_{3,2}$	$D_{4,3}$	$D_{5,4}$
	$\langle 0,0\rangle$	$\langle 0,1\rangle$ ~~$\langle 0,4\rangle$~~ $\langle 1,0\rangle$ ~~$\langle 2,0\rangle$~~	$\langle 1,1\rangle$ $\langle 2,0\rangle$	$\langle 1,2\rangle$ $\langle 3,1\rangle$	$\langle 3,3\rangle$

the modified function is named BGenerate. Suppose $\langle i,j\rangle \in D_{k-1,s-1}$. By considering each character position of the merging block sequences A and B, there are three possibilities of EOB situation:

(1) $a_i \in EOB$ and $b_j \in EOB$,
(2) $a_i \in EOB$ and $b_j \notin EOB$,
(3) $a_i \notin EOB$ and $b_j \in EOB$.

For case 1, we find the next matching character freely from the rest of merging block sequences of A and B. This is the same as Generate. For cases 2 and 3, one of these two characters is not the end character of a block (it is inside a block). If $a_i \in EOB$ and $b_j \notin EOB$, we cannot extend 2-tuples from A since both b_j and the next matching character are within the same block. Case 3 is similar to case 2. According to the above, we present our new generation in the following.

First, we introduce a new function *BlockPos*. Given a merging block sequence S, $BlockPos\ (S,i) = i'$ denotes the position of character $s_{i'}$ after s_i such that $s_{i'} \in EOB$ of S. If $s_i \in EOB$ of S, then $i' = i$. The calculation of $BlockPos(A,i)$ and $BlockPos(B,j)$, $0 \le i \le m$ and $0 \le j \le n$, can be done in $O(m)$ and $O(n)$ time in the preprocessing stage, respectively.

The spirit of our new generation function BGenerate is to ensure that the position of the non-extended sequence must be in EOB. If $a_{i_c} \notin EOB$ of A or $b_{j_c} \notin EOB$ of B, we apply the function *BlockPos* to find the proper position i_{cB} or j_{cB}.

Function 3 Generating $D_{k,s}$ by extending $D_{k-1,s-1}$ and uniting $D_{k-1,s}$ for BMLCS.

```
Input: $D_{k-1,s}$ and $D_{k-1,s-1}$
Output: $D_{k,s}$
 1: function BGenerate( $D_{k-1,s}$, $D_{k-1,s-1}$ )
 2:     $W_a, W_b \leftarrow \emptyset$
 3:     for each $\langle i_c, j_c\rangle \in D_{k-1,s-1}$ ($c = 1$ to $|D_{k-1,s-1}|$) do
 4:         $i'_c \leftarrow next_A(t_k, i_c)$
 5:         $j'_c \leftarrow next_B(t_k, j_c)$
 6:         if ($i'_c \le m$) then
 7:             if $j_c \in EOB$ of $B$ then Append $\langle i'_c, j_c\rangle$ into $W_a$
 8:             else if ($j_{cB} \leftarrow BlockPos(B, j_c)) < j'_c$ then
 9:                 Append $\langle i'_c, j_{cB}\rangle$ into $W_a$                    // BExtA
10:         if ($j'_c \le n$) then
11:             if $i_c \in EOB$ of $A$ then Append $\langle i_c, j'_c\rangle$ into $W_b$
12:             else if ($i_{cB} \leftarrow BlockPos(A, i_c)) < i'_c$ then
13:                 Append $\langle i_{cB}, j'_c\rangle$ into $W_b$                    // BExtB
14:     $D \leftarrow$ Dominate($D_{k-1,s} W_a$)          // Dominate($D_{k-1,s} \cup$ BExtA)
15:     $D \leftarrow$ Dominate($D, W_b$)          // Dominate(Dominate($D_{k-1,s} \cup$ BExtA) $\cup$ BExtB)
16:     return $D$
17: end function
```

An example of our BMLCS algorithm is shown in Table 3, where $A = a\#cg\#$, $B = ccc\#a\#$ and $T = actcgc$. In the first round, we first set $D_{0,0} = \{\langle 0,0\rangle\}$, and then $D_{1,1}$ is constructed from Dominate($D_{0,1} \cup$ BExtA($D_{0,0}$) $\cup$ BExtB($D_{0,0}$)). Here, BExtA and BExtB have the similar concept of ExtA and ExtB but with blocks. $D_{0,1}$ is an empty set by Fact 1. Then we find the first matching character a (t_1) freely from A and B since $\langle 0,0\rangle$ is in EOB of both A and B. Thus, we get $D_{1,1} = \{\langle 0,4\rangle, \langle 1,0\rangle\}$. Then $D_{2,2}$ can be constructed by Dominate($D_{1,2} \cup$ BExtA($D_{1,1}$) $\cup$ BExtB($D_{1,1}$)). Note that $\langle 2,4\rangle$ has to be removed, since $\langle 2,0\rangle$ dominates $\langle 2,4\rangle$. Thus, we get $D_{2,2} = \{\langle 1,1\rangle, \langle 2,0\rangle\}$. The first round ends, because $D_{3,3}$ is empty.

In the second round, we obtain $D_{2,1} = \{\langle 0,1\rangle, \langle 1,0\rangle\}$. Repeatedly, we get $D_{2,1}$, $D_{3,2}$, $D_{4,3}$ and $D_{5,4}$ in this round. For details, we use $D_{4,3}$ as an example of demonstration. $D_{4,3}$ can be constructed by Dominate($D_{3,3} \cup$ BExtA($D_{3,2}$) $\cup$ BExtB($D_{3,2}$)). When we extend $\langle 1,1\rangle \in D_{3,2}$ to $D_{4,3}$, A is not considered since $b_1 \notin EOB$, and the index of the next matching character of t_4 (c) after b_1 is 2. So, $\langle 1,2\rangle$ is got. For the extension of $\langle 2,0\rangle$, we move the index of A to 3 since $a_2 \notin EOB$, and find

$b_1 = t_4$ by freely searching A and B. So, $\langle 3, 1\rangle$ is got. The possible maximum length in the second round is $r - i + 1$, which is 5. We get the length 4 in the second round and the optimal length in the third round cannot be more than 4. Therefore, we stop our BMLCS algorithm at the second round.

In summary, Algorithm 1 can calculate the BMLCS length if we replace our extension function in Generate to that in BGenerate. Let δ denote the maximum number of blocks of A and B, then we have Theorem 3.

Theorem 3. *Algorithm 1 with* BGenerate *solves the BMLCS problem in $O(n|\Sigma| + (r - L + 1)L\delta)$ time and $O(n|\Sigma| + L\delta)$ space.*

Proof. In the preprocessing stage, we take $O((m + n)|\Sigma|) = O(n|\Sigma|)$ time to construct arrays of $next_A$ and $next_B$, and $O(m + n) = O(n)$ time to construct $BlockPos$. Since BGenerate($D_{k-1,s-1}$, $D_{k-1,s}$) and Dominate(D, W) can be done in $O(|D|)$ time, where $|D| \le \delta$. Thus, the time complexity is $O(n|\Sigma| + (r - L + 1)L\delta)$. Similar to Theorem 2, the space of $D_{k-1,s}$ can be reused as the storage of $D_{k,s}$. The space complexity is $O(n|\Sigma| + L\delta)$. □

5. Experimental results

To demonstrate the efficiency of our MLCS and BMLCS algorithms, we first compare the execution time of our algorithms and some previously published algorithms on pseudorandom sequences. Then we compare the efficiency of two superior algorithms, our MLCS algorithm and the bit-parallel MLCS algorithm [9], on some real DNA sequences. These algorithms are implemented by ourselves using Visual Studio C++ 2013 software, and tested on a computer with 64-bit Windows 7 OS, CPU clock rate of 3.2 GHz (Intel i5-4570) and 32 GB of RAM, or a computer with less power. The source codes and datasets for experiments can be found on our web site http://par.cse.nsysu.edu.tw/~mlcs/ [20].

We produce some pseudorandom datasets for experiments to clarify the relation between the sequence similarity and execution time. These datasets are generated or mutated by the pseudorandom function of C++, which typically exhibits statistical randomness. Thus, we use the word "random" or "randomly" to describe the process for short.

The similarity of the input sequences for the MLCS (BMLCS) problem is defined as $\lambda = \frac{solution\ length}{\min\{|A|+|B|,|T|\}}$. For example, when the similarity λ is 96%, the MLCS length is 1920 if $|A| = |B| = 1000$ and $|T| = 2000$. The pseudorandom datasets are generated with various lengths $(1000,\ 2000,\ 5000)$, similarities $(10\% \sim 100\%)$, alphabet sizes $(4,\ 64,\ 1000)$, and ratios $\gamma = \frac{|A|}{|T|} = \frac{m}{r}$ $(0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5)$, where $|A| = m$, $|B| = n$, $|T| = r$, and $m + n = r$.

Suppose that a dataset with similarity p is desired to be generated. Our method for randomly generating experimental data is presented as follows.

Step 1: Randomly generate T. Then, randomly put each character of T into either A or B in order.
Step 2: Compute the MLCS (BMLCS) of A, B and T, and calculate their similarity λ.
Step 3: If $\lambda \in [p - \epsilon, p + \epsilon]$, output A, B and T, and stop. Otherwise, randomly mutate some characters of T, and go to Step 2. Here, we set $\epsilon = 1\%$.

The possible lowest similarity for randomly generated sequences depends on the alphabet set size. The larger the alphabet set size is, the lower the degree of sequence similarity is. The algorithms for performance comparison are denoted as follows.

- **DP**: Huang's DP algorithms of MLCS and BMLCS [15].
- **Peng**: Peng's sparse DP algorithms of MLCS and BMLCS [29].
- **Rahman**: Algorithms of MLCS and BMLCS proposed by Rahman and Rahman [31].
- **S-table**: Huang's BMLCS algorithm by using S-table [15].
- **Bit**: The bit-parallel MLCS algorithm [9] and the bit-parallel BMLCS algorithm [8] proposed by Deorowicz and Danek.
- **Ours**: Our MLCS and BMLCS algorithms presented in this paper.

5.1. Various length ratios of sequences A and T

In Fig. 1, we illustrate the influences of various length ratios $\gamma = \frac{|A|}{|T|}$ with fixed $|T|$ in different similarities. We use the format $(|T|, \gamma, |\Sigma|, Algorithm)$ to represent the experiments of each chart. When the BMLCS algorithms are compared, the maximum number δ of blocks in A and B is added as the fifth parameter. For example, $(2000, *, 64, ?)$ is used to express charts in left part of Fig. 1, where $|T| = r = 2000$, alphabet size $|\Sigma| = 64$, "$*$" is a wildcard for representing all possible contents of the parameter γ, and "?" means different algorithms in different charts.

The experiment with a combination of parameters was repeated 100 times to get the average execution time for all algorithms when $|T| = 1000$ or $|T| = 2000$. The experiment was repeated 100 times for our and bit-parallel algorithms when $|T| = 5000$, but there are only 5 times for the other three algorithms when $|T| = 5000$, because the other three algorithms require much more execution time.

Though we have performed all experiments with many various combinations of parameters, Fig. 1 only illustrates $(2000, *, 64, ?)$ and $(5000, *, 1000, ?)$, since the other experiments have similar results (for example, the cases of $|T| = 1000$ are not shown). Observing Fig. 1, we obtain the following facts.

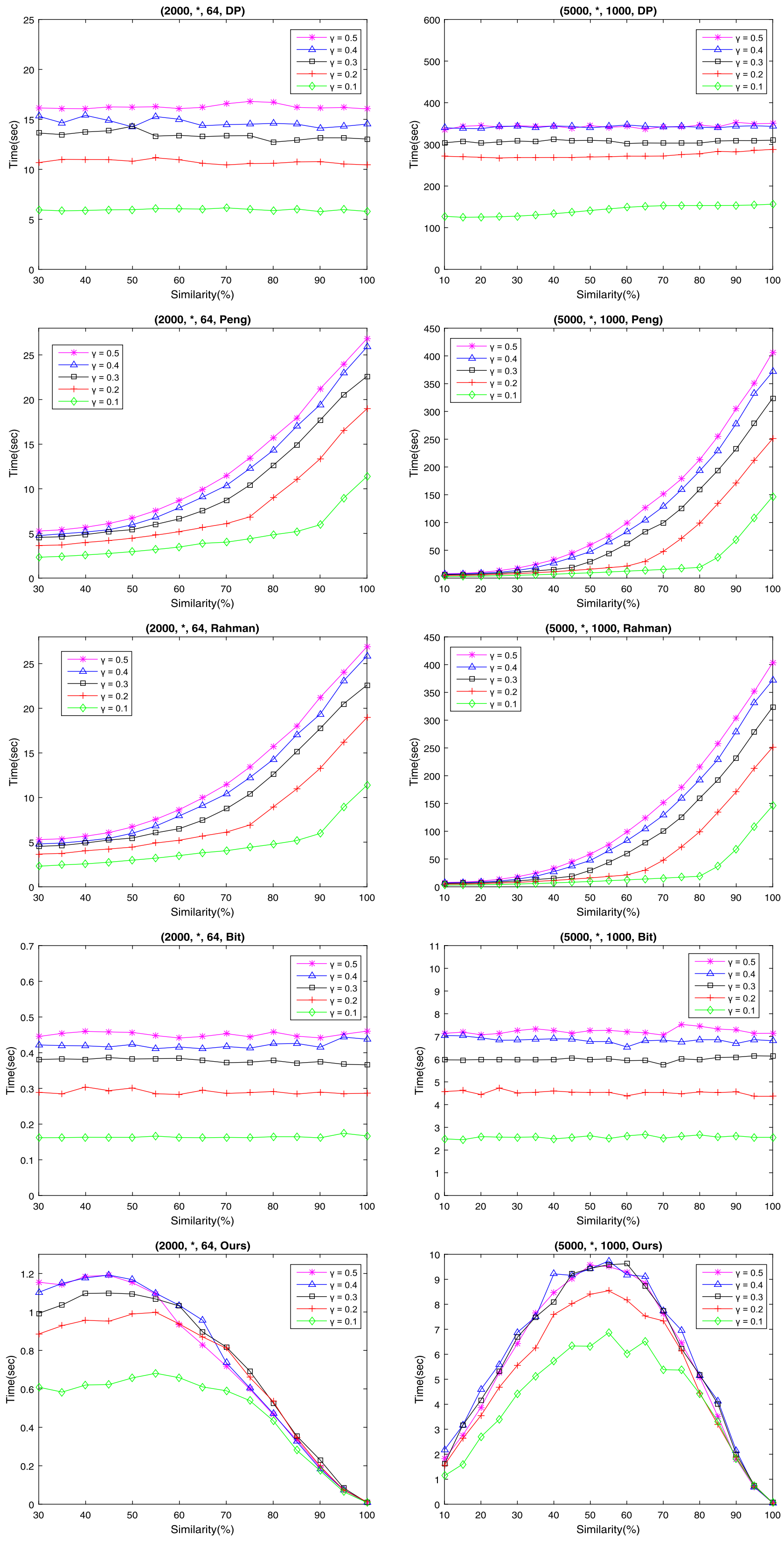

Fig. 1. The comparison of execution time with various γ in $(2000, *, 64, ?)$ and $(5000, *, 1000, ?)$.

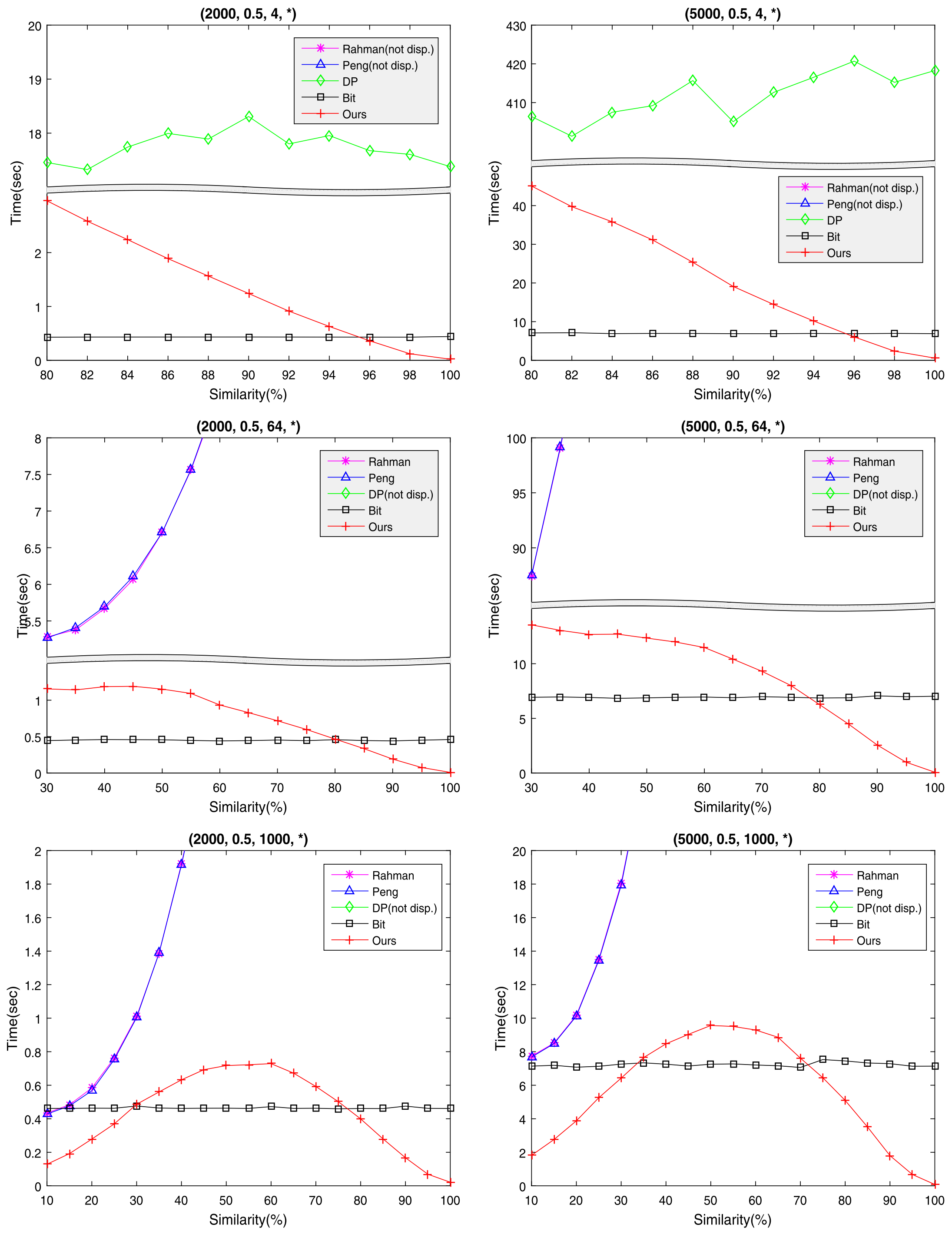

Fig. 2. The average execution time of various MLCS algorithms in $(2000, 0.5, ?, *)$ and $(5000, 0.5, ?, *)$.

First, it is obvious that all curves of DP and Bit are almost horizontal, which means that they take no advantage of sequence similarity. As expected, our algorithm works better on sequences with high similarities. However, the other two are efficient when the matchings are sparse, that almost means low similarities, although sparse matchings may not be completely identical to low similarities.

Second, all algorithms have the worst performance when γ is near 0.5. It is easy to understand from the time complexities of all algorithms. Since their time complexities are almost proportional to mn or mr, where $m \le n$ and $m + n = r$, mn or mr has the maximal value theoretically when $m = n = \frac{r}{2}$, i.e. $\gamma = 0.5$. For easy explanation, we set $\gamma = 0.5$ in all of the following experiments, except the real DNA cases.

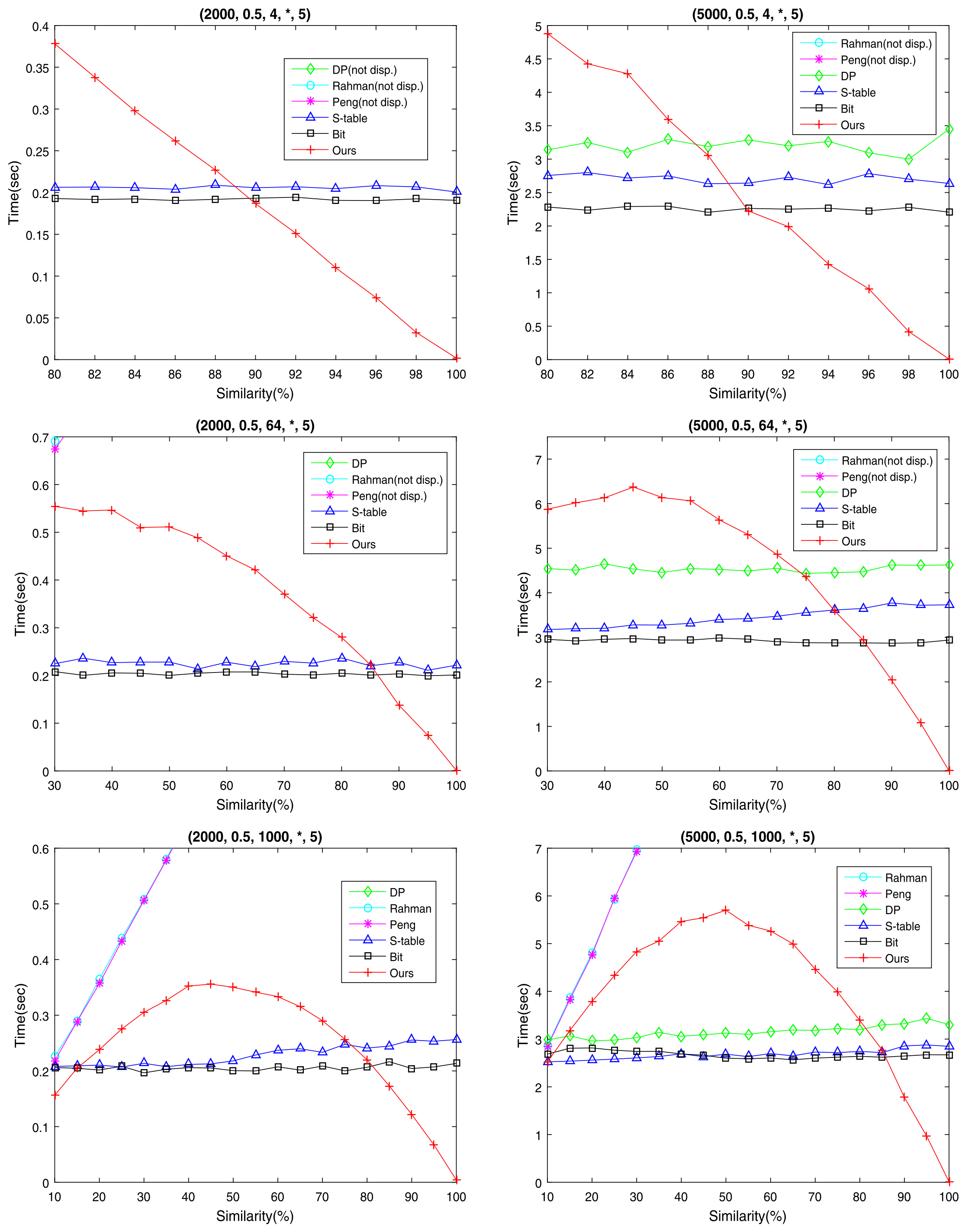

Fig. 3. The average execution time of various BMLCS algorithms in $(2000, 0.5, ?, *, 5)$ and $(5000, 0.5, ?, *, 5)$.

Third, $(r - L + 1)Lm$ is the main part of our time complexity $O(n|\Sigma| + (r - L + 1)Lm)$, and $(r - L + 1)Lm$ has the maximal value theoretically when $L = \frac{r}{2}$, whose similarity is 50%. As we can see in chart (5000, *, 1000, Ours), the curves are almost the highest when the similarities are near 50%.

5.2. Comparison of various algorithms

Fig. 2 shows the average execution times of the MLCS algorithms, where the x-axis represents the similarity of input sequences. It is hard to generate sequences with low similarity when $|\Sigma|$ is small. For example, we did not test the sequences

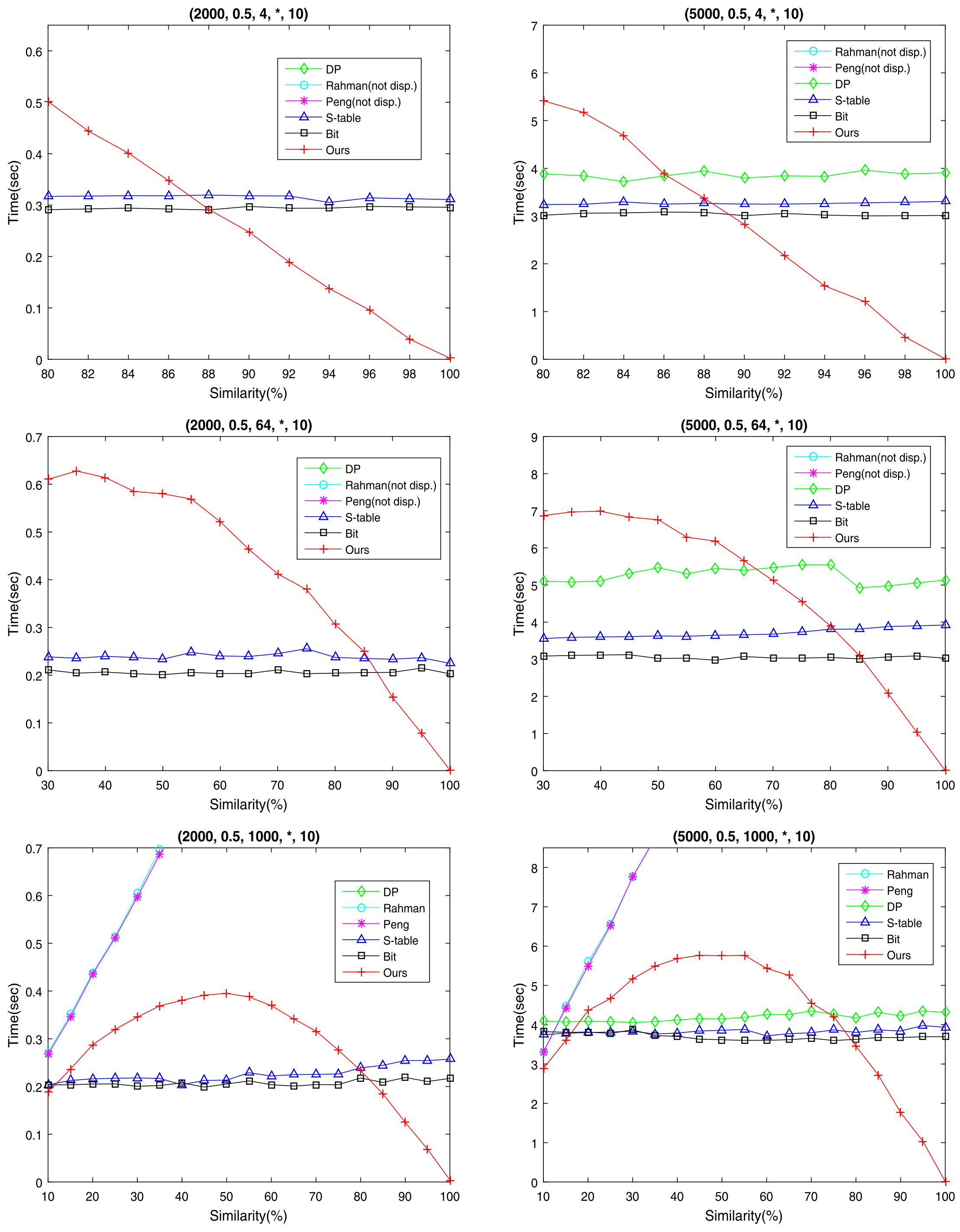

Fig. 4. The average execution time of various BMLCS algorithms in $(2000, 0.5, ?, *, 10)$ and $(5000, 0.5, ?, *, 10)$.

of similarities lower than 80% with $|\Sigma| = 4$. Since the performance of some algorithms is far worse than the others, there is a gap on the y-axis in some charts. Moreover, some results are even not displayed in the chart, because their curves fall outside the chart.

Fig. 2 indicates that the performance of our MLCS algorithm is always better than the algorithms of DP, Peng, and Rahman. It is clear that only our and bit-parallel MLCS algorithms may compete for the winner. Our MLCS is very efficient when the input sequences are extremely similar. For example, when the sequence similarity exceeds 96% with $|\Sigma| = 4$, our MLCS algorithm is the most efficient among all algorithms. Our algorithm is the winner when the sequence similarity exceeds 80% with $|\Sigma| = 64$ or $|\Sigma| = 1000$. In other words, it reveals that the larger $|\Sigma|$ is, the lower similarities we need

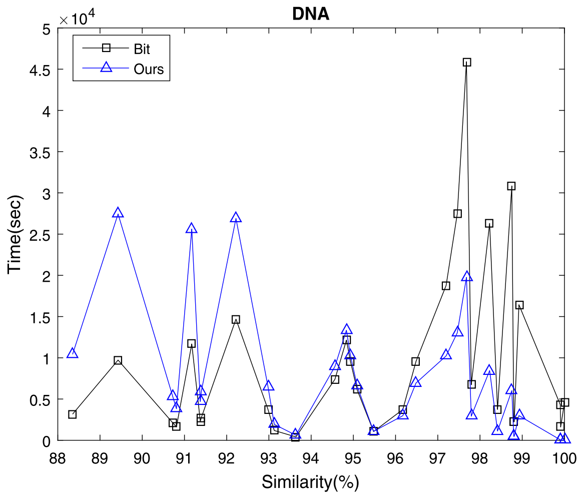

Fig. 5. The comparison of execution time between our and bit-parallel MLCS algorithms in real DNA sequences.

for our algorithm to be the winner. The other thing to be noticed is that in the case of $|\Sigma| = 1000$, our MLCS algorithm with similarity lower than 30% is also the winner, because the main part $(r - L + 1)Lm$ of our time complexity $O(n|\Sigma| + (r - L + 1)Lm)$ is small when L is low.

Figs. 3 and 4 illustrate the average execution time of BMLCS algorithms for various alphabet sizes, with two different numbers of blocks ($\alpha = \beta = 5$ and $\alpha = \beta = 10$) in sequences A and B, respectively. Figs. 3 and 4 show that the performances of DP, S-table and bit-parallel BMLCS algorithms are almost independent of sequence similarity. There is a great improvement for the DP algorithm in computing time when the BMLCS problem is applied. For example, the DP algorithm needs over 400 s in chart $(5000, 0.5, 4, *)$ of Figs. 2, but it takes only 3 or 4 s in the same case with 5 or 10 blocks of A and B in chart $(5000, 0.5, 4, *, 5)$ or $(5000, 0.5, 4, *, 10)$. Thus, the DP algorithm works better with fewer blocks. The behavior of our BMLCS algorithm is almost the same as that of our MLCS algorithm. The efficiency of our algorithm depends on the input sequence similarity, and has a better performance in a small number δ of blocks. It can still be seen that our BMLCS algorithm is the winner when the input sequences are very similar.

5.3. *Real DNA sequences*

The experimental materials are 31 sets of real DNA sequences, coming from two yeast species *Saccharomyces cerevisiae* and *Kluyveromyces waltii* [18]. In the "Supplementary info" of provided URL in [18], there are raw nucleotide sequences of all predicted Open Reading Frames (ORFs), which form the source of our sequence T. We then use the information in "Matches by chromosome" to find gene correspondences and make them into nucleotide sequences as our sequences A and B by [10]. Some statistical data of these real DNA sequences are given as follows. $15237 \le |T| \le 56451$, $8405 \le |A| \le 50835$, $14523 \le |B| \le 61695$, $0.570 \le \gamma = \frac{|A|}{|T|} \le 1.047$, $1.237 \le \frac{|A|+|B|}{|T|} \le 2.171$, and $88.35\% \le similarity \le 100\%$. Since our and bit-parallel MLCS algorithms are competitive, Fig. 5 only shows the execution time of these two algorithms. It can be seen that our algorithm is faster than the bit-parallel MLCS algorithm when the similarity is greater than 96%. The experimental results of the real DNA sequences is consistent with the results in Fig. 2 for pseudorandom sequences with $|\Sigma| = 4$. These results reveal no difference between pseudorandom and real sequences. Thus, it is feasible to evaluate the behaviors of MLCS algorithms in various parameters by using pseudorandom sequences.

6. Conclusion

In this paper, we first propose an MLCS algorithm with $O(n|\Sigma| + (r - L + 1)Lm)$ time and $O(n|\Sigma| + Lm)$ space, where r and L denote the lengths of T and MLCS, respectively, and m and n denote the minimum and maximum lengths of A and B, respectively. With slight modification, our MLCS algorithm can also solve the BMLCS problem in $O(n|\Sigma| + (r - L + 1)L\delta)$ time and $O(n|\Sigma| + L\delta)$ space, where δ denotes the maximum number of blocks in A and B. By the time complexity, our algorithms are efficient when L is large, i.e. the input sequences are similar.

We compare the execution efficiency of our algorithms with some previously published MLCS and BMLCS algorithms on pseudorandom and real DNA sequences. The experimental results show that our algorithms are better than algorithms of Huang et al. [16], Peng et al. [29], and Rahman and Rahman [31], which are DP-based algorithms. Our algorithms are faster than previous MLCS and BMLCS algorithms when the input sequences are very similar. It is reasonable to assume that input sequences have high similarities. In 2000, Batzoglou et al. showed that coding regions of mRNA sequences between humans and mice are approximately 85% identical [5]. In 2001, Venter et al. in Human Genome Project (HGP) concluded that we are 99.9% genetically similar to other people [37]. In 2005, Varki and Altheide proposed that the difference between genomes of humans and chimpanzees is 4%, i.e. 96% of genomes are identical [36]. Then, in 2007, Pontius et al. found that about 90% of the genes in the Abyssinian domestic cat are similar to humans [30]. In the real DNA sequences of our experiments, the segments of two yeast species *Saccharomyces cerevisiae* and *Kluyveromyces waltii* [18] has similarity higher than 88%.

However, for the case that sequence T is much longer than the length sum of A and B, our algorithms may not perform so well since $|T| = r$ becomes the dominant term in our time complexity $O(n|\Sigma| + (r - L + 1)Lm)$. The reason is that the extension of A and B is done along the progress of T. Fortunately, in the real applications, $\frac{|A|+|B|}{|T|}$ is not too large, neither too small.

It is worthy to apply our algorithm to other variants of the LCS problem, such as the constrained LCS problem. In addition, the efficiency of bit-parallel algorithms is increasing with technological advance. It may be interesting to speed-up our algorithms with the bit parallelism technique in the future.

References

[1] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, C.-Y. Hor, A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings, Inform. Process. Lett. 108 (2008) 360–364.
[2] A. Apostolico, Improving the worst-case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings, Inform. Process. Lett. 23 (1986) 63–69.
[3] A. Apostolico, Remark the HSU-DU new algorithm for the longest common subsequence problem, Inform. Process. Lett. 25 (1987) 235–236.
[4] A. Apostolico, S. Browne, C. Guerra, Fast linear-space computations of longest common subsequences, Theoret. Comput. Sci. 92 (1992) 3–17.
[5] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, E.S. Lander, Human and mouse gene structure: comparative analysis and application to exon prediction, Genome Res. 10 (7) (2000) 950–958.
[6] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proceedings of the Seventh International Symposium on String Processing Information Retrieval, SPIRE'00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 39–48.
[7] M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, J.F. Reid, A fast and practical bit-vector algorithm for the longest common subsequence problem, Inform. Process. Lett. 80 (2001) 279–285.
[8] A. Danek, S. Deorowicz, Bit-parallel algorithm for the block variant of the merged longest common subsequence problem, in: D. Gruca, T. Czachórski, S. Kozielski (Eds.), Man-Machine Interactions 3, in: Advances in Intelligent Systems and Computing, vol. 242, Springer, Cham, 2014, pp. 173–181.
[9] S. Deorowicz, A. Danek, Bit-parallel algorithms for the merged longest common subsequence problem, Internat. J. Found. Comput. Sci. 24 (2013) 1281–1298.
[10] Ensembl Fungi, e!EnsenmblFungi (Saccharomyces cerevisiae), http://fungi.ensembl.org/Saccharomyces_cerevisiae/Info/Index, 2017.
[11] S. Grabowski, New tabulation and sparse dynamic programming based techniques for sequence similarity problems, Discrete Appl. Math. 212 (2016) 96–103.
[12] J. Guo, F. Hwang, An almost-linear time and linear space algorithm for the longest common subsequence problem, Inform. Process. Lett. 94 (2005) 131–135.
[13] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.
[14] K. Hokamp, A. McLysaght, K.H. Wolfe, The 2R hypothesis and the human genome sequence, J. Struct. Funct. Genomics 3 (1–4) (2003) 95–110.
[15] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, Y.-H. Peng, Efficient algorithms for finding interleaving relationship between sequences, Inform. Process. Lett. 105 (2008) 188–193.
[16] K.-S. Huang, C.-B. Yang, K.-T. Tseng, Y.-H. Peng, H.-Y. Ann, Dynamic programming algorithms for the mosaic longest common subsequence problem, Inform. Process. Lett. 102 (2007) 99–103.
[17] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (5) (1977) 350–353.
[18] M. Kellis, B.W. Birren, E.S. Lander, Proof and evolutionary analysis of ancient genome duplication in the yeast saccharomyces cerevisiae, Nature 428 (6983) (2004) 617–624, http://www.nature.com/nature/journal/v428/n6983/full/nature02424.html.
[19] S. Kumar, C. Rangan, A linear space algorithm for the LCS problem, Acta Inform. 24 (1987) 353–362.
[20] Lab of Parallel Processing, Dept. of CSE, NSYSU, Merged and block merged LCS problems, http://par.cse.nsysu.edu.tw/~mlcs/, 2017.
[21] G.M. Landau, E. Myers, M. Ziv-Ukelson, Two algorithms for LCS consecutive suffix alignment, in: S.C. Sahinalp, S. Muthukrishnan, U. Dogrusoz (Eds.), Combinatorial Pattern Matching, CPM 2004, in: Lecture Notes in Computer Science, vol. 3109, Springer, Berlin, Heidelberg, 2004, pp. 173–193.
[22] J. Liu, G. Huang, Y. Wang, R. Lee, Edit distance for a run-length-encoded string and an uncompressed string, Inform. Process. Lett. 105 (2007) 12–16.
[23] M. Maes, On a cyclic string-to-string correction problem, Inform. Process. Lett. 35 (1990) 73–78.
[24] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distance, J. Comput. System Sci. 20 (1980) 18–31.
[25] E.W. Myers, An O(ND) difference algorithm and its variations, Algorithmica 1 (1986) 251–266.
[26] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inform. 18 (1982) 171–179.
[27] G. Panopoulou, A.J. Poustka, Timing and mechanism of ancient vertebrate genome duplications – the adventure of a hypothesis, Trends Genet. 21 (10) (2005) 559–567.
[28] M. Pawlik, N. Augsten, RTED: a robust algorithm for the tree edit distance, Proc. VLDB Endow. 5 (4) (2011) 334–345.
[29] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, C.-Y. Hor, Efficient sparse dynamic programming for the merged LCS problem with block constraints, Int. J. Innov. Comput. Inf. Control 6 (2010) 1935–1947.
[30] J.U. Pontius, J.C. Mullikin, D.R. Smith, A.S. Team, K. Lindblad-Toh, S. Gnerre, M. Clamp, J. Chang, R. Stephens, B. Neelam, N. Volfovsky, A.A. Schäffer, R. Agarwala, K. Narfström, W.J. Murphy, U. Giger, A.L. Roca, A. Antunes, M. Menotti-Raymond, N. Yuhki, J. Pecon-Slattery, W.E. Johnson, G. Bourque, G. Tesler, N.C.S. Program, S.J. O'Brien, Initial sequence and comparative analysis of the cat genome, Genome Res. 17 (11) (2007) 1675–1689.
[31] A.M. Rahman, M.S. Rahman, Effective sparse dynamic programming algorithms for merged and block merged LCS problems, J. Comput. 9 (8) (2014) 1743–1754.
[32] C. Rick, Simple and fast linear space computation of longest common subsequence, Inform. Process. Lett. 75 (2000) 275–281.
[33] C.-T. Tseng, C.-B. Yang, H.-Y. Ann, Efficient algorithms for the longest common subsequence problem with sequential substring constraints, J. Complexity 29 (1) (2013) 44–52.
[34] K.-T. Tseng, D.-S. Chan, C.-B. Yang, An efficient merged longest common subsequence algorithm for similar sequences, in: Proceedings of the 20th World Multi-Conference on Systemics, Cybernetics and Informatics, vol. I, WMSCI 2016, Orlando, Florida, USA, 2016, pp. 93–98.
[35] E. Ukkonen, Algorithm for approximate string matching, Inf. Control 64 (1985) 100–118.
[36] A. Varki, T.K. Altheide, Comparing the human and chimpanzee genomes: searching for needles in a haystack, Genome Res. 15 (12) (2005) 1746–1758.
[37] J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, R.J. Mural, G.G. Sutton, H.O. Smith, M. Yandell, C.A. Evans, R.A. Holt, J.D. Gocayne, P. Amanatides, R.M. Ballew, D.H. Huson, J.R. Wortman, Q. Zhang, C.D. Kodira, X.H. Zheng, L. Chen, M. Skupski, G. Subramanian, P.D. Thomas, J. Zhang, G.L. Gabor Miklos, C. Nelson, S. Broder, A.G. Clark, J. Nadeau, V.A. McKusick, N. Zinder, A.J. Levine, R.J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos,

A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran, R. Charlab, K. Chaturvedi, Z. Deng, V.D. Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A.E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T.J. Heiman, M.E. Higgins, R.-R. Ji, Z. Ke, K.A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G.V. Merkulov, N. Milshina, H.M. Moore, A.K. Naik, V.A. Narayan, B. Neelam, D. Nusskern, D.B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z.Y. Wang, A. Wang, X. Wang, J. Wang, M.-H. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S.C. Zhu, S. Zhao, D. Gilbert, S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M.L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doup, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. Howland, C. Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri, H. Qureshi, M. Reardon, R. Rodriguez, Y.-H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N.N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Winn-Deen, K. Wolfe, J. Zaveri, K. Zaveri, J.F. Abril, R. Guigó, M.J. Campbell, K.V. Sjolander, B. Karlak, A. Kejariwal, H. Mi, B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. Basu, J. Baxendale, L. Blick, M. Caminha, J. Carnes-Stine, P. Caulk, Y.-H. Chiang, M. Coyne, C. Dahlke, A.D. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, M. Peterson, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, X. Zhu, The sequence of the human genome, Science 291 (5507) (2001) 1304–1351.

[38] R. Wagner, M. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.