International Journal of Innovative
Computing, Information and Control
Volume **6**, Number **4**, April **2010**

ICIC International ©2010 ISSN 1349-4198
pp. 1935–1947

EFFICIENT SPARSE DYNAMIC PROGRAMMING FOR THE MERGED LCS PROBLEM WITH BLOCK CONSTRAINTS

Yung-Hsing Peng, Chang-Biau Yang*, Kuo-Si Huang, Chiou-Ting Tseng and Chiou-Yi Hor

Department of Computer Science and Engineering
National Sun Yat-sen University
No. 70, Lienhai Rd., Kaohsiung 80424, Taiwan
*Corresponding author: cbyang@cse.nsysu.edu.tw

Received September 2008; revised January 2009

Abstract. *Detecting the interleaving relationship between sequences has become important because of its wide applications to genomic and signal comparison. Given a target sequence T and two merging sequences A and B, recently Huang* et al. *propose algorithms for the merged LCS problem, without or with block constraint, whose aim is to find the longest common subsequence (LCS) with interleaving relationship. Without block constraint, Huang's algorithm requires $O(nmr)$-time and $O(mr)$-space, where $n = |T|$, m and r denote the longer and shorter length of A and B, respectively. In this paper, for solving the problem without block constraint, we first propose an algorithm with $O(Lnr)$ time and $O(m + Lr)$ space. We also propose an algorithm to solve the problem with block constraint. Our algorithms are more efficient than previous results, especially for sequences over large alphabets.*

Keywords: Algorithm, Dynamic programming, Longest common subsequence, Bioinformatics, Merged sequence

1. **Introduction.** In the field of sequence comparison, finding the *longest common subsequence* (LCS) between sequences is a classic approach for measuring the similarity of sequences. Given a sequence S, a subsequence $\bar{S}$ of S can be obtained by deleting zero or more characters from S. Given two sequences S_1 and S_2, the LCS of S_1 and S_2, denoted by $LCS(S_1, S_2)$, is the longest sequence $\bar{S}'$ such that $\bar{S}'$ is a subsequence of both S_1 and S_2.

Algorithms for finding the LCS of two or more sequences have been extensively studied for several decades [1, 2, 3]. Most of these algorithms are based on either traditional dynamic programming [1] or sparse dynamic programming [4, 5, 6]. Traditional dynamic programming, which compares each character in S_1 with each character in S_2, takes $O(|S_1||S_2|)$ time, where $|S_1|$ and $|S_2|$ denote the lengths of S_1 and S_2, respectively. For alphabets that cannot be sorted, this approach is optimal [7]. For sortable alphabets, however, sparse dynamic programming is faster [5, 6], because the information of matches can be obtained more efficiently.

In addition to the traditional LCS problem, some extended versions, such as the constrained LCS problem [8, 9, 3, 10, 11, 12] and the mosaic LCS problem [13, 14], have been proposed to provide more flexible comparison on sequences. Recently, Huang *et al.* [15] proposed one interesting problems: *the merged LCS problem*, which is to detect the interleaving relationship between sequences. In the problem, the block constraint may be involved.

In biology, finding the interleaving relationship of sequences can be realized as detecting the *synteny* phenomenon, which means the order of specific genes in chromosome is

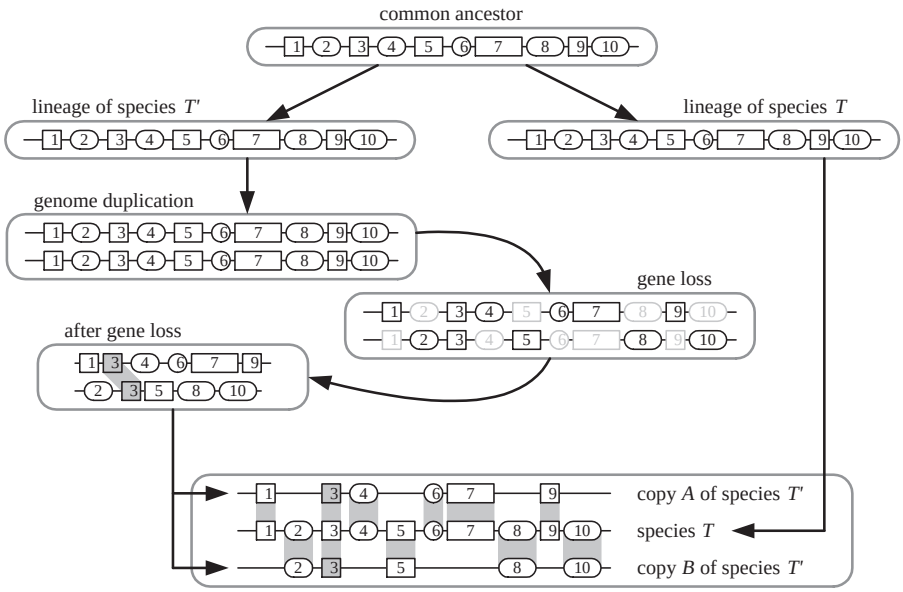

FIGURE 1. A simplified diagram of DCS block and WGD.

conserved over different organisms [16, 17, 18]. A typical example for this phenomenon can be seen between two yeast-species *Kluyveromyces waltii* and *Saccharomyces cerevisiae* [17]. By detecting the doubly conserved synteny (DCS) blocks of the two species, where each region of *K. waltii* corresponds to one region out of two yeasts in *S. cerevisiae*, Kellis et al. [17] obtain the support for the whole-genome duplication (WGD) or two rounds of gene duplication (2R) hypothesis [19, 20]. Figure 1 shows a simplified diagram of DCS block and WGD, which is an important application of the merged LCS problem.

Another application of the merged LCS problem is the signal comparison. For example, the merged LCS enables one to identify the similarity between a complete voice (one series of signals without noise), and two incomplete voices (two series of signals sampled with different noise). Therefore, detecting the interleaving relationship between sequences is very realistic, for both fixed alphabets (genes) and large alphabets (signals with various values).

Note that the alphabet is sortable for most cases. Therefore, for the merged LCS problem, without or with block constraint, it is still worth considering how to improve the existing results [15] by a sparse dynamic programming. However, no existing algorithms can be applied to achieve this goal directly. Though it is also interesting to solve these problems by machine learning [21] or iterative optimization [22, 23], these approaches may not obtain the optimal solution. To ensure that the optimal solution is always obtained, we focus on theoretical improvements of existing results [15]. In this paper, we propose the first known algorithm that solves the merged LCS problem with sparse dynamic programming. The proposed algorithm can be easily extended to solve the block-merged LCS problem. These new algorithms improve Huang's results [15], especially for sequences over large alphabets.

The rest of this paper is organized as follows. Section 2 provides explanations and annotations for the merged LCS problem and the block-merged LCS problem. Section 3 describes a simple linear time strategy for finding two-dimensional minima of non-negative integer points. In Section 4, we use the result of Section 3 to solve the merged

LCS problem. In Section 5, we extend the algorithms in Section 4 to solve the block-merged LCS problem. The analysis for large alphabets can be found in the ending part of Section 5. Section 6 concludes and discusses future work.

2. The LCS with Merging Sequences.

2.1. **The Merged LCS Problem.** Given a target sequence T and two merging sequences A and B, the *merged LCS problem*, denoted as $MLCS(T, A, B)$, is to determine the LCS of T and $A \oplus B$, where '$\oplus$' denotes the merging operation which merges A and B into an interleaving sequence. Note that the result of $A \oplus B$ may not be unique, which means the problem is not trivial. For example, suppose we have $A = a_1a_2a_3a_4 =$ actt and $B = b_1b_2b_3 =$ ctg, then three of the possible merging sequences are $M_1 = a_1a_2b_1b_2a_3a_4b_3 =$ accttg, $M_2 = a_1a_2a_3a_4b_1b_2b_3 =$ acttctg, and $M_3 = b_1a_1b_2a_2b_3a_3a_4 =$ catcgtt. Further, suppose that the target sequence is $T =$ tcatcga. Then, by examining all merging sequences, it can be seen that one of the optimal solutions of $MLCS(T, A, B)$ is $LCS(T, M_3) =$ catcg, whose length is 5. This example illustrates that the MLCS approach is useful for measuring the interleaving relationship between sequences.

Based on dynamic programming, Huang *et al.* give an on-line $O(nmr)$-time algorithm [15] for determining $MLCS(T, A, B)$, whose off-line implementation requires $O(mr)$ space, where n, m and r denote the length of T, the longer and shorter lengths of A and B, respectively. In Section 4, we propose an improved $O(Lnr)$-time on-line algorithm for solving $MLCS(T, A, B)$, whose off-line implementation requires $O(m + Lr)$ space, where $L = |MLCS(T, A, B)|$ denotes the length of the optimal solution for the merged LCS problem. For any alphabet whose size is asymptotically linear to m, our improvement is significant because $O(L)$ is expected as $O(\sqrt{m})$ [24]. Also, our $O(Lnr)$-time algorithm achieves the best known result when L is relatively small.

2.2. **The Block-merged LCS Problem.** For genomic or signal comparison that is restricted with block constraints [17], Huang *et al.* further propose the block-merged LCS model. Different from the merged LCS problem, the block-merged LCS problem asks for the LCS of T and $A \otimes B$, defined as $MLCS^{\#}(T, A, B)$, where '$\otimes$' denotes the merging operation with blocks. To give an example, let A= actt, as in the previous example. Now we divide $A =$ actt into two blocks X_1= ac# and X_2 = tt#, represented as $A = X_1X_2 =$ ac#tt#, where '#' denotes the *dividing symbol* that ends a block. Suppose $B =$ ctg is also divided into two blocks $B = Y_1Y_2 =$ ct#g#. In this case, three of the possible merging sequences with blocks are $M_1^{\#} = X_1Y_1X_2Y_2 =$ acctttg, $M_2^{\#} = X_1X_2Y_1Y_2 =$ acttctg, and $M_3^{\#} = Y_1X_1Y_2X_2 =$ ctacgtt. Similarly, by checking all merging sequences of $A \otimes B$, one can see that both $LCS(T, M_2^{\#}) =$ atcg and $LCS(T, M_3^{\#}) =$ ctcg are optimal solutions of $MLCS^{\#}(T, A, B)$. Note that for this case, the merging sequence $M_3 = b_1a_1b_2a_2b_3a_3a_4$ in our previous example cannot be obtained from $A \otimes B$, because M_3 breaks the blocks.

By considering the ends of blocks, Huang *et al.* derive an $O(nm(|A^{\#}| + |B^{\#}|))$-time on-line algorithm [15] for solving the block-merged LCS problem, where $|A^{\#}|$ and $|B^{\#}|$ denote the number of blocks in A and B, respectively. In addition, by using the technique of S-table [25, 26], they further propose an off-line algorithm that requires only $O(nm + n|A^{\#}||B^{\#}|)$ time and space. Let $L' = |MLCS^{\#}(T, A, B)|$ be the length of the optimal solution for the block-merged LCS problem. In Section 5, we will explain how to determine $MLCS^{\#}(T, A, B)$ with $O(min\{L'n(|A^{\#}| + |B^{\#}|), n(|A^{\#}|r + |B^{\#}|m)\})$ time and $O(m + L'(|A^{\#}| + |B^{\#}|))$ space.

3. **The Two-dimensional Minima of Non-negative Integer Points.** In this section, we show that the two-dimensional minima of non-negative integer points can be determined efficiently. This property will be used for our algorithm in the next section. The definition of a minimum in a set of two-dimensional points is given as follows.

Definition 3.1. *Given a set Q of two-dimensional points, a two-dimensional minimum of Q is a point $(x, y) \in Q$ such that $x' > x$ or $y' > y$, for any point $(x', y') \in Q - \{(x, y)\}$.*

When the points in Q are integer points with boundaries, the set of two-dimensional minima in Q, defined as $MIN(Q)$, can be determined with bucket sort. Let each point in Q be bounded by non-negative integers Z_x and Z_y, which means that $0 \leq x \leq Z_x$ and $0 \leq y \leq Z_y$, for any $(x, y) \in Q$. In this case, we have the following result.

Lemma 3.1. *For any set Q of two-dimensional integer points bounded by non-negative integers Z_x and Z_y, the determination of $MIN(Q)$ can be achieved in $O(|Q|+min(Z_x, Z_y))$ time.*

Proof: Without loss of generality, suppose $Z_x \leq Z_y$. Applying the bucket sort with respect to x-axis, it takes $O(|Q| + Z_x)$ time to divide Q into $(Z_x + 1)$ sets $Q_0, Q_1, \cdots, Q_{Z_x}$. Since it takes $O(|Q_i|)$ time to determine the unique two-dimensional minimum $MIN(Q_i) = (i, y_i)$, for $0 \leq i \leq Z_x$, the overall time for determining all such (i, y_i) would be $O(|Q|+Z_x)$. If Q_i is an empty set, then $(i, y_i) = (\infty, \infty)$. Now, it is clear that $MIN(Q)$ can be obtained with an $O(Z_x)$-time linear scan on $MIN(Q_0), MIN(Q_1), \cdots, MIN(Q_{Z_x})$. Therefore, the lemma holds. □

For the case of real points, known sorting algorithms allow $MIN(Q)$ to be determined in $O(|Q| \log |Q|)$ time. Here we omit the algorithm for real points, because Lemma 3.1 is sufficient to support our algorithms in Section 4.

4. **Algorithms for the Merged LCS Problem.** In this section, we solve the merged LCS problem in $O(Lnr)$ time. Briefly, our algorithm solves the problem by locating proper indices (candidates) with two-dimensional minima. To improve the efficiency of our algorithm, we also show how to reduce the space complexity to $O(min\{Lnr, Lmr\})$ for on-line applications, and $O(m + Lr)$ for off-line applications.

4.1. **Locating Candidates.** For a sequence S, let $S[i]$ denote the ith character in S. Also, let $S[i, j]$ denote the substring in S that ranges from $S[i]$ to $S[j]$, for any $1 \leq i \leq j \leq |S|$. Further, define $S[0]$ as an empty character and $S[i, j]$ as an empty string for any $i > j$. Also, let $S_1 + S_2$ denote the concatenated sequence for two sequences S_1 and S_2. In addition, we define the relationship between a pair of distinct 2-tuple numbers as follows.

Definition 4.1. *Given a pair of 2-tuple numbers (i, j) and (i', j'), we say that $(i', j') < (i, j)$ if $i' \leq i$ and $j' \leq j$ for $(i, j) \neq (i', j')$.*

In the following, we explain the idea of identifying candidates. Suppose we have the target sequence $T = t_1 t_2 \cdots t_n$, two merging sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_r$ with $m \geq r$. An (l, k)-*candidate* is defined as follows.

Definition 4.2. *For $0 \leq i \leq m$, $0 \leq j \leq r$ and $0 \leq l \leq k \leq n$, a pair of integers (i, j) is an **(l, k)-candidate** if the following conditions hold. (1)$|MLCS(T[1, k], A[1, i], B[1, j])| = l$ (2)For any integer $(i', j') < (i, j)$, $|MLCS(T[1, k], A[1, i'], B[1, j'])| < l$. Such an (l, k)-candidate is also called a **dominating candidate**.*

According to Definition 4.2, there exist some (L, k)-candidates (i, j) such that $L = |MLCS(T[1, k], A[1, i], B[1, j])| = |MLCS(T, A, B)|$, for $0 \leq k \leq n$, $0 \leq i \leq m$ and $0 \leq j \leq r$. Based on this idea, a lemma follows.

Lemma 4.1. *There exists an increasing sequence of dominating candidates* $C = (i_0, j_0) < (i_1, j_1) < \cdots < (i_L, j_L)$ *and indices* $K = k_0, k_1, \cdots, k_L$, *where* $i_0 = j_0 = k_0 = 0$, *such that* (i_p, j_p) *is a* (p, k_p)*-candidate,* $0 \le p \le L$, *and* $T[k_1], T[k_2], \cdots, T[k_L]$ *is the LCS of* T *and* $A \oplus B$.

Proof: The correctness of this lemma can be verified with recursion. □

In the following, we explain how to construct these dominating candidates and indices efficiently, so as to determine the LCS of T and $A \oplus B$.

4.2. **The On-line Algorithm for Merged LCS.** We first introduce some notations. For any symbol σ, let $next_A(\sigma, i) = i'$ denote the minimum index $i' > i$ in sequence A such that $A[i'] = \sigma$. If there is no such index, set $i' = \infty$. Also, let $next_B(\sigma, j)$ serve the same purpose for sequence B. Suppose the alphabet is of fixed size. Thus, with an $O(m + r)$-time preprocessing that constructs a mapping table on A and B, both $next_A(\sigma, i)$ and $next_B(\sigma, j)$ can be determined in constant time, for any $0 \le i \le m$, $0 \le j \le r$. Let $H_{l,k}$ denote the set of (l, k)-candidates. Also, let $H'_{l,k}$ be a set constructed by substituting each $(i, j) \in H_{l,k}$ with two integer points $(next_A(T[k+1], i), j)$ and $(i, next_B(T[k+1], j))$. Any point $(x, y) \in H'_{l,k}$ is excluded from $H'_{l,k}$ if $x = \infty$ or $y = \infty$. Hence, we have $|H'_{l,k}| \le 2|H_{l,k}|$. Note that each (x, y) in $H'_{l,k}$ is now bounded by $Z_x = |A| = m$ and $Z_y = |B| = r$. Besides, $H_{l,k}$ can also be deemed as a set of integer points bounded by $Z_x = m$ and $Z_y = r$.

Algorithm 1 Finding the merged LCS in $O(Lnr)$ time

```
Set len ← 1 and H_{0,k} ← {(0,0)}, for 0 ≤ k ≤ n.
for k = 1 to k = n do
  for l = 1 to l = len do
    Construct H'_{l-1,k-1} from H_{l-1,k-1} with next
    H_{l,k} ← MIN(H'_{l-1,k-1} ∪ H_{l,k-1})
  end for
  if |H_{len,k}| > 0 then
    len ← len + 1
  end if
end for
L ← len − 1
Retrieve the LCS by tracking from H_{L,n} to H_{0,0}.
```

Algorithm 1 provides an online method for solving the merged LCS problem in $O(Lnr)$ time. To prove the correctness of Algorithm 1, in the following we derive some useful properties of dominating candidates.

Lemma 4.2. *For any dominating candidate* $(i, j) \in H_{l,k}$ *with* $l \ge 1$ *and* $k \ge 1$, *one of the following conditions holds.*
(1)There exists $(\bar{i}, \bar{j}) \in H_{l-1,k-1}$ *such that* $i = next_A(T[k], \bar{i})$ *and* $j = \bar{j}$.
(2)There exists $(\bar{i}, \bar{j}) \in H_{l-1,k-1}$ *such that* $i = \bar{i}$ *and* $j = next_B(T[k], \bar{j})$.
(3)There exists $(\bar{i}, \bar{j}) \in H_{l,k-1}$ *such that* $i = \bar{i}$ *and* $j = \bar{j}$.

Proof: The proof of this lemma can be done by discussing whether $T[k]$ is picked as the lth common character, for each candidate $(i, j) \in H_{l,k}$. If $T[k]$ is picked as the lth common character, then $T[k]$ must be picked with either $A[i]$ or $B[j]$. Otherwise, there will be some integers $i' \le i - 1$, $j' \le j - 1$ such that $|MLCS(T[1, k], A[1, i'], B[1, j'])| = l$, which is contradictory to that (i, j) is an (l, k)-candidate. Clearly, picking $T[k]$ with $A[i]$

or $B[j]$ leads to condition (1) or (2), respectively. Also, one can see that condition (3) holds if $T[k]$ is not picked as the lth common character. □

Combining Definitions 3.1, 4.2 and Lemma 4.2, we have $H_{l,k} = MIN(H'_{l-1,k-1} \cup H_{l,k-1})$, which is the set of minima in $H'_{l-1,k-1} \cup H_{l,k-1}$. In the following, we shall show that each $H_{l,k}$ can be obtained in $O(r)$ time.

Lemma 4.3. *For $1 \le l \le L$ and $1 \le k \le n$, $H_{l,k}$ can be obtained in $O(r)$ time when $H_{l-1,k-1}$ and $H_{l,k-1}$ are given.*

Proof: With the pigeonhole principle, one can see that both $H_{l-1,k-1}$ and $H_{l,k-1}$ contain no more than $r+1$ candidates. Because $|H'_{l-1,k-1}| \le 2|H_{l-1,k-1}|$, it is clear that $|H'_{l-1,k-1} \cup H_{l,k-1}| \le 3(r+1)$. With Lemma 3.1, $H_{l,k}$ can be obtained in $O(|H'_{l-1,k-1} \cup H_{l,k-1}| + min(m,r)) = O(r)$ time. □

By Lemmas 4.2 and 4.3, one can see that Algorithm 1 determines $H_{L,n}$ in $O(Lnr)$ time. Based on Lemma 4.1, it is easy to design an $O(L+n)$-time backtracking algorithm for Algorithm 1. With additional analysis for the required space, our first result is given in Theorem 4.1.

Theorem 4.1. *$MLCS(T, A, B)$ can be determined on-line respect to T with $O(Lnr)$ time and $O(min\{Lnr, Lmr\})$ space.*

Proof for the space complexity: Given $(i,j) \in H_{l,k}$, let $Back_{l,k}(i,j)$ denote the query function for tracking the previous dominating candidate which generates (i,j). That is, we have $Back_{l,k}(i,j) = (i',j')$ if (i,j) is generated by $(i',j') \in H_{l-1,k-1}$, satisfying either $(next_A(T[k], i'), j') = (i,j)$ or $(i', next_B(T[k], j')) = (i,j)$. Otherwise, we have $Back_{l,k}(i,j) = (i,j)$ because $(i,j) \in H_{l,k}$ is generated by $(i,j) \in H_{l,k-1}$.

A straightforward implementation would fully update the tracking function $O(Lnr)$ times, which requires $O(Lnr)$ space to retrieve the LCS. However, the tracking function need not be updated with $Back_{l,k}(i,j) = (i',j')$ if for any $k' < k$ there already exists $(i,j) \in H_{l,k'}$. That is, the query $Back_{l,k}(i,j)$ can be replaced with $Back_{l,k'}(i,j)$, if a common subsequence of length l can be obtained from $T[1,k']$ and $A[1,i] \oplus B[1,j]$, for any $k' < k$. One can see that with $Back_{l,k'}(i,j)$, the retrieved LCS is still of length l. Therefore, for any triplet (i,j,l), we need to update the tracking function only one time for the minimum k that $(i,j) \in H_{l,k}$, denoted as $MK(i,j,l)$. Since the number of (i,j,l)-triplets is bounded by $O(Lmr)$, the required space for storing the tracking function is also bounded by $O(Lmr)$. Note that all $Back_{l,k}$ and MK can be implemented with a table of $(m+1) \times (r+1)$ grids, where each grid $G_{i,j}$ contains a linked list which stores each $MK(i,j,l)$ with increasing l. That is, the lth element in $G_{i,j}$ stores the minimum k such that $(i,j) \in H_{l,k}$. Therefore, to update $Back_{l,k}(i,j) = (i',j')$, one can first append to $G_{i,j}$ a new element which stores k, then add a link from the last (lth) element in $G_{i,j}$ to the last ($(l-1)$th) element in $G_{i',j'}$, which takes constant time.

The correctness of this implementation can be verified with the observation that for any $(i,j) \in H_{\bar{l},\bar{k}}$ and $(i,j) \in H_{l,k}$ which result in two different updates, we have $\bar{l} < l$ if and only if $\bar{k} < k$. Therefore, the upper bound of the required space is linear to the total number of generated pairs, which is known to be $O(Lnr)$. Note that building the table of $(m+1) \times (r+1)$ grids takes $\Omega(mr)$ time and space. Therefore, the required time and space are $O(Lnr + mr)$ and $O(min\{Lnr + mr, Lmr\})$, respectively. However, for on-line applications it is suitable to assume $Ln > m$, since n still grows whereas m is given. Therefore, by omitting the additional $O(mr)$ complexity, one can see that the theorem holds. □

Our backtracking algorithm is summarized as Algorithm 2, which serves as the supplement to Algorithm 1. One can see that Algorithm 2 retrieves the LCS in $O(L)$ time,

which is also an improvement to the $O(n+L)$-time straightforward implementation. The space efficiency of Theorem 4.1 would also be substantial for on-line applications when n is much greater than m.

Algorithm 2 Retrieving the LCS in $O(L)$ time

```
Set l ← L and k ← n.
Pick an arbitrary candidate (i, j) ∈ H_{L,n}.
while l ≠ 0 do
  k ← MK(i, j, l)
  (i', j') ← Back_{l,k}(i, j)
  if i' < i then
    Report A[i] and T[k] as the lth common character.
  else
    Report B[j] and T[k] as the lth common character.
  end if
  (i, j) ← (i', j')
  l ← l − 1
end while
```

4.3. **The Off-line Implementation.** If the on-line process on T is not required, we can give a more space-efficient implementation with $O(m+Lr)$ space, provided that the length of T is given beforehand. Though, under this assumption, the proposed algorithm becomes off-line to T, an $O(m + Lr)$-space implementation is necessary in some circumstances. Briefly, our off-line algorithm is an extension to Hirschberg's divide-and-conquer algorithm [1]. However, we avoid the reverse computation. This makes our algorithm easy to implement.

Theorem 4.2. *$MLCS(T, A, B)$ can be determined with $O(Lnr)$ time and $O(m + Lr)$ space.*

Proof: Note that for any specific k, we have $\sum_{l=1}^{L} |H_{l,k}| \leq L(r + 1)$. That is, it takes $O(Lr)$ space to store all $H_{l,\lfloor \frac{n}{2} \rfloor}$, for $1 \leq l \leq L$. Here we treat the case $L = 0$ as the boundary condition that needs no further tracking. Since $H_{l,k} = MIN(H'_{l-1,k-1} \cup H_{l,k-1})$, any tracking path from $H_{1\leq l\leq L,\lfloor \frac{n}{2} \rfloor \leq k\leq n}$ to $H_{0,0}$ must pass through a unique dominating candidate $(i_b, j_b) \in H_{l_b,\lfloor \frac{n}{2} \rfloor}$ with a unique length l_b, $0 \leq l_b \leq L$. In other words, this unique (i_b, j_b) can be used as the breaking point in Hirschberg's divide-and-conquer strategy. Let $Mid_{l,k}(i, j)$ denote such a unique dominating candidate obtained by tracking from $(i, j) \in H_{l,k\geq\lfloor \frac{n}{2} \rfloor}$ to $H_{0,0}$. Next, with the following formulas, we show how to obtain $Mid_{l,k}(i, j)$ without reverse computation.
(1)$Mid_{l,\lfloor \frac{n}{2} \rfloor}(i, j) = (i, j)$.
(2)For $\lfloor \frac{n}{2} \rfloor < k \leq n$, $Mid_{l,k}(i, j) = Mid_{l,k-1}(i, j)$ if $(i, j) \in H_{l,k}$ and $(i, j) \in H_{l,k-1}$.
(3)Otherwise, let $(i', j') = Back_{l,k}(i, j)$, we have $Mid_{l,k}(i, j) = Mid_{l-1,k-1}(i', j')$.

The correctness of the above formula can be easily seen, since it is a simple strategy which passes down the information of the demanded candidates. For all candidates $(\hat{i}, \hat{j}) \in H_{L,n}$, $Mid_{L,n}(\hat{i}, \hat{j})$ can be obtained with $O(Lnr)$ time and $O(Lr)$-space. Suppose $(i_b, j_b) \in H_{l_b,\lfloor \frac{n}{2} \rfloor} = Mid_{L,n}(\hat{i}, \hat{j})$ is the demanded candidate for some $(\hat{i}, \hat{j}) \in H_{L,n}$. Let $T_\alpha = T[1, \lfloor \frac{n}{2} \rfloor - 1]$, $T_\beta = T[\lfloor \frac{n}{2} \rfloor + 1, n]$, $A_\alpha = A[1, i_b]$, $A_\beta = A[i_b + 1, m]$, $B_\alpha = B[1, j_b]$, $B_\beta = B[j_b + 1, r]$ be six divided sequences, the problem can now be split according to different conditions.

(1)If $(i_b, j_b) \in H_{l_b,\lfloor\frac{n}{2}\rfloor-1}$, then we have $|MLCS(T[1, \lfloor\frac{n}{2}\rfloor - 1], A[1, i_b], B[1, j_b])| = l_b$ and $|MLCS(T[\lfloor\frac{n}{2}\rfloor + 1, n], A[i_b + 1, m], B[j_b + 1, r])| = L - l_b$. That is, $MLCS(T, A, B)$ can be solved by computing $MLCS(T_\alpha, A_\alpha, B_\alpha)$+$MLCS(T_\beta, A_\beta, B_\beta)$. Note that it takes only constant time to determine whether (i_b, j_b) is in $H_{l_b,\lfloor\frac{n}{2}\rfloor-1}$, since one can pass down the answer along with the Mid function.
(2)If $(i_b, j_b) \notin H_{l_b,\lfloor\frac{n}{2}\rfloor-1}$, then based on the proof of Lemma 4.1, either $A[i_b]$ or $B[j_b]$ must be picked with $T[\lfloor\frac{n}{2}\rfloor]$ to form the l_bth common character. Suppose $A[i_b]$ is picked, then the whole longest common subsequence $MLCS(T, A, B)$ must be of the form $MLCS(T_\alpha, A_\alpha[1, i_b - 1], B_\alpha) + T[\lfloor\frac{n}{2}\rfloor] + MLCS(T_\beta, A_\beta, B_\beta)$. Similarly, one can see that $MLCS(T, A, B) = MLCS(T_\alpha, A_\alpha, B_\alpha[1, j_b - 1]) + T[\lfloor\frac{n}{2}\rfloor] + MLCS(T_\beta, A_\beta, B_\beta)$ if $B[j_b]$ is picked. With properties $|T_\alpha| \le \lfloor\frac{n}{2}\rfloor$, $|T_\beta| \le \lfloor\frac{n}{2}\rfloor$, $|A_\alpha| + |A_\beta| = m$, $|B_\alpha| + |B_\beta| = r$ and $l_b + (L - l_b) = L$, we get that $rl_b|T_\alpha| + r(L - l_b)|T_\beta| \le \frac{Lnr}{2}$.

The above discussion indicates that the overall time for solving the subproblems is always bounded by half of that needed to solve $MLCS(T, A, B)$. Therefore, this implementation takes $O(Lnr)$ time but $O(Lr)$ space. Note that it still takes $O(m + r)$ space to construct the functions $next_A$ and $next_B$. Therefore, the overall space complexity is $O(m + r + Lr) = O(m + Lr)$. □

In the worst case, this off-line implementation takes no more than $O(mr)$ space, because $L \le 2m$. In some cases, however, the required space is much less. If, for example, $L = r = \sqrt{m}$, then our implementation takes only $O(m)$ space, rather than $O(mr) = O(m^{1.5})$ space. In addition, for the case that L is relatively small, the required space reduces to near $O(m + r)$.

5. Extending the Algorithm with Block Constraints.

In this section, we show how to extend the algorithms proposed in the previous section to solve the problem with block constraints. Our extension is done by first relating $MLCS^{\#}(T, A, B)$ to dominating candidates, and then showing that these new dominating candidates can still be obtained with a similar sparse dynamic programming.

5.1. Candidates with Block Constraints.

Recall that the symbol $'\#'$ means the end of a block. To fit the problem, we assume that '#' does not appear in T, and that each block ends with a dividing symbol. That is, we have $A[m]$='#' and $B[r]$='#' if both A and B contain at least one block. For the case of an empty block, we assume it ends with an empty character. For any $A[i] \neq$'#' and $B[j] \neq$'#', let $num^{\#}_A(i)$ and $num^{\#}_B(j)$ denote the number of '#'s in $A[1, i]$ and $B[1, j]$, respectively. That is, for any $1 \le i \le m$, $num^{\#}_A(i)$ means the block index in A where $A[i]$ can be located. Therefore, $A[i_1]$ and $A[i_2]$ are in the same block if and only if $num^{\#}_A(i_1) = num^{\#}_A(i_2)$, for $1 \le i_1, i_2 \le m$. In T, let $k_1 \le k' \le k_2$ be three arbitrary indices of the LCS obtained from T and $A \otimes B$. Clearly, any LCS obtained from T and $A \otimes B$ must satisfy the following two conditions:
(1)For any $A[i_1]$ and $A[i_2]$ picked with $T[k_1]$ and $T[k_2]$, respectively, $T[k']$ cannot be picked with any $B[j]$, for $1 \le j \le r$, if $num^{\#}_A(i_1) = num^{\#}_A(i_2)$.
(2)For any $B[j_1]$ and $B[j_2]$ picked with $T[k_1]$ and $T[k_2]$, respectively, $T[k']$ cannot be picked with any $A[i]$, for $1 \le i \le m$, if $num^{\#}_B(j_1) = num^{\#}_B(j_2)$.

Let $K' = k'_0, k'_1, k'_2, \cdots k'_{L'}$ be some indices of T such that $T[k'_1], T[k'_2], \cdots, T[k'_{L'}]$ is an LCS obtained from $MLCS^{\#}(T, A, B)$. For $L' = 0$, let $k'_0 = 0$ represent the empty index which means no common subsequence can be found between T and $A \otimes B$. The following lemma describes the relationship between $MLCS^{\#}(T, A, B)$ and dominating candidates.

Lemma 5.1. *For any sequence of indices $K' = k'_0, k'_1, k'_2, \cdots k'_{L'}$ where $T[k'_1], T[k'_2], \cdots, T[k'_{L'}]$ is an LCS obtained from $MLCS^{\#}(T, A, B)$, there exists a sequence of pairs $C^{\#} =$*

$(i'_0, j'_0), (i'_1, j'_1), (i'_2, j'_2), \cdots, (i'_{L'}, j'_{L'})$ *such that for* $1 \le p \le L'$, *(1)*$(i'_{p-1}, j'_{p-1}) < (i'_p, j'_p)$ *and (2)for each* $(i'_p, j'_p) \in C^{\#}$, *one of* $A[i'_p]$ *and* $B[j'_p]$ *serves as the pth common character with* $T[k'_p]$ *while the other must be a dividing symbol '#' or an empty character.*

Proof: This lemma can be verified by a recursion considering block constraints. □

To support the algorithm we are to describe, we give a conditional definition of $X \otimes Y$ for two given sequences X and Y, which does not alter Huang's derivation but will simplify our further presentation. Let X' and Y' denote the longest prefixes of X and Y that end with '#', respectively.

Definition 5.1. *(1)If* $|X| = 0$ *or* $|Y| = 0$, *let* $X \otimes Y = X + Y$. *(2)For* $X[|X|]$='#' *and* $Y[|Y|]$='#', *let* $X \otimes Y$ *denote a merged sequence of* X *and* Y *with block constraints. (3)For* $X[|X|] \neq$'#' *and* $Y[|Y|] =$'#', *let* $X \otimes Y = (X' \otimes Y) + X[|X'| + 1, |X|]$. *(4)For* $X[|X|] =$'#' *and* $Y[|Y|] \neq$'#', *let* $X \otimes Y = (X \otimes Y') + Y[|Y'| + 1, |Y|]$.

The first two conditions in Definition 5.1 describe Huang's derivation of block constraints [15], while the third and the fourth conditions are mainly used for supporting our algorithm. Definition 5.1 does not consider the case where both $X[|X|] \notin \{\text{'\#'}, \phi\}$ and $Y[|Y|] \notin \{\text{'\#'}, \phi\}$, because it will be excluded by the following definition of an $(l, k)^{\#}$-candidate.

Definition 5.2. *For* $0 \le i \le m$, $0 \le j \le r$ *and* $0 \le l \le k \le n$, *a pair of integers* (i, j) *is called an* $\boldsymbol{(l, k)^{\#}}$***-candidate*** *if the following three conditions hold. (1)*$A[i] \in \{\text{'\#'}, \phi\}$ *or* $B[j] \in \{\text{'\#'}, \phi\}$. *(2)*$|MLCS^{\#}(T[1, k], A[1, i], B[1, j])| = l$ *(3)For any pair of integers* (i', j') *where* $A[i'] \in \{\text{'\#'}, \phi\}$ *or* $B[j'] \in \{\text{'\#'}, \phi\}$, $|MLCS^{\#}(T[1, k], A[1, i'], B[1, j'])| < l$ *if* $(i', j') < (i, j)$.

In fact, Definition 5.2 is an extension of Definition 4.2 to include block constraints.

5.2. **Extended On-line and Off-line Algorithms.** Now we explain how to determine $MLCS^{\#}(T, A, B)$ by a sparse dynamic programming that locates candidates. The new sparse dynamic programming is obtained with a slight modification to the result of Section 4. Let $H^{\#}_{l,k}$ denote the set of $(l, k)^{\#}$-candidates. Also, let $H'^{\#}_{l,k}$ be the set obtained by replacing each candidate $(i, j) \in H^{\#}_{l,k}$ with two pairs of integers according to the following rules.

(1)For $(i, j) = (0, 0)$:
Replace (i, j) by $(0, next_B(T[k + 1], 0))$ and $(next_A(T[k + 1], 0), 0)$.
(2)For $A[i] \notin \{\text{'\#'}, \phi\}$:
Replace (i, j) by $(next_A(T[k + 1], i), j)$ and $(next_A(\text{'\#'}, i), next_B(T[k + 1], j))$.
(3)For $B[j] \notin \{\text{'\#'}, \phi\}$:
Replace (i, j) by $(i, next_B(T[k + 1], j))$ and $(next_A(T[k + 1], i), next_B(\text{'\#'}, j))$.

Any $(x, y) \in H'^{\#}_{l,k}$ is excluded if $x = \infty$ or $y = \infty$. Here we have $|H^{\#}_{l,k}| \le (|A^{\#}| + |B^{\#}| + 2)$, because for each $(i, j) \in H^{\#}_{l,k}$, either $A[i]$ or $B[j]$ is an end of block. Since $|H'^{\#}_{l,k}| \le 2|H^{\#}_{l,k}|$, both $|H^{\#}_{l,k}|$ and $|H'^{\#}_{l,k}|$ are bounded by $O(|A^{\#}| + |B^{\#}|)$. Based on the idea in Section 4, each $H^{\#}_{l,k}$ can be constructed recursively.

Lemma 5.2. $H^{\#}_{l,k} = MIN(H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1})$, *for* $1 \le l \le k \le n$.

Proof: This lemma is correct based on the result of Section 4. □

Nonetheless, note that Lemma 3.1 cannot be directly applied to obtain $H^{\#}_{l,k}$, since it still takes $O(|A^{\#}| + |B^{\#}| + r)$ time even if $|H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1}|$ is bounded by $O(|A^{\#}| + |B^{\#}|)$. In the following, we propose the last trick, which guarantees that $H^{\#}_{l,k}$ can be obtained in $O(|A^{\#}| + |B^{\#}|)$ time.

Lemma 5.3. *For $1 \le l \le k \le n$, each $H^{\#}_{l,k}$ can be determined with $O(|A^{\#}| + |B^{\#}|)$ time if both $H^{\#}_{l-1,k-1}$ and $H^{\#}_{l,k-1}$ are given.*

Proof: Let $Q_{A_0}, Q_{A_1}, Q_{A_2}, \cdots, Q_{A_{|A^{\#}|}}$ and $Q_{B_0}, Q_{B_1}, Q_{B_2}, \cdots, Q_{B_{|B^{\#}|}}$ denote distinct sets of integer pairs which are constructed from $H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1}$ with the following two rules.
(1)For any $(i,j) \in H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1}$, if $A[i] \in \{'\#'\phi\}$, then $(i,j) \in Q_{A_I}$, where $I = num^{\#}_A(i)$.
(2)Otherwise, we have $B[j] \in \{'\#'\phi\}$ and $(i,j) \in Q_{B_J}$, where $J = num^{\#}_B(j)$.

Based on these rules, one can partition $H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1}$ into $(|A^{\#}| + |B^{\#}| + 2)$ sets in $O(|A^{\#}| + |B^{\#}|)$ time. Keeping the minimum pair in each Q_{A_I} and Q_{B_J}, one can then perform an $O(|A^{\#}| + |B^{\#}|)$-time linear scan to obtain two sets of 2D minima $Q_A = MIN(Q_{A_0} \cup Q_{A_1} \cup \cdots \cup Q_{A_{|A^{\#}|}})$ and $Q_B = MIN(Q_{B_0} \cup Q_{B_1} \cup \cdots \cup Q_{B_{|B^{\#}|}})$. Since both Q_A and Q_B are 2D minima, they are two sorted sets of points with respect to the x-coordinate. That is, it takes $O(|Q_A| + |Q_B|)$ time to merge Q_A and Q_B into a set Q_M so that Q_M is also sorted with respect to the x-coordinate. Therefore, $MIN(Q_M)$ can be obtained by an $O(|Q_M|)$-time linear scan on Q_M. This means $MIN(Q_M)$ can be determined in $O(|A^{\#}| + |B^{\#}|)$ time. Since we have $MIN(Q_M) = MIN(H'^{\#}_{l-1,k-1} \cup H^{\#}_{l,k-1}) = H^{\#}_{l,k}$, we conclude that the lemma holds. □

Definition 5.2 reveals that for any specific k and $l_1 \neq l_2$, $H^{\#}_{l_1,k} \cap H^{\#}_{l_2,k} = \{\phi\}$. Therefore, $\sum_{l=0}^{L'} |H^{\#}_{l,k}|$ is bounded by both $O(L'(|A^{\#}| + |B^{\#}|))$ and $O(|A^{\#}|r + |B^{\#}|m)$. According to Lemma 5.3, it is not difficult to design an $O(min\{L'n(|A^{\#}| + |B^{\#}|), n(|A^{\#}|r + |B^{\#}|m)\})$-time algorithm for determining $H^{\#}_{L',n}$ (see Algorithm 1). With the concept of Theorem 4.1, one can easily verify the correctness of the following theorem, thus its proof is omitted.

Theorem 5.1. *$MLCS^{\#}(T, A, B)$ can be determined on-line respect to T with $O(min\{L'n(|A^{\#}|+|B^{\#}|), n(|A^{\#}|r+|B^{\#}|m)\})$ time and $O(min\{L'n(|A^{\#}|+|B^{\#}|), L'(|A^{\#}|r + |B^{\#}|m)\})$ space.*

Note that both $O(L'n(|A^{\#}| + |B^{\#}|))$ and $O(n(|A^{\#}|r + |B^{\#}|m))$ are necessary for measuring the time complexity. Taking $r = \log n$, $|A^{\#}| = m$ and $|B^{\#}| = 1$ for example, we have $O(L'n(|A^{\#}| + |B^{\#}|)) = O(L'nm)$ but $O(n(|A^{\#}|r + |B^{\#}|m)) = O(nm \log n)$. The required space of this example is only $O(L'm \log n)$, which is a substantial improvement for the case where $n > m$. For off-line applications, Hirschberg's idea [1] can still be applied to reduce the space complexity. Based on the proof of Theorem 4.2, the following result can be easily obtained.

Theorem 5.2. *$MLCS^{\#}(T, A, B)$ can be determined with $O(min\{L'n(|A^{\#}| + |B^{\#}|), n(|A^{\#}|r + |B^{\#}|m)\})$ time and $O(m + L'(|A^{\#}| + |B^{\#}|))$ space.*

For applications that both $|A^{\#}|$ and $|B^{\#}|$ are relatively small, our algorithm takes only $O(L'n)$ time. In contrast, previous algorithm with S-table [15] does not meet this bound. Consider two sequences $A = X_1, X_2, \cdots, X_{|A^{\#}|}$ and $B = Y_1, Y_2, \cdots, Y_{|B^{\#}|}$, where each $X_i \in A$ denotes a single block and each $Y_j \in B$ does, too. The construction of S-tables takes $\Omega(\sum_{i=1}^{|A^{\#}|} nL_i^A + \sum_{j=1}^{|B^{\#}|} nL_j^B)$ time and space, where L_i^A and L_j^B denote $|LCS(T, X_i)|$ and $|LCS(T, Y_j)|$, respectively. Obviously, we have $(\sum_{i=1}^{|A^{\#}|} L_i^A + \sum_{j=1}^{|B^{\#}|} L_j^B) \ge L'$. Therefore, the needed time for computing S-tables cannot be kept in $O(L'n)$. Theorem 5.2 also indicates that our algorithm is more space-efficient than S-tables, since we have $(|A^{\#}| + |B^{\#}|) < n$ in general. When the number of blocks is limited, our off-line algorithm takes only $O(m)$ space.

TABLE 1. Algorithms for the merged LCS and the block-merged LCS problem. Here, n, m and r denote the length of T, the longer and shorter lengths of A and B, respectively, L or L' denotes the length of the answer, and δ denotes the number of blocks in A and B.

Merged LCS			
Algorithm	Complexity	Fixed Alphabets	Large Alphabets
Huang's on-line [15]	Time	$O(nmr)$	$O(nmr)$
	Space	$O(nmr)$	$O(nmr)$
Our on-line	Time	$O(Lnr)$	$O(n\sqrt{mr})$
	Space	$O(Lmr)$	$O(m^{1.5}r)$
Huang's off-line [15]	Time	$O(nmr)$	$O(nmr)$
	Space	$O(mr)$	$O(mr)$
Our off-line	Time	$O(Lnr)$	$O(n\sqrt{mr})$
	Space	$O(m+Lr)$	$O(m+\sqrt{mr})$
Block-merged LCS			
Algorithm	Complexity	Fixed Alphabets	Large Alphabets
Huang's on-line [15]	Time	$O(nm\delta)$	$O(nm\delta)$
	Space	$O(nm\delta)$	$O(nm\delta)$
Our on-line	Time	$O(L'n\delta)$	$O(n\sqrt{m\delta})$
	Space	$O(L'm\delta)$	$O(m^{1.5}\delta)$
Huang's off-line [15]	Time	$O(nm+n\delta^2)$	$O(nm+n\delta^2)$
	Space	$O(nm+n\delta^2)$	$O(nm+n\delta^2)$
Our off-line	Time	$O(L'n\delta)$	$O(n\sqrt{m\delta})$
	Space	$O(m+L'\delta)$	$O(m+\sqrt{m\delta})$

5.3. **Analysis for Large Alphabets.** In this subsection, we give a brief analysis for the case of large alphabets. Let $|\Sigma_A| \leq m$ and $|\Sigma_B| \leq r$ denote the number of distinct symbols in A and B, respectively. Thus, $|\Sigma| = |\Sigma_A \cup \Sigma_B| \leq m + r$, where Σ denotes the overall alphabet. In this case, it takes $O(|\Sigma| \log |\Sigma| + |\Sigma_A|m + |\Sigma_B|r)$ time and $O(|\Sigma_A|m + |\Sigma_B|r)$ space to construct the lexical-ordered tables for storing $next_A$ and $next_B$. However, note that the constructing time and space can be reduced to $O(m \log m)$ and $O(m)$, respectively, by using $(|\Sigma_A| + |\Sigma_B|)$ arrays that separately store the indices of each symbol in A and B. Therefore, for each $T[k]$, it takes $O(\log m)$ time to locate the array of $T[k]$. Then, each query for $next_A(T[k], i)$ and $next_B(T[k], j)$ can be answered in $O(\log |T[k]|)$ time, where $|T[k]|$ denotes number of indices i' that $A[i'] = T[k]$ or $B[i'] = T[k]$. Note that $|T[k]|$ is expected as $O(\frac{m}{|\Sigma|})$, which is almost a constant for large alphabets. In addition, recall that for large alphabets, $O(L)$ is expected to be $O(\sqrt{m})$ [24]. Therefore, our algorithms achieve significant improvements for large alphabets.

6. **Conclusions.** Table 1 summarizes related results for the merged LCS problem and the block-merged LCS problem. In Table 1, we adopt the general case that $O(n) = O(m)$. However, we do not replace each m with n, because they have different meaning in the problem. In addition, for simplicity, we assume $O(|A^{\#}|) = O(|B^{\#}|) = O(\delta) \leq O(m)$. For large alphabets, we assume $m = c|\Sigma|$, for some constant c, by which we apply the expected case that $O(L') \leq O(L) = O(\sqrt{m})$ and $O(\frac{m}{|\Sigma|}) = O(1)$ [24].

In Table 1, most of our algorithms are more efficient than previous results. Here we briefly discuss the only exception: our off-line algorithm for the block-merged LCS problem with fixed alphabets. In this case, for $O(L'\delta) \geq O(m + \delta^2)$, our algorithm would be less efficient in time. However, one should note that our off-line algorithm is always more efficient in space. Therefore, it is suitable for one to design a hybrid algorithm that combines our results with Huang's.

For future study, here is an interesting problem. Recall that our on-line algorithms for solving $MLCS(T, A, B)$ and $MLCS^{\#}(T, A, B)$ require $O(min\{Lnr, Lmr\})$ space and

$O(min\{L'n(|A^{\#}|+|B^{\#}|), L'(|A^{\#}|r+|B^{\#}|m)\})$ space, respectively. The space complexities for the average case, however, have not yet been analyzed. A few simple tests even suggest that our analysis for the worst cases may not be tight. For example, suppose that $T = \mathrm{a}^n$, $A = \mathrm{a}^n$ and $B = \mathrm{a}^n$ are three identical sequences of length n, which are formed with a unique symbol 'a'. This is one of the worst cases for traditional on-line sparse dynamic programming to determine $LCS(T, A)$ and $LCS(T, B)$, because the number of matches is maximized. By the implementation of our on-line algorithm with $O(min\{Lnr, Lmr\})$-space in Section 4, nevertheless, this case requires merely $O(n^2)$ space, rather than the worst $O(n^3)$ space. Since the worst case cannot be derived by simply maximizing the number of matches, to propose a tighter space analysis for our on-line algorithms would be an interesting topic in the future.

Acknowledgment. This research work was partially supported by the National Science Council of Taiwan under contract NSC-96-2221-E-110-010. The authors gratefully acknowledge the helpful comments and suggestions of Steve Haga and the reviewers, which have improved the presentation.

REFERENCES

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

[2] C.-B. Yang and R. C. T. Lee, "Systolic algorithms for the longest common subsequence problem," *Journal of the Chinese Institute of Engineers*, vol. 10(6), pp. 691–699, 1987.

[3] C. S. Iliopoulos and M. S. Rahman, "New efficient algorithms for the lcs and constrained lcs problems," *Information Processing Letters*, vol. 106(1), pp. 13–18, 2008.

[4] B. S. Baker and R. Giancarlo, "Sparse dynamic programming for longest common subsequence from fragments," *Journal of Algorithms*, vol. 42, no. 2, pp. 231–254, 2002.

[5] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24(4), pp. 664–675, 1977.

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

[7] J. D. Ullman, A. V. Aho, and D. S. Hirschberg, "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM*, vol. 23(1), pp. 1–12, 1976.

[8] A. N. Arslan and Ömer Eğecioğlu, "Algorithms for the constrained longest common subsequence problems," *International Journal of Foundations of Computer Science*, vol. 16(6), pp. 1099–1109, 2005.

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

[10] C.-L. Lu and Y.-P. Huang, "A memory-efficient algorithm for multiple sequence alignment with constraints," *Bioinformatics*, vol. 21(1), pp. 20–30, 2005.

[11] Y.-T. Tsai, "The constrained common sequence problem," *Information Processing Letters*, vol. 88, pp. 173–176, 2003.

[12] Z. Gotthilf, D. Hermelin, and M. Lewenstein, "Constrained LCS: hardness and approximation," in *Combinatorial Pattern Matching, 19th Annual Symposium (CPM2008)*, (Pisa, Italy), pp. 255–262, 2008.

[13] K.-S. Huang, C.-B. Yang, K.-T. Tseng, Y.-H. Peng, and H.-Y. Ann, "Dynamic programming algorithms for the mosaic longest common subsequence problem," *Information Processing Letters*, vol. 102, pp. 99–103, 2007.

[14] G. A. Komatsoulis and M. S. Waterman, "Chimeric alignment by dynamic programming: algorithm and biological uses," in *RECOMB '97: Proceedings of the first annual international conference on computational molecular biology*, (New York, NY, USA), pp. 174–180, ACM Press, 1997.

[15] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng, "Efficient algorithms for finding interleaving relationship between sequences," *Information Processing Letters*, vol. 105(5), pp. 188–193, 2008.

[16] O. Jaillon, J.-M. Aury, F. Brunet, J.-L. Petit, N. Stange-Thomann, E. Mauceli, L. Bouneau, C. Fischer, C. Ozouf-Costaz, A. Bernot, S. Nicaud, D. Jaffe, S. Fisher, G. Lutfalla, C. Dossat, B. Segurens,

C. Dasilva, M. Salanoubat, M. Levy, N. Boudet, S. Castellano, V. Anthouard, C. Jubin, V. Castelli, M. Katinka, B. Vacherie, C. Biemont, Z. Skalli, L. Cattolico, J. Poulain, V. de Berardinis, C. Cruaud, S. Duprat, P. Brottier, J.-P. Coutanceau, J. Gouzy, G. Parra, G. Lardier, C. Chapple, K. J. McKernan, P. McEwan, S. Bosak, M. Kellis, J.-N. Volff, R. Guigo, M. C. Zody, J. Mesirov, K. Lindblad-Toh, B. Birren, C. Nusbaum, D. Kahn, M. Robinson-Rechavi, V. Laudet, V. Schachter, F. Quetier, W. Saurin, C. Scarpelli, P. Wincker, E. S. Lander, J. Weissenbach, and H. R. Crollius, "Genome duplication in the teleost fish *tetraodon nigroviridis* reveals the early vertebrate proto-karyotype," *Nature*, vol. 431, no. 7011, pp. 946–957, 2004.

[17] M. Kellis, B. W. Birren, and E. S. Lander, "Proof and evolutionary analysis of ancient genome duplication in the yeast *saccharomyces cerevisiae*," *Nature*, vol. 428, no. 6983, pp. 617–624, 2004.

[18] D. Vallenet, L. Labarre, Z. Rouy, V. Barbe, S. Bocs, S. Cruveiller, A. Lajus, G. Pascal, C. Scarpelli, and C. Medigue, "MaGe: a microbial genome annotation system supported by synteny results," *Nucleic Acids Research*, vol. 34, no. 1, pp. 53–65, 2006.

[19] K. Hokamp, A. McLysaght, and K. H. Wolfe, "The 2R hypothesis and the human genome sequence," *Journal of Structural and Functional Genomics*, vol. 3, no. 1-4, pp. 95–110, 2003.

[20] G. Panopoulou and A. J. Poustka, "Timing and mechanism of ancient vertebrate genome duplications - the adventure of a hypothesis," *TRENDS in Genetics*, vol. 21, no. 10, pp. 559–567, 2005.

[21] H. Zhu, H. Kai, K. Eguchi, and Z. Guo, "Application of BPNN in classification of time intervals for intelligent intrusion detection decision response system," *International Journal of Innovative Computing Information and Control*, vol. 4(10), pp. 2483–2491, 2008.

[22] T. Furusho, T. Nishi, and M. Konishi, "Distributed optimization method for simultaneous production scheduling and transportation routing in semiconductor fabrication bays," *International Journal of Innovative Computing Information and Control*, vol. 4(3), pp. 559–575, 2008.

[23] K. Najim, E. Ikonen, and E. Gómez-Ramírez, "Trajectory tracking controlbased on a genealogical decision tree controller for robot manipulators," *International Journal of Innovative Computing Information and Control*, vol. 4(1), pp. 53–62, 2008.

[24] M. Kiwi, M. Loebl, and J. Matoušek, "Expected length of the longest common subsequence for large alphabets," *Advances in Mathematics*, vol. 197, pp. 480–498, 2005.

[25] G. M. Landau and M. Ziv-Ukelson, "On the common substring alignment problem," *Journal of Algorithms*, vol. 41, no. 2, pp. 338–354, 2001.

[26] G. M. Landau, B. Schieber, and M. Ziv-Ukelson, "Sparse LCS common substring alignment," *Information Processing Letters*, vol. 88, no. 6, pp. 259–270, 2003.