Efficient Sparse Dynamic Programming for the Merged LCS Problem

Yung-Hsing Peng, Chang-Biau Yang†, Chiou-Ting Tseng and Chiou-Yi Hor
Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan 80424
†cbyang@cse.nsysu.edu.tw

Abstract *Detecting the interleaving relationship between sequences is a new research topic which begins to draw much attention. Recently, Huang* et al. *proposed the merged LCS model, which can be realized as finding the longest common subsequence between a target sequence and two merging sequences. Given the targe sequence T and two merging sequences A and B, Huang* et al. *gave an $O(nmr)$-time and $O(mr)$-space algorithm for solving this problem, where n, m and r denote the length of T, the longer and shorter length of A and B, respectively. In this paper, we propose an improved $O(Lnr)$-time and $O(m + Lr)$-space algorithm for solving the merged LCS problem, where L denotes the length of the optimal solution. Compared with Huang's results, our algorithms are more efficient for applications where L could be relatively small.*

Keywords: algorithm, dynamic programming, longest common subsequence, merged sequence

1 Introduction

Given a sequence S, a subsequence $\bar{S}$ of S is a sequence obtained from S by deleting zero or more characters. Given two sequences S_1 and S_2, the *longest common subsequence* (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. Related problems of finding the LCS of two or more sequences have been extensively studied for several decades [4, 9, 16]. In general, most algorithms for finding the LCS are based on either traditional dynamic programming [4] or sparse dynamic programming [2, 5, 8]. From the perspective of complexity, the traditional dynamic programming takes $O(|S_1||S_2|)$ time, which compares each character in S_1 with each character in S_2, where $|S_1|$ and $|S_2|$ denote the lengths of S_1 and S_2, respectively. For unbounded alphabets, this approach is optimal because it requires $\Omega(|S_1||S_2|)$ time to obtain the solution [15]. However, for fixed-sized alphabets, such as DNA or protein sequences, the sparse dynamic programming takes less time in determining the LCS [5, 8], provided that the information of matches can be obtained efficiently.

In addition to the traditional LCS problem, some extended versions with their algorithms, such as the constrained LCS problem [1, 3, 9, 13, 14] and the mosaic LCS problem [7, 10], have also begun to draw more and more attention in sequence analysis. Recently, inspired by shuffles in playing poker cards, Huang *et al.* [6] proposed the merged LCS problem and the blocked merged LCS problem, which can be used to detect the interleaving relationship among sequences. 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 $A = a_1a_2a_3a_4 =$ tgat and $B = b_1b_2b_3 =$ tgc, then three of the possible merging sequences include $M_1 = a_1a_2b_1a_3a_4b_2b_3 =$ tgtatgc, $M_2 = b_1a_1b_2a_2b_3a_3a_4 =$ ttggcat and $M_3 = a_1b_1b_2b_3a_2a_3a_4 =$ ttgcgat. Suppose we have $T =$ attcgag, by examining all merging sequences, it can be seen that one of the optimal solutions of $LCS(T, A \oplus B)$ is $LCS(T, M_3) =$ ttcga, whose length is 5. Huang *et al.* first gave an $O(nmr)$-time algorithm [6] for determining $MLCS(T, A, B)$, whose off-line implementation takes $O(mr)$ space, where n, m and r denote the length of T, the longer and shorter lengths of A and B, respectively. For the *blocked merged LCS* problem, in which A and B are merged with block constraint, they first proposed an on-line algorithm with $O(nm(|A^{\#}| + |B^{\#}|))$ time and space, where $|A^{\#}|$ and $|B^{\#}|$ denote the number of blocks in A and B, respectively. In addi-

tion, based on the technique of S-table [11, 12], they also proposed an off-line algorithm which takes only $O(nm + n|A^{\#}||B^{\#}|)$ time and space.

In this paper, we will focus on the merged LCS problem, instead of its variant with blocks. However, the concept of our algorithm can still be applied for the case of blocks. Here we assume the merging sequences A and B are stored in database, while the target sequence T could be an on-line input. That is, we have both $|A|$ and $|B|$ for designing our algorithms, while $|T|$ could be unknown beforehand. In general, this assumption is practical, especially for applications of on-line searching. For fixed-sized alphabets, we first propose an on-line algorithm for solving the merged LCS problem with $O(Lnr)$ time and $O(min\{Lnr, Lmr\})$ space, where L represents the length of the obtained LCS. In addition, we also give an off-line $O(m + Lr)$-space implementation of our algorithm, which is more practical for real applications.

The rest of this paper is organized as follows. First, in Section 2, we will describe a simple linear time strategy for finding two-dimensional minima on grids, which will be used to support our algorithms for the merged LCS problem in Section 3. Finally, in Section 4 we draw our conclusions along with some future works.

2 The Two-dimensional Minima of Grid Points

A minimum in a set of two-dimensional points is defined as follows.

Definition 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 confined to grid points with proper boundaries, the two-dimensional minima $MIN(Q)$ of Q can be determined efficiently. Let each point in Q be bounded by $Z_x, Z_y \in Z^+$, 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 1. *For any set Q of two-dimensional grid points bounded by $Z_x, Z_y \in Z^+$, one can determine $MIN(Q)$ 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 simply set $(i, y_i) = (\infty, \infty)$. Now, it it 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, based on sorting one can verify that $MIN(Q)$ can be determined in $O(|Q| \log |Q|)$ time. We do not discuss the detailed algorithm for real points, because Lemma 1 is sufficient to support our algorithm in Section 3.

3 Algorithms for the Merged LCS Problem

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 $1 \leq i \leq j \leq |S|$. Considering the boundary conditions in our algorithms, here we define $S[0]$ as an empty character and $S[i, j]$ as an empty string for any $i > j$. For clearly explaining our algorithm, we define the relationship between a pair of distinct 2-tuple numbers as follows.

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

To perform our sparse dynamic programming, we begin with the idea of identifying candidates. To simplify descriptions, 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 3. *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 called an (l, k)-candidate if the following situations hold. (1)$LCS(T[1, k], A[1, i] \oplus B[1, j]) = l$ (2)For any integer $(i', j') < (i, j)$, $LCS(T[1, k], A[1, i'] \oplus B[1, j']) < l$. Such an (l, k)-candidate is also called a dominating candidate.*

According to Definition 3, there may be more than one (L, k)-candidate (i, j) that $L = LCS(T[1, k], A[1, i] \oplus B[1, j]) = LCS(T, A \oplus B)$, for $0 \leq k \leq n$, $0 \leq i \leq m$ and $0 \leq j \leq r$. By extending this idea, we have the following lemma.

Lemma 2. *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: For $L = 0$, which corresponds to no common subsequence of T and $A \oplus B$, one can see that $C = (i_0, j_0) = (0, 0)$ and $K = k_0 = 0$. Therefore, the lemma holds for the case $L = 0$.

For $L > 0$, there exists an (L, k')-candidate (i, j) with minimum index k' such that $LCS(T[1, k'], A[1, i] \oplus B[1, j]) = L$. Therefore, we have $LCS(T[1, k' - 1], A[1, i] \oplus B[1, j]) = L - 1$. Based on the second condition in Definition 3, $LCS(T[1, k'], A[1, i - 1] \oplus B[1, j]) = L - 1$ and $LCS(T[1, k'], A[1, i] \oplus B[1, j - 1]) = L - 1$. This implies $LCS(T[1, k'], A[1, i-1] \oplus B[1, j-1]) \le L-1$. Hence, to make a common subsequence of length L, either $A[i]$ or $B[j]$ must be picked with $T[k']$ as the Lth common character. That is, either $T[k'] = A[i]$ or $T[k'] = B[j]$ holds.

If $T[k'] = A[i]$, then we have $(i_L, j_L) = (i, j)$ and $k_L = k'$. This indicates that the remaining sequence $(i_1, j_1), (i_2, j_2), \cdots, (i_{L-1}, j_{L-1})$ and indices $k_1, k_2, \cdots, k_{L-1}$ can be obtained by applying the same idea to $T[1, k'-1]$ and $A[1, i-1] \oplus B[1, j]$, whose LCS is of length $L-1$. Similarly, if $T[k'] = B[j]$, then $(i_1, j_1), (i_2, j_2), \cdots, (i_{L-1}, j_{L-1})$ and $k_1, k_2, \cdots, k_{L-1}$ can be obtained via sequences $T[1, k' - 1]$, $A[1, i]$ and $B[1, j - 1]$. Because $(i - 1, j) < (i, j)$ and $(i, j-1) < (i, j)$, we have $(i_{L-1}, j_{L-1}) < (i, j)$. Since $L - 1$ is bounded by zero, with the concept of recursion, one can see that this lemma holds. □

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

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 both Σ_A and Σ_B are of fixed size, with an $O(m + r)$-time preprocessing on A and B, it is clear that both $next_A(\sigma, i)$ and $next_B(\sigma, j)$ can be determined in constant time, for any $1 \le i \le m$, $1 \le j \le r$. Note that σ is not necessarily an element of Σ_A or Σ_B.

Let $H_{l,k}$ denote the set of (l, k)-candidates, for any $l \ge 1$ and $k \ge 1$. We have the following lemma.

Lemma 3. *For any dominating candidate* $(i, j) \in H_{l,k}$, *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, with the consideration of an arbitrary 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 $LCS(T[1, k], A[1, i'] \oplus B[1, j']) = l$, which is contradictory to that (i, j) is an (l, k)-candidate. Therefore, either $A[i] = T[k]$ or $B[j] = T[k]$ holds.

Suppose $A[i]$ is picked with $T[k]$ as the lth common character. In this case, based on the proof of Lemma 2, there exists at least one $(\bar{i} < i, \bar{j} \le j) \in H_{l-1,k-1}$ which satisfies $next_A(T[k], \bar{i}) = i$. That is, the first condition holds. Similarly, the second condition also holds by picking $B[j]$ with $T[k]$.

If $T[k]$ is not picked as the lth common character, $LCS(T[1, k], A[1, i] \oplus B[1, j]) = l$ would induce $LCS(T[1, k - 1], A[1, i] \oplus B[1, j]) = l$. In this case, we have that $(i, j) \in H_{l,k}$ and $(i, j) \in H_{l,k-1}$, which concludes the third condition. □

Let $H'_{l,k}$ be a set constructed by substituting each candidate (i, j) in $H_{l,k}$ with two grid 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 grid points bounded by $Z_x = m$ and $Z_y = r$. Based on Definitions 1, 3 and Lemma 3, 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 $O(min(m, r)) = O(r)$ time, which is a key point to our on-line algorithm.

Lemma 4. *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: By Definition 3, it is clear that $H_{0,k} = \{(0, 0)\}$, for $0 \le k \le n$. Furthermore, for $l > k$ we have $H_{l,k} = \phi$. In these two cases, $H_{l,k}$ can be determined with constant time, which is used as the boundary condition.

For the remaining case $1 \le l \le k \le n$, recall that $H_{l,k} = MIN(H'_{l-1,k-1} \cup H_{l,k-1})$. According to the second condition in Definition 3, 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

$|H'_{l-1,k-1}| \leq 2|H_{l-1,k-1}|$, it is clear that $|H'_{l-1,k-1} \cup H_{l,k-1}| \leq 3(r+1)$. With Lemma 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. Note that we also have $|H_{l,k}| \leq r+1$. Therefore, the lemma holds. □

Based on Lemma 4, $H_{L,n}$ can be obtained in $O(Lnr)$-time by a traditional dynamic programming on-line with respect to T, as shown in Algorithm 1. Moreover, based on Lemma 2, it is easy to design an $O(L+n)$-time backtracking algorithm, which reports the increasing sequence of dominating candidates and the resulting LCS. With the analysis for the required space, our first result is given in Theorem 1.

Algorithm 1 Determining $H_{L,n}$ 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
    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}.
```

Theorem 1. *There exists an algorithm on-line respect to T which determines the LCS of T and $A \oplus B$ with $O(Lnr)$ time and $O(min\{Lnr, Lmr\})$ space.*

Proof: Here we only give our analysis on space complexity, because the time complexity $O(Lnr)$ has been discussed. Given $(i,j) \in H_{l,k}$, let $Back_{l,k}(i,j)$ denote the query function for tracking back 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 $(i,j) \in H_{l,k-1}$. One can see that a straightforward implementation would totally update the tracking function $O(Lnr)$ times, which requires $O(Lnr)$ space in order to retrieve the LCS. However, some updates can be avoided. Actually, 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), the tracking function needs to be updated with $Back_{l,k}(i,j) = (i',j')$ only for the minimum k that $(i,j) \in H_{l,k}$ and $(i',j') \in H_{l-1,k-1}$, denoted as $MK(i,j,l)$. Since the number of (i,j,l)-triplet 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 minimum k with increasing l. More precisely, the lth element in $G_{i,j}$ stores the minimum k that $(i,j) \in H_{l,k}$. By doing so, one can see that the update of $Back_{l,k}(i,j) = (i',j')$ can be done by first appending to $G_{i,j}$ a new element which stores k, then adding 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. One can see that the last element in $G_{i',j'}$ stores the minimum k' that $(i',j') \in H_{l-1,k'}$. The correctness of this implementation can be verified with the observation that for any specific k and $l \neq l'$, we cannot have both $(i,j) \in H_{l,k}$ and $(i,j) \in H_{l',k}$. 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 can be written as $O(Lnr + mr)$ and $O(min\{Lnr+mr, Lmr\})$, respectively. However, for on-line applications it is suitable to assume $Ln > m$, since n still grows steadily while m is given. Therefore, by omitting the additional $O(mr)$ complexity, one can see the lemma holds. □

The concept of our backtracking algorithm is summarized in 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 straightforward implementation with $O(n+L)$ time. Besides, the space efficiency of Theorem 1 would be substantial for on-line applications when n is much greater than m.

If the on-line process on T is not required, based on Hirschberg's idea [4], a more space-efficient implementation with $O(m+Lr)$ space can be further derived, provided that the length of T is given beforehand.

Theorem 2. *There exists an $O(Lnr)$-time and $O(m+Lr)$-space algorithm which determines the LCS of T and $A \oplus B$.*

Proof: **Omitted for simplicity.** □

Note that the worse case of this off-line implementation takes no more than $O(mr)$ space, because

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
```

$L \leq 2m$. For some cases, however, the required space is much less than $O(mr)$. Take $L = r = \sqrt{m}$ for example, the space complexity is merely $O(m)$, rather than $O(mr) = O(m^{1.5})$. In addition, for the case that L is relatively small, the required space would be reduced to near $O(m+r)$. Therefore, Theorem 2 is very practical for real applications.

4 Concluding Remark

In this paper, with the idea of sparse dynamic programming, we propose an efficient algorithm for solving the merged LCS problem. For the case of fixed alphabet, we first propose an $O(Lnr)$-time and $O(min\{Lnr, Lmr\})$-space on-line algorithm, which improves Huang's result with $O(nmr)$-time and $O(nmr)$-space [6]. In addition, our $O(m+Lr)$-space off-line algorithm is also more efficient than their $O(mr)$-space off-line implementation [6]. Besides, the advantages of our algorithms can also be seen when they are applied to large-size alphabets, because the expected values of L would be smaller.

Recall that in Section 3, our on-line algorithms for solving $MLCS(T, A, B)$ takes $O(min\{Lnr, Lmr\})$ space. However, the required space for the average case have not yet been analyzed. Also, a few simple tests imply that our analysis may not be tight for the worst cases. 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 the traditional on-line sparse dynamic programming to determine $LCS(T, A)$ and $LCS(T, B)$, because the number of matches is maximized. Via the implementation of our on-line algorithm with $O(min\{Lnr, Lmr\})$-space in Section 3, nevertheless, one can see that 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 algorithms would be an interesting topic for future study.

Finally, the idea of our algorithm in Section 3 can, in fact, be extended to solve the variant problem with blocks [6]. From this extension we further derive two algorithms. The first one is an on-line algorithm which takes $O(min\{L'n(|A^{\#}| + |B^{\#}|), n(|A^{\#}|r + |B^{\#}|m)\})$ time and $O(min\{L'n(|A^{\#}| + |B^{\#}|), L'(|A^{\#}|r + |B^{\#}|m)\})$ space, where L' is the length of the optimal solution. The second one is an off-line algorithm which takes the same time but requires only $O(m + L'(|A^{\#}| + |B^{\#}|))$ space. However, for simplicity we omit their detailed descriptions and related proofs, which are left to the full version of this paper.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.