The Longest Common Subsequence Problem with Variable Gapped Constraints [*]

Yung-Hsing Peng[a] and Chang-Biau Yang[b†]

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

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

Abstract

The longest common subsequence (LCS) problem with gap constraints (or the gapped LCS), which has applications to genetics and molecular biology, is an interesting and useful variant to the LCS problem. In previous work, this problem can be solved in $O(nm)$ time when the gap constraints are fixed to a single integer, where n and m denote the lengths of the two input sequences, respectively. In this paper, we generalize the problem from fixed gaps to variable gap constraints, offering a new flexible approach for sequence analysis. By using an efficient technique for incremental suffix maximum queries, we show that this generalized problem can be solved in $O(nm)$ time, which improves the previous result.

1 Introduction

Algorithms for finding the *longest common subsequence* (LCS) [2, 5–7, 9, 13, 14] have been widely and extensively studied for a long time. Motivated by its applications to genetics and molecular biology, Iliopoulos and Rahman [11] introduced an interesting variant for finding the LCS, called the *fixed gap LCS* (FGLCS) problem, where a value k of the fixed gap constraint is given and the distance between two consecutive matches is required to be limited to at most $k+1$. The best-known algorithm for solving the FGLCS problem was also proposed by Iliopoulos and Rahman [10], which takes $O(nm)$ time, where n and m denote the lengths of the two input sequences.

[*]This research was partially supported by the “Applied Information Services Development & Integration project, phase II” of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China. This research was also partially supported by the National Science Council of Taiwan under contract NSC 98-2221-E-110-062.

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

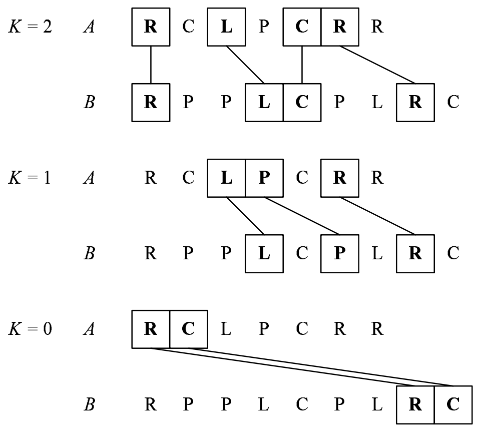

Figure 1: An example for illustrating the FGLCS between two sequences A=“RCLPCRR” and B=“RPPLCPLRC” with three different fixed gap constraints $k = 2$, $k = 1$, and $k = 0$.

An important application of the FGLCS is to detect motif patterns [11]. Taking two protein sequences A=“RCLPCRR” and B=“RPPLCPLRC” in Figure 1 for example, for three different fixed gap constraints $k = 2$, $k = 1$, and $k = 0$, the detected motifs of A and B are of the form “R..L..C..R”, “L.P.R”, and “RC”, respectively, where ‘.’ represents the wildcard symbol that can match any amino acids. Here the symbol ‘.’ may be an empty character.

For general motifs, however, the length of each segment containing ‘.’ may not be fixed. As a

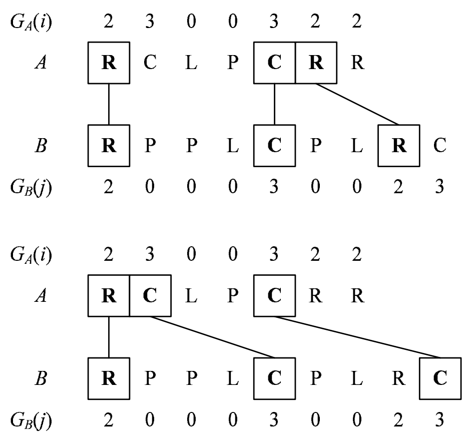

Figure 2: An example for illustrating the VGLCS between two sequences A=“RCLPCRR” and B=“RPPLCPLRC” with $G_A = [2, 3, 0, 0, 3, 2, 2]$ and $G_B = [2, 0, 0, 0, 3, 0, 0, 2, 3]$.

result, some motif sequences may not be revealed by the approach of FGLCS. In the above example, one can verify that the motifs “R...C..R” and “R...C...C” cannot be obtained by applying any fixed k. Note that “R...L...C...R”, rather than “R...C...C”, will be obtained if $k = 3$. To overcome such circumstances, in this paper we consider a more flexible variant, called the *variable gap LCS* (VGLCS) problem, in which the gap constraints are defined by two gap functions G_A and G_B, where $G_A(i)$ and $G_B(j)$ denote the two gap constraints (integers) applied to the ith character of A and the jth character of B, respectively. More precisely, when the ith character in A and the jth character in B are picked up as a common character, the distances between them and their previous common characters in A and B are limited to at most $G_A(i) + 1$ and $G_B(j) + 1$, respectively. Figure 2 presents an example showing that two motifs “R...C..R” and “R...C...C”, which cannot be obtained by FGLCS, can now be revealed by VGLCS. In this example, we set the gap constraints to 2, 3, 0, and 0 for amino acids “R”, “C”, “L”, and “P”, respectively. This example also indicates an important application of VGLCS: to detect motifs where different types of amino acids are of different interests [3, 12]. In this paper, we propose an efficient algorithm with $O(nm)$ time for finding VGLCS, offering a new flexible tool for analyzing sequences.

The rest of this paper is organized as follows. In Section 2, we formally define the VGLCS problem, and present a simple algorithm that takes $O(n^2m^2)$ time. Next, in Section 3 we improve the required time to $O(nm)$ by using a newly devised technique called the *incremental suffix maximum query* (ISMQ). Finally, in Section 4, we conclude our results.

2 Preliminaries

For a sequence (array) S, let $S[i]$ denote the ith character (element) in S, and $S[i, j]$ denote the substring (subarray) ranging from $S[i]$ to $S[j]$, for $1 \le i \le j \le |S|$, where $|S|$ denotes the length (size) of S. A sequence $S' = S[i_1], S[i_2], \cdots, S[i_p]$ is called a subsequence of length p in S, where $1 \le i_1 < i_2 < \cdots < i_p \le |S|$. For a two-dimensional $n \times m$ array X, let $X[i][j]$ denote the element in the ith row and the jth column of X. For specifying subarrays, let $X[i_1, i_2][j_1, j_2]$ denote the two-dimensional subarray (rectangle) of X, which is a collection of $X[i][j]$ with $1 \le i_1 \le i \le i_2 \le n$ and $1 \le j_1 \le j \le j_2 \le m$. In the following, we first define the VGLCS problem. After that, we present a simple $O(n^2m^2)$-time algorithm with for solving the problem.

Definition 1. Variable Gap Subsequence: *Given a sequence S and a gap function G_S, a subsequence $S' = S[i_1], S[i_2], \cdots, S[i_p]$ is called a variable gap subsequence of length p in S if $i_x - i_{x-1} \le G_S(i_x) + 1$, for $2 \le x \le p$, where $G_S(i_x)$ denotes the gap constraint on $S[i_x]$.*

Definition 2. The Variable Gap Common Subsequence (VGCS): *Given two sequences A and B with their gap functions G_A and G_B, a matching list of indices $(i_1, j_1), (i_2, j_2), \cdots, (i_p, j_p)$ forms a variable gap common subsequence between A and B, if (1)$A[i_x] = B[j_x]$, for $1 \le x \le p$, and (2)$1 \le i_1 < i_2 < \cdots < i_p \le |A|$ and $1 \le j_1 < j_2 < \cdots < j_p \le |B|$, and (3)$i_x - i_{x-1} \le G_A(i_x) + 1$ and $j_x - j_{x-1} \le G_B(j_x) + 1$, for $2 \le x \le p$.*

Given two sequences A and B with their gap functions G_A and G_B, the *variable gap LCS* (VGLCS) problem is to find the longest matching list that forms a variable gap common subsequence between A and B. Clearly, by setting $G_A(i) = k$ and $G_B(j) = k$, for $2 \le i \le |A|$ and $2 \le j \le |B|$, one can solve the FGLCS problem by the VGLCS algorithm.

Now we present a simple algorithm for solving the VGLCS problem, which is adapted from a previous algorithm for FGLCS [11]. Let V be an $n \times m$ array, where $V[i][j]$ denotes the length of the longest matching list that forms a variable gap common subsequence between $A[1,i]$ and $B[1,j]$. Also, let F be an $n \times m$ array, where $F[i][j]$ denotes the maximal length of the VGCS between $A[1,i]$ and $B[1,j]$ whose last common character is $A[i]$ and $B[j]$. Let $M_G(i,j)$ be the set of matching indices (i',j') satisfying that $1 \le i' \le i-1$, $1 \le j' \le j-1$, $A[i'] = B[j']$, $i - i' \le G_A(i)+1$, and $j - j' \le G_B(j)+1$. A simple recursive formula for computing $V[i][j]$ and $F[i][j]$ is given as follows, where cases 1, 2, and 3 refer to the case $i=0$ or $j=0$, the case $A[i] \neq B[j]$, and the case $A[i] = B[j]$, respectively.

$$F[i][j] = \begin{cases} 0 & \text{(case 1)} \\ 0 & \text{(case 2)} \\ max\{_{(i',j') \in M_G(i,j)} F[i'][j']\} + 1 & \text{(case 3).} \end{cases}$$

$$V[i][j] = \begin{cases} 0 & \text{(case 1)} \\ max\{V[i-1][j], V[i][j-1]\} & \text{(case 2)} \\ max\{F[i][j], V[i-1][j], V[i][j-1]\} & \text{(case 3).} \end{cases}$$

The correctness of the above formula can be easily verified by Definition 2. However, note that a straightforward implementation takes $O(n^2m^2)$ time in the worst case, since the size of $M_G(i,j)$ could be $O(nm)$. In the next section, we propose an efficient algorithm that determines $V[n][m]$ in $O(nm)$ time.

3 An Improved Algorithm for the VGLCS Problem

In this section, we first relate the above VGLCS algorithm to the *incremental suffix maximum query* (ISMQ). Then, we propose an efficient technique for solving ISMQ, with which our algorithm for solving the VGLCS problem reduces the required time to $O(nm)$.

3.1 Finding VGLCS with Incremental Suffix Maximum Queries

Given a string (array) D of numbers, a suffix maximum query $SMQ_D(i)$ asks for the maximum number in the suffix $D[i,|D|]$ of D. When D is given beforehand, one can reversely scan D, storing all answers for $SMQ_D(i)$ in $O(|D|)$ time, for $1 \le i \le |D|$. After that, each $SMQ_D(i)$ can be determined in $O(1)$ time by a simple table lookup. However, if D is given incrementally, this one-time scanning is not applicable for determining $SMQ_D(i)$, because each $SMQ_D(i)$ can vary as D grows. Taking $D = [10,3,7,2]$ for example, we have $SMQ_D = [10,7,7,2]$, where the ith number in SMQ_D represents the answer for $SMQ_D(i)$. Suppose that D grows to $[10,3,7,2,5]$ and $[10,3,7,2,5,8]$, then we have $SMQ_D = [10,7,7,5,5]$ and $SMQ_D = [10,8,8,8,8,8]$, respectively. Since we assume D to be incremental, we thereby name these queries as *incremental suffix maximum queries* (ISMQ). For ease of understanding, we leave our technique for ISMQ to the next section, but first propose a new algorithm that finds VGLCS by using ISMQ.

Recall that in the VGLCS problem, we are given two sequences A and B, and their gap functions G_A and G_B. For keeping the information of the VGLCS, we use two $n \times m$ arrays V and F defined in Section 2. To store the required maxima, we construct two $n \times m$ arrays Col and All, where $Col[i][j]$ and $All[i][j]$ denote the maximum element in the one-dimensional subarrays $F[i-1-G_A(i), i-1][j,j]$ and $Col[i,i][j-1-G_B(j), j-1]$, respectively. With this arrangement, one can see that $All[i][j]$ stores the maximum element of the two-dimensional $(G_A(i)+1) \times (G_B(j)+1)$ rectangle $F[i-1-G_A(i), i-1][j-1-G_B(j), j-1]$. With the above variables, our algorithm for finding the VGLCS is proposed in Algorithm 1. For completeness, let $V[i][j]=0$, $F[i][j]=0$, $Col[i][j]=0$, and $All[i][j]=0$ if $i \le 0$ or $j \le 0$.

Lemma 1. *Algorithm 1 solves the VGLCS problem in $O(\alpha nm)$ time, provided that each $Col[i][j]$ and $All[i][j]$ can be determined in $O(\alpha)$ time.*

Proof: To verify the correctness of this algorithm, we discuss both conditions $A[i] = B[j]$ and $A[i] \neq B[j]$. For $A[i] = B[j]$, one can see that $F[i][j]$ refers to $All[i][j]$, which stores the maximum length of VGCS that ends with some $A[i']$ and $B[j']$ satisfying that $A[i'] = B[j']$, $i-1-G_A(i) \le i' \le i-1$, and $j-1-G_B(j) \le j' \le j-1$. Note that $A[i]$ and $B[j]$ need not form a common character, we have $V[i][j] = max\{F[i][j], V[i-1][j], V[i][j-1]\}$. For $A[i] \neq B[j]$, it is clear that $F[i][j]=0$, and that $V[i][j]$ should be determined by $max\{V[i-1][j], V[i][j-1]\}$. Therefore, Algorithm 1 is an implementation for the recursive formula in Section 2. Suppose that each $Col[i][j]$ and $All[i][j]$ in the loop can be computed in $O(\alpha)$ time, then $V[n][m]$, $F[n][m]$, $Col[n][m]$, and $All[n][m]$ can all be determined in $O(\alpha nm)$ time. Finally, one can easily design an algorithm with $O(n+m)$ time for tracing the VGLCS. Hence, the lemma

Algorithm 1 Algorithm for Finding VGLCS

```
for $i = 1$ to $n$ do
  for $j = 1$ to $m$ do
    $Col[i][j] \leftarrow max\{F[i-1-G_A(i)][j], F[i-G_A(i)][j], \cdots, F[i-1][j]\}$.
    $All[i][j] \leftarrow max\{Col[i][j-1-G_B(j)], Col[i][j-G_B(j)], \cdots, Col[i][j-1]\}$.
    if $A[i] = B[j]$ then
      $F[i][j] \leftarrow All[i][j] + 1$.
      $V[i][j] \leftarrow max\{F[i][j], V[i-1][j], V[i][j-1]\}$ .
    else
      $F[i][j] \leftarrow 0$.
      $V[i][j] \leftarrow max\{V[i-1][j], V[i][j-1]\}$.
    end if
  end for
end for
Retrieve the VGLCS by tracing $V[n][m]$.
```

holds. □

Note that in Algorithm 1, each $Col[i][j]$ and $All[i][j]$ can be determined by ISMQ, because each column of F and each row of Col can be deemed as incremental strings of numbers. To handle ISMQ, a typical approach is to use a balanced binary search tree or a van Emde Boas tree [10, 16], which reduces α to $\log n$ or $\log\log n$, respectively. In the following, we propose a more efficient technique for ISMQ, reducing the factor $O(\alpha)$ to $O(1)$.

3.2 Handling ISMQ by Union and Find

The *disjoint set union* problem (or the *union-find* problem) [4, 15] is a well-known issue that has many applications in algorithm design. A data structure for solving the union-find problem is therefore called a union-find data structure [4, 15]. In a union-find data structure, there are three operations available:

***make*(x, C)**: Create a new singleton set {x} whose name is C. This operation is forbidden if x is already in some existing set.
***find*(x)**: Retrieve the name of the unique set containing x.
***unite*(x, y, C)**: Unite the two different sets containing x and y into one new set named C.

Interestingly, we notice that the ISMQ problem can in fact be reduced to the union-find problem. With Algorithm 2, we show how to accomplish ISMQ by a union-find data structure. Here we treat the indices of the number string in ISMQ as elements, and treat each number in the string as a name of some set. In this way, we can ensure that each element x is unique. Suppose that D is an incremental string of numbers that can grow by adding some number d_i, where i denotes the order (index) of the added number. In addition, let $W = w_1, w_2, \cdots, w_{|W|}$ be a list of increasing indices, where each d_{w_j} denotes a suffix maximum and it represents the name of one existing set, for $1 \le j \le |W|$. For simplicity, we always use $w_{|W|}$ to refer to the last index in W, even though some indices may be appended to or removed from W. Also, for ease of understanding, we again take $D = [10, 3, 7, 2, 5, 8]$ as an example, showing the process of Algorithm 2 with Figure 3 from left to right. In Figure 3, the numbers in grey circles represent the indices stored in W, and nodes surrounded by the same bold oval are the elements in the same set, whose name is denoted by the bold number beside the bold oval.

Lemma 2. *Algorithm 2 reports all $SMQ_D(j)$ in $O(\beta(|D| + |Q|))$ time, where β, $|D|$, and $|Q|$ denote the required time of one union-find operation, the length of the input string, and the number of ISMQs, respectively.*

Proof: Based on the *make* and the *unite* operations applied in Algorithm 2, one can verify that for any indices $i > j$ that $d_i \ge d_j$, i and j belong to the same set C such that $C \ge d_i$. That is, the set containing j is always named by the maximum number among $d_j, d_{j+1}, \cdots, d_{|D|}$. Therefore, one can use $find(j)$ to determine $SMQ_D(j)$, which proves the correctness of Algorithm 2. Note that the number of operations caused by *make* and *unite* is bounded by $O(|D|)$. In addition, the number of operations caused by $find$ is exactly $|Q|$. Finally, the total operations on W take $O(|D|)$ time. As a result, Algorithm 2 takes

Algorithm 2 Answering ISMQ with Union-find Operations

```
Create an empty union-find data structure, and an empty $W$ with $|W| = 0$.
$i \leftarrow 1$.
while there exists input $d_i$ do
  $make(i, d_i)$.
  while $|W| > 0$ and $d_i \geq d_{w_{|W|}}$ do
    //If $d_i \geq d_{w_{|W|}}$, then $d_i$ is the suffix maximum for each element in $d_{w_{|W|}}$.
    $unite(w_{|W|}, i, d_i)$.
    Delete $w_{|W|}$ from $W$.
  end while
  Append $i$ to $W$.
  while there exists a query $SMQ_D(j)$ do
    Report $SMQ_D(j) = find(j)$.
  end while
  $i \leftarrow i + 1$.
end while
```

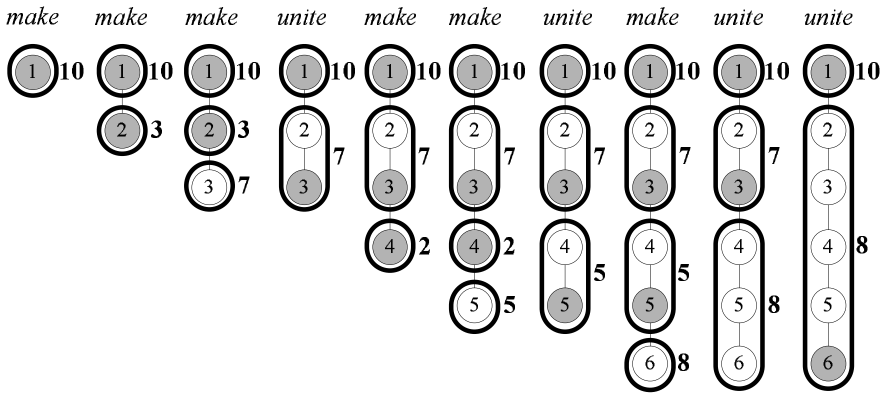

Figure 3: The process of Algorithm 2 with $D = [10, 3, 7, 2, 5, 8]$, where the numbers in grey circles represent the indices stored in W, and nodes surrounded by the same bold oval belong to the same set, whose name is denoted by the bold number beside the bold oval.

I S B N：978-986-02-7580-3

$O(\beta(|D| + |Q|) + |D|) = O(\beta(|D| + |Q|))$ time, where β denotes the required time to perform one union-find operation. □

For the general case of the union-find problem, the lower bound of β was proved to be a functional inverse of Ackermann's function [15], which unfortunately cannot be eliminated. However, noticing that the union-find operations involved in Algorithm 2 correspond to the *incremental tree set union* [4], the factor β can be reduced to $O(1)$. In Algorithm 2, each newly added element can be deemed as a single node of a *union tree* T that contains only one single path (please refer to Figure 3). Besides, the sets created by *make* and *unite* always correspond to consecutive disjoint paths in T. In Figure 3, one can see that each *make* inserts one single node into T. In addition, each *unite* can be achieved by uniting two nodes v and $p(v)$ in T, where $p(v)$ is the parent of v. Note that v is always the highest node of a set, thus it takes merely $O(1)$ time to locate such v for any given set. Therefore, the union-find operations in Algorithm 2 can be implemented with the incremental tree set union [4], which reduces β to $O(1)$.

For finding the VGLCS, one can verify that Algorithm 1 invokes no more than $O(nm)$ *make*s and *unite*s. Meanwhile, the number of invoked ISMQs in Algorithm 1, which equals to the number of $finds$, is also bounded by $O(nm)$. That is, we have reduced $O(\alpha)$ to $O(1)$, obtaining an algorithm with $O(nm)$ time for finding the VGLCS.

Theorem 1. *The VGLCS problem for two given sequences A and B can be solved in $O(nm)$ time, where n and m denote the lengths of A and B, respectively.*

4 Conclusion

In this paper, we propose an efficient algorithm with $O(nm)$ time for solving the VGLCS problem, which is achieved by our optimal approach for handling the incremental suffix maximum query (ISMQ). To our knowledge, our extension to the gap constraint is the first-known investigation on variable gaps, which offers a more flexible tool for sequence analysis. Besides, we notice that our technique for ISMQ can be applied to improve recent results on the block edit problems proposed by Ann *et al.*[1]. Though the detailed definitions for their problems are omitted here, we point out that by applying our technique for ISMQ, the required time for solving their second problem [1] can be improved from $O(nm \log m + m^2)$ to $O(nm + m^2)$, where n and m denote the lengths of the two input strings, respectively. Therefore, the proposed technique in this paper is beneficial to both sequence analysis and stringology. In the future, we would like to apply our techniques to sequence alignment [8], devising new flexible tools for aligning sequences.

References

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

[2] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings," *Information Processing Letters*, Vol. 108, pp. 360–364, 2008.

[3] Y. Elbaz, T. Salomon, and S. Schuldiner, "Identification of a glycine motif required for packing in EmrE, a multidrug transporter from *escherichia coli*," *Journal of Biological Chemistry*, Vol. 283(18), pp. 12276–12283, 2008.

[4] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Journal of Computer and System Sciences*, Vol. 30(2), pp. 209–221, 1985.

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

[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] X. Huang and K.-M. Chao, "A generalized global alignment algorithm," *Bioinformatics*, Vol. 19(2), pp. 228–233, 2003.

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

[10] C. S. Iliopoulos, M. Kubica, M. S. Rahman, and T. Walen, "Algorithms for computing the longest parameterized common subsequence," *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, London, Ontario, Canada, pp. 265–273, 2007.

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

[12] E. Lidome, C. Graf, M. Jaritz, A. Schanzer, P. Rovina, R. Nikolay, and F. Bornancin, "A conserved cysteine motif essential for ceramide kinase function," *Biochimie*, Vol. 90(10), pp. 1560–1565, 2008.

[13] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor, "Efficient sparse dynamic programming for the merged LCS problem with block constraints," *International Journal of Innovative Computing, Information and Control*, Vol. 6, pp. 1935–1947, 2010.

[14] Y.-H. Peng, C.-B. Yang, K.-S. Huang, and K.-T. Tseng, "An algorithm and applications to sequence alignment with weighted constraints," *International Journal of Foundations of Computer Science*, Vol. 21, pp. 51–59, 2010.

[15] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, Vol. 22(2), pp. 215–225, 1975.

[16] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Information Processing Letters*, Vol. 6, No. 3, pp. 80–82, 1977.