Information and Computation 237 (2014) 95–100

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Finding the gapped longest common subsequence by incremental suffix maximum queries ☆

Yung-Hsing Peng [a], 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

ARTICLE INFO

Article history:
Received 20 April 2011
Received in revised form 7 December 2012
Available online 13 June 2014

Keywords:
algorithm
longest common subsequence
incremental suffix maximum query

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 is 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 A and B, respectively. In this paper, we first generalize the problem from fixed gaps to variable gap constraints. Then, we devise an optimal approach for the incremental suffix maximum query (ISMQ), which helps us obtain an efficient algorithm with $O(nm)$ time for finding LCS with variable gap constraints. In addition, our technique for ISMQ can be applied to solve one of the block edit problems on strings, reducing the time complexity from $O(nm\log m + m^2)$ to $O(nm + m^2)$. Hence, the result of this paper is beneficial to related research on sequence analysis and stringology.

1. Introduction

Algorithms for finding the *longest common subsequence* (LCS) [1–7] have been widely and extensively studied for a long time. Motivated by its applications to genetics and molecular biology, Iliopoulos and Rahman [8] 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 [9], which takes $O(nm)$ time, where n and m denote the lengths of the two input sequences A and B, respectively.

An important application of the FGLCS is to detect motif patterns in bio-sequences [8]. Taking two protein sequences $A =$ “RCLPCRR” and $B =$ “RPPLCPLRC” in Fig. 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 acid. Here the symbol ‘.’ may be an empty character.

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

☆ This research was partially supported by the “Online and Offline Integrated Smart Commerce Platform (1/4)” 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 address: cbyang@cse.nsysu.edu.tw (C.-B. Yang).

http://dx.doi.org/10.1016/j.ic.2014.06.001
0890-5401/© 2014 Elsevier Inc. All rights reserved.

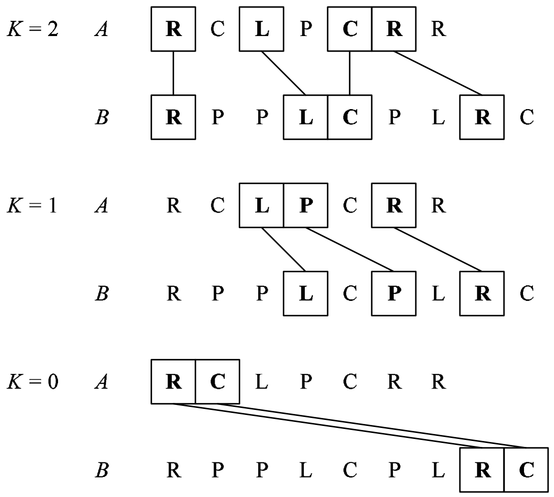

Fig. 1. An example for illustrating the FGLCS between two sequences $A =$ "RCLPCRR" and $B =$ "RPPLCPLRC" with three different fixed gap constraints $k = 2$, $k = 1$, and $k = 0$.

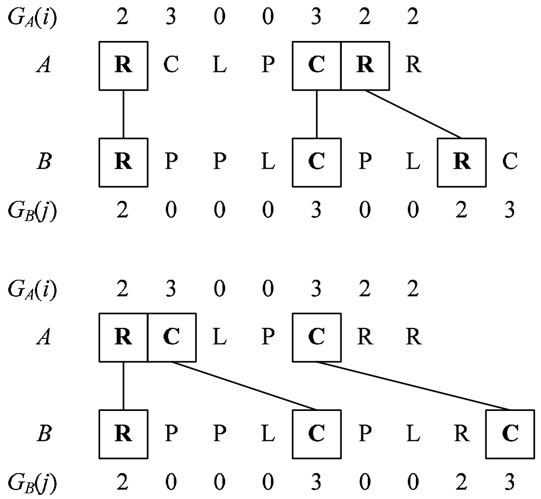

Fig. 2. An example for illustrating the VGLCS between two sequences $A =$ "RCLPCRR" and $B =$ "RPPLCPLRC" with $G_A = [2, 3, 0, 0, 3, 2, 2]$ and $G_B = [2, 0, 0, 0, 3, 0, 0, 2, 3]$.

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 is picked in the common subsequence, then its previously picked character is bounded by the distance $G_A(i) + 1$ (likewise, for the jth character of B). Fig. 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 [10,11]. 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 the *incremental suffix maximum query* (ISMQ). In Section 4, we give two other applications of ISMQ. Finally, in Section 5, 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 $1 \le i_1 < i_2 < \cdots < i_p \le |S|$. Also, let G_S denote the gap function of S, and $G_S(i)$ denote the gap constraint (a non-negative integer) on $S[i]$. 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 $X[i][j]$ that $1 \le i_1 \le i \le i_2 \le n$ 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 algorithm with $O(n^2m^2)$ time for solving the problem.

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

Definition 2. *The variable gap common subsequence (VGCS)*: Given two sequences A and B with their gap functions G_A and G_B, a sequence Z is a VGCS of A and B if Z is both a VGS of A with G_A and a VGS of B with G_B.

Given two sequences A and B with their gap functions G_A and G_B, the *variable gap LCS* (VGLCS) problem asks one to find the VGCS of maximal length. Clearly, by setting $G_A(i) = k$ and $G_B(j) = k$, for $2 \le i \le n$ and $2 \le j \le m$, 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 [8]. Let V be an $n \times m$ array, where each $V[i][j]$ denotes the length of VGLCS between $A[1, i]$ and $B[1, j]$. Also, let F be an $n \times m$ array, where each $F[i][j]$ denotes the maximal length of VGCS between $A[1, i]$ and $B[1, j]$ when $A[1, i]$ and $B[1, j]$ are picked as the last common character. According to this denotation, we have $F[i][j] \le V[i][j]$. 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:

$$F[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 0 & \text{if } A[i] \ne B[j], \\ max\{_{(i',j') \in M_G(i,j)} F[i'][j']\} + 1 & \text{if } A[i] = B[j], \end{cases}$$

$$V[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ max\{V[i-1][j], V[i][j-1]\} & \text{if } A[i] \ne B[j], \\ max\{F[i][j], V[i-1][j], V[i][j-1]\} & \text{if } A[i] = B[j]. \end{cases}$$

The correctness of the above formula can be easily verified by discussing whether $A[i]$ and $B[j]$ are picked as the last character in the VGLCS. 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 $SMQ_D(i)$ reports 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 calculate the maximum value in the formula for $F[i][j]$, 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 $j \le 0$.

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 $V[n][m]$.
```

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 $A[i] \ne B[j]$. For $A[i] = B[j]$, one can see that $F[i][j]$ refers to $All[i][j]$, which stores the length of VGLCS 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, so 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 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 [9,12], 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) [13,14] is a well-known problem that has many applications in algorithm design. A data structure for solving the union-find problem is therefore called a union-find data structure [13,14]. In a union-find data structure, there are three operations available as follows:

make$(\boldsymbol{x}, \boldsymbol{C})$: Create a new singleton set {x} whose name is C. This operation is forbidden if x is already in some existing set.

find$(\boldsymbol{x})$: Retrieve the name of the unique set containing x.

unite$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{C})$: 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 $D = d_1, d_2, \cdots, d_{i-1}$ is an incremental string of numbers that can be further incremented 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 an increasing list of dominating indices of D, where $d_{w_j} = max\{d_{w_{j-1}+1}, \cdots, d_{w_j - 1}, d_{w_j}\}$. In other words, d_{w_j} denotes a suffix maximum (incremental maximum scanned backward) and it represents a unique name for one existing set, for $1 \leq j \leq |W|$. For example, if $D = [10, 3, 7, 2, 5]$, then $W = [1, 3, 5]$. 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. For ease of understanding, we again take $D = [10, 3, 7, 2, 5, 8]$ as an example, showing the process of Algorithm 2 with Fig. 3 from left to right. In Fig. 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.

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 $d_i$ becomes a new suffix maximum that causes updates.
        $unite(w_{|W|}, i, d_i)$.
        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
```

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 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 $O(\beta(|D| + |Q|) + |D|) = O(\beta(|D| + |Q|))$ 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 [13], which unfortunately cannot be eliminated. However, noticing that the union-find operations involved

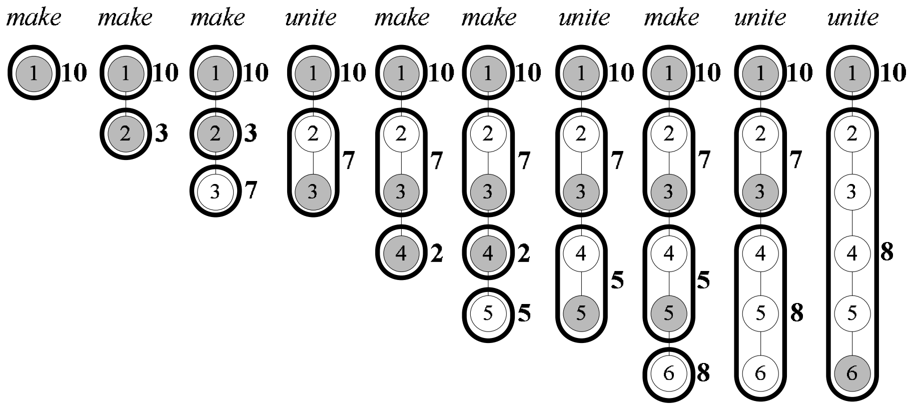

Fig. 3. The process of Algorithm 2 with $D = [10, 3, 7, 2, 5, 8]$, where the numbers in grey circles represent the indices stored in W, and nodes surrounded by the same bold oval belong to the same set, whose name is denoted by the bold number beside the bold oval.

in Algorithm 2 correspond to the *incremental tree set union* [14], 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 Fig. 3). Besides, the sets created by *make* and *unite* always correspond to consecutive disjoint paths in T. In Fig. 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 [14], 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 *find*s, 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.*

Note that for the FGLCS problem, the invoked ISMQs are special queries having the same suffix length. For this case, one can simplify the union-find structure to a list, which turns out to be the previous result of Iliopoulos and Rahman [9].

4. Other applications of ISMQ

In this section, we present two other applications of ISMQ, showing its contribution to sequence analysis and stringology.

4.1. Extension for elastic gaps

The first related application of ISMQ is to extend the case of *elastic gaps* [8]. In the original elastic gap LCS problem [8], two values k_1 and k_2 are given and the distance between two consecutive matches is now required to be at least $k_1 + 1$ and at most $k_2 + 1$. Similar to the VGLCS problem, we can extend k_1 and k_2 to involve variable gaps.

Definition 3. *The variable elastic gap subsequence (VEGS)*: Given a sequence S and its pair of elastic gap functions E_{1S} and E_{2S}, a subsequence $S' = S[i_1], S[i_2], \cdots, S[i_p]$ is called a VEGS of length p in S if $E_{1S}(i_x) + 1 \le i_x - i_{x-1} \le E_{2S}(i_x) + 1$, for $2 \le x \le p$, where $E_{1S}(i_x)$ and $E_{2S}(i_x)$ denote the lower and the upper bounds of the distance constraint on $S[i_x]$.

Definition 4. *The variable elastic gap common subsequence (VEGCS)*: Given two sequences A and B with their elastic gap functions E_{1A}, E_{2A}, E_{1B}, and E_{2B}, a sequence Z is a VEGCS of A and B if Z is both a VEGS of A with E_{1A} and E_{2A}, and a VEGS of B with E_{1B} and E_{2B}.

Here we redefine $F[i][j]$ as the maximal length of the VEGCS between $A[1,i]$ and $B[1,j]$ whose last common character is $A[i]$ and $B[j]$, and $M_G(i,j)$ as the set of matching indices (i',j') satisfying that $1 \le i' \le i-1$, $1 \le j' \le j-1$, $A[i'] = B[j']$, $E_{1A}(i) + 1 \le i - i' \le E_{2A}(i) + 1$, and $E_{1B}(j) + 1 \le j - j' \le E_{2B}(j) + 1$. With this slight modification, the formula in Section 2 can then be applied to finding the longest VEGCS between A and B. Note that the elastic gap functions of A and B are given beforehand, which means we can generate all required ISMQs with an $O(nm)$-time preprocessing. Therefore, by proper arrangement of these ISMQs, one can easily derive an algorithm with $O(nm)$ time to obtain the longest VEGCS, which extends Iliopoulos and Rahman's results for finding LCS with elastic gaps [9].

4.2. The gapped all-suffix copy

Interestingly, we notice that our technique for ISMQ can also be applied to improving the result on the block-edit problem $P(EI, L)$ [15]. $P(EI, L)$ is a block-edit problem which allows both external copies and internal copies, and the cost of a block-copy operation is linear to the copied length. For the detailed explanation of $P(EI, L)$, one can refer to Ann et al.'s article [15]. Here we focus on a related little problem. For clarity, we name this problem as the *gapped all-suffix copy problem*, which is an important part in Ann et al.'s algorithm for solving $P(EI, L)$. Given a string S of length $|S|$ with its gap function G_S and its splitting cost function H_S, the gapped all-suffix copy problem on A is to compute each minimal copy cost $R_S(i) = min\{H_S(j) + h \times (i - j) \mid i - G_S(i) \le j \le i\}$ for $1 \le i \le |S|$, where $G_S(i)$ is a non-negative integer (gap constraint) that limits the distance between i and j, $H_S(j)$ denotes the cost to split the string $S[1, i]$ with index j, and h is a constant factor (not necessarily an integer). For solving this problem, Ann et al. first compute the function $H'_S(i) = H_S(i) - h \times (i - 1)$, and then locate the required index j for $R_S(i)$ by computing $R'_S(i) = min\{H'_S(j) | i - G_S(i) \le j \le i\}$. To compute $R'_S(i)$ for non-integer h, Ann et al. adopt a balanced binary search tree, resulting in an $O(|S| \log |S|)$-time algorithm. Obviously, with our technique of ISMQ, the required time can be reduced to $O(|S|)$. We point out that by applying our technique for ISMQ, the required time for solving $P(EI, L)$ [15] 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.

5. Conclusion

In this paper, we propose an optimal approach for handling the incremental suffix maximum query (ISMQ), which can be adopted to derive an efficient algorithm with $O(nm)$ time for solving the VGLCS problem. In addition to the VGLCS problem, the definition of elastic gaps [8] can also be extended from two fixed integers to four gap functions. 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 solving the block edit problem with gap constraints on suffix copy [15], reducing the time complexity from $O(nm \log m + m^2)$ to $O(nm + m^2)$. Therefore, our result is also beneficial to stringology.

References

[1] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (1975) 341–343.
[2] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (5) (1977) 350–353.
[3] K.-S. Huang, C.-B. Yang, K.-T. Tseng, Y.-H. Peng, H.-Y. Ann, Dynamic programming algorithms for the mosaic longest common subsequence problem, Inf. Process. Lett. 102 (2007) 99–103.
[4] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, Y.-H. Peng, Efficient algorithms for finding interleaving relationship between sequences, Inf. Process. Lett. 105 (5) (2008) 188–193.
[5] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, C.-Y. Hor, A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings, Inf. Process. Lett. 108 (2008) 360–364.
[6] Y.-H. Peng, C.-B. Yang, K.-S. Huang, K.-T. Tseng, An algorithm and applications to sequence alignment with weighted constraints, Int. J. Found. Comput. Sci. 21 (2010) 51–59.
[7] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, C.-Y. Hor, Efficient sparse dynamic programming for the merged LCS problem with block constraints, Int. J. Innov. Comput. Inf. Control 6 (2010) 1935–1947.
[8] C.S. Iliopoulos, M.S. Rahman, Algorithms for computing variants of the longest common subsequence problem, Theor. Comput. Sci. 395 (2008) 255–267.
[9] C.S. Iliopoulos, M. Kubica, M.S. Rahman, T. Walen, Algorithms for computing the longest parameterized common subsequence, in: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007), London, Ontario, Canada, 2007, pp. 265–273.
[10] Y. Elbaz, T. Salomon, S. Schuldiner, Identification of a glycine motif required for packing in EmrE, a multidrug transporter from escherichia coli, J. Biol. Chem. 283 (18) (2008) 12276–12283.
[11] E. Lidome, C. Graf, M. Jaritz, A. Schanzer, P. Rovina, R. Nikolay, F. Bornancin, A conserved cysteine motif essential for ceramide kinase function, Biochimie 90 (10) (2008) 1560–1565.
[12] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (3) (1977) 80–82.
[13] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (2) (1975) 215–225.
[14] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. Syst. Sci. 30 (2) (1985) 209–221.
[15] H.-Y. Ann, C.-B. Yang, Y.-H. Peng, B.-C. Liaw, Efficient algorithms for the block edit problems, Inf. Comput. 208 (3) (2010) 221–229.