Available online at www.sciencedirect.com

ScienceDirect

Theoretical Computer Science 395 (2008) 255–267

Theoretical Computer Science

www.elsevier.com/locate/tcs

Algorithms for computing variants of the longest common subsequence problem[☆]

Costas S. Iliopoulos[*], M. Sohel Rahman[1]

Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, England, United Kingdom

Abstract

The *longest common subsequence* (LCS) problem is one of the classical and well-studied problems in computer science. The computation of the LCS is a frequent task in DNA sequence analysis, and has applications to genetics and molecular biology. In this paper we introduce new variants of LCS problem and present efficient algorithms to solve them. In particular we introduce the notion of gap constraints in the LCS problems. For the LCS problem with fixed gap, we first present a naive algorithm runs in $O(n^2 + \mathcal{R}(K+1)^2)$ time, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match and K is the fixed gap constraint. We then improve the running time to $O(n^2 + \mathcal{R}K + \mathcal{R}\log\log n)$ using some novel techniques. Furthermore, we present an algorithm that is independent of K and runs in $O(n^2 + \mathcal{R}\log\log n)$ time. Using these techniques, we also present a new $O(n^2)$ algorithm to solve the original LCS problem. Additionally, we modify our algorithms to handle elastic and rigid gaps. We also apply the notion of rigidness to the original LCS problem and modify the traditional dynamic programming solution to handle the rigidness presenting a $O(n^2)$ algorithm to solve the problem. Finally, we also improve the solution to Rigid Fixed Gap LCS to $O(n^2)$. Notably, in all of the above cases, we assume that the two given strings are of equal length i.e. n. But our results can be easily extended to handle two strings of different length.

Keywords: Algorithm; Longest common subsequence; Strings

1. Introduction

The *longest common subsequence* (LCS) problem is one of the classical and well-studied problems in computer science which has extensive applications in diverse areas ranging from spelling error corrections to molecular biology: a spelling error correction program tries to find the dictionary entry which resembles most a given word; in a file archive we want to store several versions of a source program compactly by storing only the original version and construct all other versions from the differences to the previous one; in Molecular Biology [30,3] we want to compare

[☆] Preliminary version [M. Sohel Rahman, Costas S. Iliopoulos, Algorithms for computing variants of the longest common subsequence problem, in: T. Asano (Ed.), ISAAC, in: Lecture Notes in Computer Science, vol. 4288, Springer, 2006, pp. 399–408] was presented in ISAAC 2006.

[*] Corresponding author.
E-mail addresses: csi@dcs.kcl.ac.uk (C.S. Iliopoulos), sohel@dcs.kcl.ac.uk (M. Sohel Rahman).
URLs: http://www.dcs.kcl.ac.uk/staff/csi (C.S. Iliopoulos), http://www.dcs.kcl.ac.uk/pg/sohel (M. Sohel Rahman).

[1] On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.

0304-3975/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.01.009

DNA or protein sequences to learn how similar they are. All these cases can be seen as an investigation for the *'closeness'* among strings. And an obvious measure for the closeness of strings is to find the maximum number of identical symbols in them preserving the order of the symbols. This, by definition, is the longest common subsequence of the strings.

Perhaps, the strongest motivation for the LCS problem and variants thereof, to this date, comes from Computational Molecular Biology. The LCS problem is a common task in DNA sequence analysis, and has applications to genetics and molecular biology. Variants of LCS problem have been used to study similarity in RNA secondary structure [10,20,13,23]. Very recently, Bereg and Zhu in [8], presented a new model for RNA multiple sequence structural alignment based on the longest common subsequence. The Constrained LCS problem, where the solution to the problem must also be a subsequence of a third string (given), also finds motivation from bioinformatics [33,6,5]. We, however, are interested in applying another type of constraint on the LCS problem. We introduce the notion of gap constraints in LCS. Our versions of LCS, on one hand, offers the possibility to handle gap constraints between the consecutive matches among the sequences. On the other hand, they provide us with the tool to handle motif finding problems where not all positions of the motif are important [24]. For example, by examining nine different protein sequences that bind to the SH3 domain, a consensus sequence, $RPLP**P$, was revealed as the SH3 domain binding motif [2]. The symbol $*$ in the sequence is a wildcard that can match any amino acids, which indicates that the amino acids at the two $*$-positions are not important to the biochemical function of the motif. Such a consensus sequence can be obtained by applying gap constraints between the appropriate positions.

Another motivation of our work comes from the problem of extracting long multiple repeats in DNA sequences. One approach to solve this problem efficiently is to apply 'lossless' filters [1,32] where filters apply a necessary condition that sequences must meet to be part of repeats. One way to improve the computational time and the sensitivity of the filters is to compute Longest Common Subsequences between the ordered sequences of exact k-mers used in the filtering technique. However in the case of the filter, the LCS that needs to be computed has bounded span [31] which, again, can be obtained by applying the gap constraints in LCS.

The rest of the paper is organized as follows. In Section 2, we present all the definitions and notations to introduce the new concepts of gap constraints in LCS as well as the new variants that we wish to handle in this paper. In Section 3, we present a brief literature review. In Section 4, we review the traditional dynamic programming technique to solve the original LCS problem. In Sections 5 to 9, we present new algorithms for all the variants discussed in this paper. In particular, we first present a naive algorithm for the problem with fixed gap, in Section 5, that runs in $O(n^2+\mathcal{R}(K+1)^2)$ time and then improve the running time to $O(n^2+\mathcal{R}K+\mathcal{R}\log\log n)$ in Section 6. In Section 7, we present another algorithm for the same problem which runs in $O(n^2+\mathcal{R}\log\log n)$ time and hence independent of the parameter K. Using these techniques, we also present a new $O(n^2+\mathcal{R}\log\log n)$ time algorithm to solve the original LCS problem in this section. In Sections 8 and 9, we modify these algorithms to handle, respectively, the elastic gaps and rigid gaps. Also in Section 8, we modify the traditional solution to LCS to handle the notion of rigidness in the context of the original LCS problem. Note that this particular variant of the problem has been previously studied in [24]. Finally, we discuss some applications in Section 10 before concluding in Section 11.

2. Preliminaries

Suppose that we are given two strings $X[1..n] = X[1]\ X[2]\ \ldots X[n]$ and $Y[1..n] = Y[1]\ Y[2]\ \ldots Y[n]$. A subsequence $S[1..r] = S[1]\ S[2]\ \ldots S[r]$ of X is obtained by deleting $[0, n-r]$ symbols from X. A common subsequence of two strings X and Y, denoted $cs(X,Y)$, is a subsequence common to both X and Y. The longest common subsequence of X and Y, denoted $lcs(X,Y)$ or $LCS(X,Y)$, is a common subsequence of maximum length. We denote the length of $lcs(X,Y)$ by $r(X,Y)$.

Problem 1 (*"LCS"*). Given 2 strings X and Y, we want to find out the Longest Common Subsequence of X and Y.

In this paper we are interested in several variants of LCS Problem. In the rest of this section we formally define the new variants and give some examples. In what follows we assume that the two given strings are of equal length. But our results can be easily extended to handle two strings of different length.

Definition 1 (*"Correspondence Sequence"*). Given a string $X[1..n]$ and a subsequence $S[1..r]$ of X, we define the correspondence sequence of X and S, $C(X,S) = C[1]\ C[2]\ \ldots C[r]$ to be the strictly increasing sequence of integers

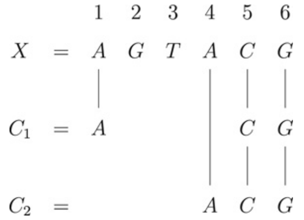

Fig. 1. Correspondence sequences for Example 1.

taken from $[1, n]$ such that $S[i] = X[C[i]]$ for all $1 \leq i \leq r$. When it is clear from the context we simply use C instead of $C(X, S)$.

Note that, given X and one of its subsequences S, the $C(X, S)$ may not be unique.

Example 1. Suppose that $X = AGTACG$. $S = ACG$ is a subsequence of X. As is evident from Fig. 1, here we have two different Correspondence Sequences of X and S namely, $C_1 = 1\ 5\ 6$ and $C_2 = 4\ 5\ 6$.

Definition 2 (*"Fixed Gapped Correspondence Sequence"*). A correspondence sequence of a string X of length n and one of its subsequences S of length r is said to be a Fixed Gapped Correspondence Sequence with respect to a given integer K if and only if we have $C[i] - C[i-1] \leq K + 1$ for all $2 \leq i \leq r$. We sometimes use $C_{FG(K)}$ to denote a Fixed Gapped Correspondence Sequence with respect to K.

Definition 3 (*"Elastic Gapped Correspondence Sequence"*). A correspondence sequence of a string X of length n and one of its subsequences S of length r is said to be an Elastic Gapped Correspondence Sequence with respect to a given integer K_1 and K_2, $K_2 > K_1$ if and only if we have $K_1 < C[i] - C[i-1] \leq K_2 + 1$ for all $2 \leq i \leq r$. We sometimes use $C_{EG(K_1,K_2)}$ to denote an Elastic Gapped Correspondence Sequence with respect to K_1 and K_2.

Example 2. Suppose that $K = 2$. Then in Example 1, $C_2(X, S)$ is a Fixed Gapped Correspondence Sequence but $C_1(X, S)$ is not because $C_1[2] - C_1[1] = 4 \not\leq K$. However, for $K = 5$ both C_1 and C_2 are Fixed Gapped Correspondence Sequences.

Definition 4 (*"Fixed Gapped Common Subsequence and Elastic Gapped Common Subsequence"*). Suppose that we are given two strings $X[1..n]$ and $Y[1..n]$ and an integer K. A common subsequence $S[1..r]$ of X and Y is a Fixed Gapped Common Subsequence, if there exist Fixed Gapped Correspondence Sequences $C_{FG(K)}(X, S)$ and $C_{FG(K)}(Y, S)$. Elastic Gapped Common Subsequences can be defined analogously.

Definition 5 (*"Rigid Fixed Gapped Common Subsequence and Rigid Elastic Gapped Common Subsequence"*). Suppose that we are given two strings $X[1..n]$ and $Y[1..n]$ and an integer K. A common subsequence $S[1..r]$ of X and Y is a Rigid Fixed Gapped Common Subsequence, if there exists Fixed Gapped Correspondence Sequences $C_{FG(K)}(X, S)$ and $C_{FG(K)}(Y, S)$ such that for all $2 \leq i \leq r$

$$C_{FG(K)}(X, S)[i] - C_{FG(K)}(X, S)[i-1] = C_{FG(K)}(Y, S)[i] - C_{FG(K)}(Y, S)[i-1].$$

Rigid Elastic Gapped Common Subsequences can be defined analogously.

Problem 2. *"FIG" (LCS Problem with Fixed Gap).* Given 2 strings X and Y, and an integer K, we want to find out the Fixed Gapped Common Subsequence of X and Y having maximum length.

Problem 3. *"ELAG" (LCS Problem with Elastic Gap).* Given 2 strings X and Y, and integers K_1 and K_2, we want to find out the Elastic Gapped Common Subsequence of X and Y having maximum length.

Problem 4. *"RIFIG" (LCS Problem with Rigid Fixed Gap).* Given 2 strings X and Y, and an integer K, we want to find out the Rigid Fixed Gapped Common Subsequence of X and Y having maximum length.

Problem 5. *"RELAG" (LCS Problem with Rigid Elastic Gap).* Given 2 strings X and Y, and integers K_1 and K_2, we want to find out the Rigid Elastic Gapped Common Subsequence of X and Y having maximum length.

Example 3. Suppose that $X = ABCCDEFGACD$ and $Y = AFCGFCABD$. A solution to LCS problem would be the subsequence $S_1 = ACFAD$ which is of length 5. Similarly, $S_2 = ACGAD$ is another solution to LCS problem. However, if we consider FIG with $K = 1$, none of the above sequences can be a solution as explained below. For S_1 we have two $C(X, S_1)$, namely $C_1(X, S_1) = 1\ 3\ 7\ 9\ 11$ and $C_2(X, S_1) = 1\ 4\ 7\ 9\ 11$. In both the cases the gap constraint ($K = 1$) is violated because, for example, $C_1(X, S_1)[3] - C_1(X, S_1)[2] = 4 > K + 1$. In fact it turns out that the length of the solution to FIG is only 3 for $K = 1$, a solution being $S_3 = FAD$. Another solution to FIG with $K = 1$ is $S_4 = FGC$. Interestingly enough, assuming $K = 2$, S_1 is a solution to FIG but S_2 is not. This is because in both $C_1(X, S_2) = 1\ 3\ 8\ 9\ 11$ and $C_2(X, S_2) = 1\ 4\ 8\ 9\ 11$ the gap constraint is violated as follows: $C_1(X, S_2)[3] - C_1(X, S_2)[2] = 5 > K + 1$ and $C_2(X, S_2)[3] - C_2(X, S_2)[2] = 4 > K + 1$.
Let us now consider ELAG for $K_1 = 1$ and $K_2 = 3$. In this case it is easy to see that S_1 is a solution. However, S_2 is not a solution because we have a violation of the gap constraint due to K_1 in $C(Y, S_2) = 1\ 3\ 4\ 7\ 9$ as follows: $C(Y, S_2)[3] - C(Y, S_2)[2] = 1 \not> K_1$.
Considering RIFIG, with $K = 1$, it is easy to see that S_3 is a solution. S_4 however is not a solution because it does not preserve the rigidness as shown below. We have $C(X, S_4) = 7\ 8\ 10$ and $C(Y, S_4) = 2\ 4\ 6$. As we can see $C(X, S_4)[2] - C(X, S_4)[1] = 1 \neq C(Y, S_4)[2] - C(X, S_4)[1] = 2$, which violates the required rigidness.

In this paper we use the following notions. We say that a pair (i, j), $1 \le i, j \le n$ defines a match, if $X[i] = Y[j]$. The set of all matches, $\mathcal{M}$, is defined as follows:

$$\mathcal{M} = \{(i, j) | X[i] = Y[j], 1 \le i, j \le n\}.$$

We define $|\mathcal{M}| = \mathcal{R}$.

3. Literature review

The longest common subsequence problem for k strings ($k > 2$) was first shown to be NP-hard [25] and later proved to be hard to be approximated [19]. The restricted but probably the more studied problem that deals with two strings has been studied extensively [28,35,29,27,26,18,17,16]. The classic dynamic programming solution to LCS problem, invented by Wagner and Fischer [35], has $O(n^2)$ worst case running time. Masek and Paterson [26] improved this algorithm using the "Four-Russians" technique [4] to reduce the worst case running time to $O(n^2/\log n)$.[2] Since then not much improvement in terms of n can be found in the literature. However, several algorithms exist with complexities depending on other parameters. For example Myers in [27] and Nakatsu et al. in [29] presented an $O(nD)$ algorithm where the parameter D is the simple Levenshtein distance between the two given strings [22]. Another interesting and perhaps more relevant parameter for this problem is $\mathcal{R}$. Hunt and Szymanski [18] presented an algorithm running in $O((\mathcal{R} + n) \log n)$. They have also cited applications where $\mathcal{R} \sim n$ and thereby claimed that for these applications the algorithm would run in $O(n \log n)$ time. For a comprehensive comparison of the well-known algorithms for LCS problem and study of their behaviour in various application environments the readers are referred to [9].

4. LCS algorithms

We start with a brief review of the traditional dynamic programming technique employed to solve LCS [35]. Here the idea is to determine the longest common subsequences for all possible prefix combinations of the input string. The recurrence relation for extending the length of LCS for each prefix pair $(X[1..i], Y[1..j])$, i.e. $r(X[1..i], Y[1..j])$, is as follows [35]:

$$\mathcal{T}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \mathcal{T}[i-1, j-1] + 1 & \text{if } X[i] = Y[j], \\ \max(\mathcal{T}[i-1, j], \mathcal{T}[i, j-1]) & \text{if } X[i] \neq Y[j]. \end{cases} \tag{1}$$

[2] Employing different techniques, the same worst case bound was achieved in [12]. In particular, for most texts, the achieved time complexity in [12] is $O(hn^2/\log n)$, where $h \le 1$ is the entropy of the text.

Here we have used the tabular notion $\mathcal{T}[i, j]$ (also denoted by $\mathcal{T}[i][j]$) to denote $r(X[1..i], Y[1..j])$. After the table has been filled, $r(X, Y)$ can be found in $\mathcal{T}[n, n]$ and $lcs(X, Y)$ can be found by backtracking from $\mathcal{T}[n, n]$ (for detail please refer to [35] or any textbook on algorithms, e.g. [11]).

5. An algorithm for FIG

In this section we first present a naive algorithm for FIG. In subsequent sections, we show how to improve this algorithm with some non-trivial modifications. Note that, in FIG, due to the gap constraint, a continuing common sequence may have to stop at an arbitrary $\mathcal{T}[i, j]$ because the next match is not within the gap constraint.

Example 4. Consider X and Y as defined in Example 3. Now consider the match $X[7] = Y[2] = F$. Considering LCS problem, this match would give us a subsequence, $S = AF$, of length 2 (note that we already have $X[1] = Y[1] = A$) common to both $X[1..7]$ and $Y[1..2]$. So Eq. (1) would give us $\mathcal{T}[2, 7] = 2$. But now consider FIG with $K = 4$. In this case we cannot continue the subsequence between $X[1..7]$ and $Y[1..2]$ after the first match, i.e. $X[1] = Y[1] = A$, because $C(X[1..7], S)[2] - C(X[1..7], S)[1] = 6 > K + 1$, which would violate the gap constraint. So, with $K = 4$, we would have a start of a new common subsequence at $\mathcal{T}[2, 7]$ as opposed to getting a continuing subsequence reaching length 2.

In order to cope with this situation what we do is as follows. For each tabular entry $\mathcal{T}[i, j]$, $(i, j) \in \mathcal{M}$ we calculate and store two values namely $\mathcal{T}_{local}[i, j]$, $\mathcal{T}_{global}[i, j]$. For all other (i, j), $\mathcal{T}_{local}[i, j]$ is irrelevant and, hence, is undefined. The recurrence relations are defined below:

$$\mathcal{T}_{local}[i, j] = \begin{cases} \mathit{Undefined} & \text{if } (i, j) \notin \mathcal{M}, \\ \max_{\substack{i-1-K \le \ell_i < i \\ j-1-K \le \ell_j < j \\ (\ell_i, \ell_j) \in \mathcal{M}}} (\mathcal{T}_{local}[\ell_i, \ell_j]) + 1 & \text{if } (i, j) \in \mathcal{M}. \end{cases} \tag{2}$$

Remark 1. The max operation in Eq. (2) returns 0, when there is no $(\ell_i, \ell_j) \in \mathcal{M}, i - 1 - K \le \ell_i < i, j - 1 - K \le \ell_j < j$.

$$\mathcal{T}_{global}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1]) & \text{if } (i, j) \notin \mathcal{M}, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1], \mathcal{T}_{local}[i, j]) & \text{if } (i, j) \in \mathcal{M}. \end{cases} \tag{3}$$

It is easy to see that $\mathcal{T}_{global}[i, j]$ is used to store the information of the LCS so far, i.e. the *'global'* LCS, on other hand $\mathcal{T}_{local}[i, j]$ tracks any *'local'* LCS in growth. As soon as a local LCS becomes the global one the value of the corresponding $\mathcal{T}_{global}$ changes. What will be the running time of this algorithm? Since, for each $(i, j) \in \mathcal{M}$, we have to check a $(K + 1)^2$ area to find the maximum of $\mathcal{T}_{local}$ in that area, the total running time is $O(n^2 + \mathcal{R}(K + 1)^2)$. The space requirement is $\theta(n^2)$. Also note that by keeping appropriate pointer information or by examining the calculation of the two recurrences we can easily construct $lcs(X, Y)$ as can be done in the case of the traditional solution of LCS problem.

Theorem 1. *Problem FIG can be solved in $O(n^2 + \mathcal{R}(K + 1)^2)$ time using $\theta(n^2)$ space.*

Remark 2. Unfortunately, this strategy, if used for the LCS problem, would lead to an inefficient algorithm with $O(n^2 + \sum_{(i,j)\in\mathcal{M}}(i - 1)(j - 1))$ worst case running time.

As in the case of the traditional solution of LCS problem the complete tabular information is required for our algorithm, basically, to provide the solution for the subproblems if required. If that is not required, we can get rid of $\mathcal{T}_{global}$ altogether by keeping a variable to keep track of the current global LCS. As soon as a local LCS becomes a global one we just change this variable. In this case, instead of considering each (i, j), $1 \le i \le n$, $1 \le j \le n$, we only need to consider each $(i, j) \in \mathcal{M}$. It is easy to see that this would give us a running time of $O(\mathcal{R}(K + 1)^2)$ provided we have a pre-processing step to construct the set $\mathcal{M}$ in sorted order according to their position they would be considered in the algorithm. This pre-processing step is as follows. We construct for each symbol $a \in \Sigma$ two separate lists, $L_X[a]$ and $L_Y[a]$. For each $a \in \Sigma$, $L_X[a]$ ($L_Y[a]$) stores, in sorted order, the positions of a in X (Y), if any. We now use an elegant data structure invented by van Emde Boas [34] that allows us to maintain a sorted list of integers in the range $[1..n]$ in $O(\log\log n)$ time per insertion and deletion. In addition to that it can return $next(i)$ (successor element of i in the list) and $prev(i)$ (predecessor element of i in the list) in constant time. In the rest of this paper, we refer to this data structure as vEB data structure. We construct a vEB data structure E_P where we insert each pair $(i, j), i \in L_X[a], j \in L_Y[a], a \in \Sigma$. In this case the order of the elements in E_P is maintained according to the value $(i*(n-1)+j)$. Note that these values are within the range $[1..n^2]$ and hence the cost is $O(\log\log n^2) = O(\log\log n)$ per insertion and deletion. The pre-processing steps are formally stated in Algorithm 1. It is easy to verify that, using E_P, we can get all the pairs in the correct order to process them in row-by-row manner. We now analyze the running time of this pre-processing step. The $2*|\Sigma|$ lists can be constructed in $O(n)$ by simply scanning X and Y in turn. Since there are in total $\mathcal{R}$ elements in $\mathcal{M}$, the construction of E_P requires $O(R\log\log n)$ time. The space requirement is $O(\mathcal{R})$. We note, however, that for the complete algorithm, we still need the $\theta(n^2)$ space to guaranty a linear time search for the highest $\mathcal{T}_{local}$ in the $(K+1)^2$ area.

Theorem 2. *Given a pre-processing time of $O(\mathcal{R}\log\log n)$, Problem FIG can be solved in $O(\mathcal{R}(K+1)^2)$ time.*

Algorithm 1 Pre-processing Step to get $\mathcal{M}$ in the prescribed order

```
for $a \in \Sigma$ do
    Insert the positions of $a$ in $X$ in $L_X[a]$ in sorted order
    Insert the positions of $a$ in $Y$ in $L_Y[a]$ in sorted order
end for
$E_P = \epsilon$ {initializing a vEB data structure}
for $a \in \Sigma$ do
    for $i \in L_X[a]$ do
        for $j \in L_Y[a]$ do
            insert $(i, j)$ in $E_P$ according to $(i*(n-1)+j)$.
        end for
    end for
end for
return $E$
```

6. An improved algorithm for FIG

In this section we try to improve the running time of the algorithm presented in Section 5. Ideally, we would like to reduce the quadratic term $(K+1)^2$ to linear. Note that we can easily improve the running time of the algorithm to the LCS problem reported in Remark 2 using the following interesting facts.

Fact 6. *Suppose that $(i, j) \in \mathcal{M}$. Then for all $(i', j), i' > i$ $((i, j'), j' > j)$, we must have $\mathcal{T}[i', j] \geq \mathcal{T}[i, j]$ $(\mathcal{T}[i, j'] \geq \mathcal{T}[i, j])$, where $\mathcal{T}$ is the table filled up by the traditional dynamic programming algorithm using Eq. (1).*

Fact 7. *The calculation of a $\mathcal{T}[i, j], (i, j) \in \mathcal{M}, 1 \leq i, j \leq n$ is independent of any $\mathcal{T}[\ell, q], (\ell, q) \in \mathcal{M}, \ell = i, 1 \leq q \leq n$.*

The idea is to avoid checking the $(i-1)(j-1)$ entries and check only $(j-1)$ (or $(i-1)$) entries instead. We maintain an array H of length n where, for $\mathcal{T}[i, j]$ we have, $H[\ell] = \max_{1<k<i,(i,\ell)\in\mathcal{M}}(\mathcal{T}[k, \ell]), 1 \leq \ell \leq n$. The 'max' operation, here, returns 0 if there exists no $(i, \ell) \in \mathcal{M}$ within the range. Given the updated array H, we can easily perform the task by checking only the $(j-1)$ entries of H. And Fact 6 makes it easy to maintain the array H on the fly as we proceed as follows. As usual, we proceed in a row-by-row manner. We use another array S, of length n, as a temporary storage. When we find an $(i, j) \in \mathcal{M}$, after calculating $\mathcal{T}[i, j]$ we store $S[j] = \mathcal{T}[i, j]$. We continue to store in this way as long as we are in the same row. As soon as we find an $(i', j) \in \mathcal{M}, i' > i$, i.e. we start processing a new row, we update H with new values from S.

The correctness of the above procedure[3] follows from Facts 6 and 7. But this idea does not work for FIG because Fact 6 does not hold when we consider FIG. This is because due to the gap constraint a new local LCS may start which would surely have lesser $\mathcal{T}$-value than another previous local LCS. We, however, use the similar idea to improve our previous running time. But we need to do something more than just maintaining an array. In the rest of this section, we present a novel technique to present the improved algorithm. The basic idea depends on the following fact which is, basically, an extension of Fact 7.

Fact 8. *The calculation of a $\mathcal{T}_{local}[i, j]$, $(i, j) \in \mathcal{M}$, $1 \leq i, j \leq n$ is independent of any $\mathcal{T}_{local}[\ell, q]$, $(\ell, q) \in \mathcal{M}$, $(\ell = i$ or $\ell < i - K - 1)$, $1 \leq q \leq n$.*

We maintain n vEB data structures E_i, $1 \leq i \leq n$, one for each column. We also need to maintain one *insert list*, $\mathcal{I}$ and one *delete list*, $\mathcal{D}$. Recall that we proceed in a row-by-row manner. Suppose that we are starting to process row $i + K + 2$ i.e. we are considering the *'first'* match in this row, namely, $(i + K + 2, j) \in \mathcal{M}$. So we need to calculate $\mathcal{T}_{local}[i + K + 2, j]$ and $\mathcal{T}_{global}[i + K + 2, j]$. At this instant the delete list $\mathcal{D}$ contains all (i, ℓ) such that $1 \leq \ell \leq n$, $(i, \ell) \in \mathcal{M}$ and the insert list $\mathcal{I}$ contains all $(i + K + 1, \ell)$ such that $1 \leq \ell \leq n$, $(i + K + 1, \ell) \in \mathcal{M}$. In other words, when we consider the first match in row $(i + K + 2)$, $\mathcal{D}$ contains all the matches in row i and $\mathcal{I}$ contains all the matches in row $i + K + 1$. For each $(p, \ell) \in \mathcal{D}$ we delete (p, ℓ) from E_ℓ and for each $(p, \ell) \in \mathcal{I}$ we insert (p, ℓ) in E_ℓ. Note that the sorted order in E_ℓ is maintained according to the value $\mathcal{T}_{local}[p, \ell]$. We then calculate $\mathcal{T}_{local}[i + K + 2, j]$ for all $1 \leq j \leq n$ such that $(i + K + 2, j) \in \mathcal{M}$. It should be clear that we can calculate $\mathcal{T}_{local}[i + K + 2, j]$ as follows in $O(K)$ time:

$$\mathcal{T}_{local}[i + K + 1, j] = \max_{j-K-1 \leq \ell \leq j-1} (value(\max(E_\ell))) + 1. \tag{4}$$

Note that $value(\max(E_\ell)) = \mathcal{T}_{local}[m, n]$ when $\max(E_\ell) = (m, n)$. The running time of $O(K)$ follows from the fact that we can find the maximum of each E_ℓ in constant time. And the correctness follows from Fact 8.

What should be the running time of this algorithm? It is clear that we spend $O(n^2 + \mathcal{R}K)$ time in computing the LCS for FIG. But this improved time comes at the cost of maintaining n vEB data structure E_i, $1 \leq i \leq n$. It is easy to verify that the total time to maintain E_i, $1 \leq i \leq n$ is $O(\mathcal{R} \log\log n)$ because we never insert nor delete more than $\mathcal{R}$ elements in total from/to E_i, $1 \leq i \leq n$. Note that the values to be inserted is always within the range [1..n] since no subsequence can be of length greater than n.

Theorem 3. *Problem FIG can be solved in $O(n^2 + \mathcal{R}K + \mathcal{R} \log\log n)$ time using $\theta(n^2)$ space.*

If the solutions for the subproblems are not required we need only compute $\mathcal{T}_{local}[i, j]$, $(i, j) \in \mathcal{M}$. We, however, would need to use a variable to finally report $r(X, Y)$ and use appropriate pointers to construct $lcs(X, Y)$. Therefore, we get the following theorem (The corresponding algorithm is formally stated in the form of Algorithm 2.)

Theorem 4. *Problem FIG can be solved in $O(\mathcal{R}K + \mathcal{R} \log\log n)$ time using $\theta(R)$ space.*

7. A K-independent algorithm for FIG

In this section we try to devise an algorithm for FIG that is independent of K. As we shall see later that this would give us an efficient algorithm for Problem LCS as well. We make use of a classical problem in computer science, namely, Range Maxima Query Problem.

Problem 9 (*"RMAX" (Range Maxima Query Problem)*). Suppose that we are given a sequence $A = a_1 a_2 \ldots a_n$. A Range Maxima (minima) Query specifies an interval $I = (i_s, i_e)$, $1 \leq i_s \leq i_e \leq n$ and the goal is to find the index ℓ with maximum (minimum) value a_ℓ for $\ell \in I$.

Theorem 5 (*[14,7]*). *The RMAX problem can be solved in $O(n)$ pre-processing time and $O(1)$ time per query.*

[3] Although we do not still achieve a good running time for Problem LCS in the worst case.

Algorithm 2

```
1: Construct the set $\mathcal{M}$ using Algorithm 1. Let $\mathcal{M}_i = (i, j) \in \mathcal{M}, 1 \leq j \leq n$.
2: for $i = j$ to $n$ do
3:   $E_j = \epsilon$ {Initialize the $n$ vEB structure one for each column}
4: end for
5: globalLCS.Instance $= \epsilon$
6: globalLCS.Value $= \epsilon$
7: for $i = 1$ $to$ $n$ do
8:   Insert $(i-1, j) \in \mathcal{M}_{i-1}$ $in$ $E_j, 1 \leq j \leq n$ {If $i - 1 \leq 0$ then insert nothing}
9:   Delete $(i-K-2, j) \in \mathcal{M}_{i-K-2}$ $in$ $E_j, 1 \leq j \leq n$ {If $i - K - 2 \leq 0$ then delete nothing}
10:  for each $(i, j) \in \mathcal{M}_i$ do
11:    $maxresult = \max_{(j-K-1) \leq \ell \leq (j-1)}(\max(E_\ell))$
12:    $\mathcal{T}.Value[i, j] = maxresult.Value + 1$
13:    $\mathcal{T}.Prev[i, j] = maxresult.Instance$
14:    if globalLCS.Value $<$ $\mathcal{T}$.Value$[i, j]$ then
15:      globalLCS.Value $= \mathcal{T}.Value[i, j]$
16:      globalLCS.Instance $= (i, j)$
17:    end if
18:  end for
19: end for
20: return globalLCS
```

With Theorem 5 in our hand we can modify Algorithm 2 as follows. We want to implement Step 11 in constant time so that we can avoid the dependency on K completely. Before we start processing a particular row, just after Step 9, we create an array of length n, $A = \max(E_j), 1 \leq j \leq n$. Now we simply replace the Step 11 with an *appropriate* Range Maxima Query. It is easy to see that, due to Fact 8, this will work correctly. Since we have a constant time implementation for Step 11, we now can escape the dependency on K. However there is a pre-processing time of $O(n)$ in case any E_j gets updated. But since this pre-processing is needed once per row (due to Fact 8), the computational effort added is $O(n^2)$ in total.

Theorem 6. *Problem FIG can be solved in $O(n^2 + \mathcal{R} \log\log n)$ time using $\theta(\max(\mathcal{R}, n))$ space.*

Finally, it is easy to see that this algorithm can be easily used to solve LCS problem, virtually, without any modification. We, however, can do better using Fact 6 and the subsequent discussion in Section 6. We can get rid of the vEB data structures altogether and use a simple array (array H in Section 6) instead. So we get the following theorem.

Theorem 7. *Problem LCS can be solved in $O(n^2 + \mathcal{R} \log\log n)$ time using $\theta(\max(\mathcal{R}, n))$ space.*

We can shave off the $\log\log n$ term from the running time of Theorem 7 at the cost of using $\theta(n^2)$ space. This can be achieved if we do not use Algorithm 1 as a pre-processing step. In this case, however, we need to process each entry of $\mathcal{T}[i, j], 1 \leq i \leq n, 1 \leq j \leq n$, instead of processing only each $(i, j) \in \mathcal{M}$. Notably, however, we perform 'useful' computation only for each $(i, j) \in \mathcal{M}$. As a result the running time would be $O(n^2 + \mathcal{R})$. Since $\mathcal{R}$ can be n^2 in the worst case, this running time matches that of the classic solution to LCS problem.

Theorem 8. *Problem LCS can be solved in $O(n^2)$ time using $\theta(n^2)$ space.*

8. Algorithm for elastic gapped LCS

In this section we modify the algorithms in Sections 5–7 to solve ELAG. Recall that, in ELAG, we have two parameters, namely K_1 and K_2 and the consecutive letters in the common subsequence must exist within distance K_2 in both X and Y like FIG. But unlike FIG, here, the other parameter K_1, applies the added constraint that the consecutive letters in the common subsequence must **not** exist within distance K_1 in both X and Y. So FIG can be thought of as a special case of ELAG when $K_1 = 0$ and $K_2 = K$. The obvious modification to Eq. (2) in Section 5 to solve ELAG is as follows:

$$\mathcal{T}_{local}[i,j] = \begin{cases} \textit{Undefined} & \text{if } (i,j) \notin \mathcal{M}, \\ \max\limits_{\substack{i-1-K_2 \le \ell_i < i-K_1 \\ j-1-K_2 \le \ell_j < j-K_1 \\ (\ell_i,\ell_j) \in \mathcal{M}}} (\mathcal{T}_{local}[\ell_i,\ell_j]) + 1 & \text{if } (i,j) \in \mathcal{M}. \end{cases} \tag{5}$$

Theorem 9. *Problem ELAG can be solved in $O(n^2 + \mathcal{R}(K+1)^2)$ time using $\theta(n^2)$ space where $K = K_2 - K_1$.*

For Algorithm 2, described in Section 6, the only modification that is needed to solve ELAG, is in Step 8, 9 and 11. The modified statements are as follows:

Modified Step 8: Insert $(i - 1 - K_1, j) \in \mathcal{M}_{i-1}\ in\ E_j, 1 \le j \le n$
Modified Step 9: Delete $(i - K_2 - 2, j) \in \mathcal{M}_{i-K_2-2}\ in\ E_j, 1 \le j \le n$
Modified Step 11: $maxresult = \max_{(j-K_2-1) \le \ell \le (j-K_1)}(\max(E_\ell))$.

Theorem 10. *Problem ELAG can be solved in $O(\mathcal{R}K + \mathcal{R}\log\log n)$ time, using $\theta(\max(\mathcal{R}, n))$ space, where $K = K_2 - K_1$.*

The following result holds if we need the solutions to the subproblems.

Theorem 11. *Problem ELAG can be solved in $O(n^2 + \mathcal{R}K + \mathcal{R}\log\log n)$ time, using $\theta(n^2)$ space, where $K = K_2 - K_1$.*

Finally, it should be clear that in the algorithm in Section 7, virtually, there is no modification at all except for that we have to adjust the Range Maxima Query to incorporate the elastic gap constraint.

Theorem 12. *Problem ELAG can be solved in $O(n^2 + \mathcal{R}\log\log n)$ time, using $\theta(\max(\mathcal{R}, n))$ space.*

9. Algorithms for rigid gapped LCS

This section is dedicated to solve Problem RIFIG and Problem RELAG. RIFIG, by nature, is a bit more restricted because, in addition to the K-gap constraint, the consecutive characters in the common subsequence must have the same distance between them (rigidness) both in X and Y. Interestingly enough, this restriction makes this problem rather easier to solve. And in fact we will see that we can modify the algorithm in Section 5 easily to solve RIFIG and this slight modification would even improve the running time of the algorithm. The key idea lies in the fact that to calculate a $\mathcal{T}_{local}[i,j]$ we just need to check the $K+1$ diagonal entries before it. This is true because of the required *rigidness*. The modified version of Eq. (2) to handle RIFIG is given below.

$$\mathcal{T}_{local}[i,j] = \begin{cases} \textit{Undefined} & \text{if } (i,j) \notin \mathcal{M}, \\ \max\limits_{\substack{(\ell_i,\ell_j) \in \{(i-1,j-1),(i-2,j-2) \\ \ldots(i-1-K,j-1-K)\} \\ (\ell_i,\ell_j) \in \mathcal{M}}} (\mathcal{T}_{local}[\ell_i,\ell_j]) + 1 & \text{if } (i,j) \in \mathcal{M}. \end{cases} \tag{6}$$

Also we can easily modify Eq. (6) to solve RELAG. So we get the following theorems.

Theorem 13. *Problem RIFIG can be solved in $O(n^2 + \mathcal{R}K)$ time using $\theta(n^2)$ space.*

Theorem 14. *Problem RELAG can be solved in $O(n^2 + \mathcal{R}(K_2 - K_1))$ time using $\theta(n^2)$ space.*

In the rest of this section we will try to achieve better solutions for RIFIG. We first introduce a variant of RIFIG where the gap constraint is withdrawn. In other words we can say that in this variant we have $K = n$.

Problem 10 (*"RLCS" (Rigid LCS Problem)*[4]). Given two strings X and Y, each of length n, we want to find out a Rigid Common Subsequence of the maximum length. A Rigid Common Subsequence of X and Y is a subsequence $S[1..r] = S[1]\ S[2] \ldots S[r]$ of both X and Y such that $C(X,S)[i] - C(X,S)[i-1] = C(Y,S)[i] - C(Y,S)[i-1]$ for all $2 \le i \le r$.

It is easy to see that using Eq. (6) we can easily solve Problem RLCS by assuming $K = n$. But this will not give us a very good running time at all. On the other hand, it turns out that we can achieve a better running time by appropriate modification to Eq. (1). To solve RLCS, however, for each tabular entry $\mathcal{T}[i, j]$ we calculate and store two values namely $\mathcal{T}_{local}[i, j]$, $\mathcal{T}_{global}[i, j]$. The recurrence relations are defined below:

$$\mathcal{T}_{local}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \mathcal{T}_{local}[i-1, j-1] + 1 & \text{if } X[i] = Y[j], \\ \mathcal{T}_{local}[i-1, j-1] & \text{if } X[i] \neq Y[j]. \end{cases} \tag{7}$$

$$\mathcal{T}_{global}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1]) & \text{if } (i, j) \notin \mathcal{M}, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1], \mathcal{T}_{local}[i, j] & \text{if } (i, j) \in \mathcal{M}. \end{cases} \tag{8}$$

It is easy to see that Eq. (7) preserves the rigidness of the subsequence. Note that, Eq. (8) is required to keep track of the global solution.

Theorem 15. *Problem RLCS can be solved in $O(n^2)$ time using $\theta(n^2)$ space.*

Inspired by the idea of the above solution to RLCS in the rest of this section we try to devise an algorithm to solve RIFIG in $O(n^2)$ time. The idea is to some how propagate the *constraint information* up through the diagonal entries as soon as we find a match and whenever a match is found check this information. What we plan to do is as follows. For the calculation of $\mathcal{T}_{local}$ we apply K-modulo arithmetic. The actual length of LCS would be $\lceil \mathcal{T}_{local}[n, n]/K \rceil$.

$$\mathcal{T}_{local}[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 0 & \text{if } X[i] \neq Y[j] \text{ and} \\ & \mathcal{T}_{local}[i-1, j-1] \bmod K = 1, \\ \lceil \mathcal{T}_{local}[i-1, j-1]/K \rceil * K + K & \text{if } X[i] = Y[j], \\ \mathcal{T}_{local}[i-1, j-1] - 1 & \text{if } X[i] \neq Y[j] \text{ and} \\ & \mathcal{T}_{local}[i-1, j-1] > 0, \\ 0 & \text{if } X[i] \neq Y[j] \text{ and} \\ & \mathcal{T}_{local}[i-1, j-1] = 0. \end{cases} \tag{9}$$

$$\mathcal{T}_{global}[i, j] = \begin{cases} \mathcal{T}_{global}[i, j] & \text{if } i = 1 \text{ or } j = 1, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1]) & \text{if } (i, j) \notin \mathcal{M}, \\ \max(\mathcal{T}_{global}[i-1, j], \mathcal{T}_{global}[i, j-1], \mathcal{T}_{local}[i, j] & \text{if } (i, j) \in \mathcal{M}. \end{cases} \tag{10}$$

Theorem 16. *Problem RIFIG can be solved in $O(n^2)$ time using $\theta(n^2)$ space.*

For both Problem RIFIG and RLCS, the space complexity can be improved to linear using a different approach as follows. It is easy to realize that, due to the rigidity, the solution in each case, is always the meet of two suffixes. So, we can superimpose one string to the other in all its $2n - 1$ ways and then check all matches in linear time. This results in a different algorithm for both the problems, exhibiting linear space requirement with the same running time. Further, it seems to be possible to reduce the running time to linear, combining careful algorithmic design with techniques similar to generalized suffix tree [15].

10. Applications

In this section we discuss extensions of some interesting problems in stringology and show how they can be solved using the algorithms described in this paper. In particular we present some problems motivated by biological applications and discuss efficient solutions for them. We first show how we can tackle the degenerate strings in biological applications, for e.g., using a technique invented by Lee et al. [21]. This gives us the opportunity to handle degenerate strings in all the variants of the LCS problem we discuss in this paper. Moreover we show how we can solve the Longest Common Substring problem for degenerate strings, a very common problem in molecular biology. We start with the definition of degenerate strings.

Definition 6 (*Degenerate String*). A string X is said to be degenerate, if it is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters belonging to Σ.

Example 5. Suppose that we are considering DNA alphabet i.e. $\Sigma = \Sigma_{DNA} = \{A, C, T, G\}$. Then we have 15 non-empty sets of letters belonging to Σ_{DNA}. In what follows, the set containing A and T will be denoted by $[AT]$ and the singleton $[C]$ will be simply denoted by C for ease of reading. The set containing all the letters, namely $[ACTG]$, is known as the do not care character in the literature.

Definition 7 (*Degenerate Equality/Matching*). For degenerate strings the notion of *matching/equality* between two letters is extended in the following way. Given two degenerate strings X and Y each of length n, we say that $X[i]$ *matches* $Y[j]$, $1 \le i, j \le n$, denoted by $X[i] =_d Y[j]$ if $X[i] \cap Y[j] \ne \emptyset$.

Example 6. Suppose that we have degenerate strings $X = AC[CTG]TG[AC]C$ and $Y = TC[AT][AT]TTC$. Here we have $X[3] =_d Y[3]$ because $X[3] = [CTG] \cap Y[3] = [AT] = T \ne \emptyset$. Similarly we have, $X[3] =_d Y[1]$, and also $X[3] =_d Y[2]$ etc.

With the definitions of degenerate strings and the extended notion of degenerate equality it is easy to extend the definitions of our problems for degenerate strings. To handle the degenerate strings we first present a clever technique presented in [21] where the authors applied a bit masking technique exploiting the property of bitwise 'and' operations as follows. Assuming a predefined order among the letters of the alphabet, each letter of the alphabet is encoded using $|\Sigma|$ bits where a '1' in a bit position indicates the presence of that letter. For example if we consider DNA alphabet $\Sigma_{DNA} = \{A, C, T, G\}$ in the given order, then the codes for A, C, G, and T would be, respectively, 1000, 0100, 0010, and 0001. Sets of characters are also encoded in the same way. For example, [AC] and [CTG] would be encoded, respectively, as 1100 and 0111. Interestingly enough, this simple but clever encoding works perfectly with the notion of degenerate equality/matching. The idea is to perform bitwise 'and' operations among the degenerate letters. If the 'and' operation returns 0, it means no match; otherwise we have a match.

Example 7. Suppose that we have degenerate strings of Example 6. Then the encoding of the two strings would be as follows.

$$X' = 1000\,0100\,0111\,0010\,0001\,1100\,0100$$
$$Y' = 0010\,0100\,1010\,1010\,0010\,0010\,0100.$$

Here we have $X[3] =_d Y[3]$ because $X[3](= 0111)$ and $Y[3](= 1010) = 0010 \ne 0$. However $X[6] \ne_d Y[1]$ because $X[6](= 1100)$ and $Y[1](= 0010) = 0000 = 0$.

Hence it is easy to see that we can pre-process the two input strings $X = X[1]X[2]...X[n]$ and $Y = Y[1]Y[2]...Y[n]$ using the technique of [21] to get X' and Y' and then employ a modified version of our algorithms on X' and Y'. The only modification that need to be done is to use bitwise 'and' operation between $X'[(i-1)*|\Sigma|+1..(i-1)*|\Sigma|+|\Sigma|]$ and $Y'[(j-1)*|\Sigma|+1..(j-1)*|\Sigma|+|\Sigma|]$ and to check whether the result is 0 or not instead of 'normal' equality checking between $X[i]$ and $Y[j]$. What would be the complexity of the modified algorithm? It seems that the traditional LCSP dynamic programming algorithm would require $O(|\Sigma|n^2)$ time. For DNA and protein alphabet this running time is not very high at all because $|\Sigma_{DNA}| = 4$ and $|\Sigma_{Protein}| = 20$. However, we can do better. As we can do the 'and' operations letter by letter using word-by-word 'and' operation. So, as long as the word size of the target machine, w, is greater than or equal to $|\Sigma|$ (which is almost always the case for biological applications) we would still have $O(n^2)$ running time to handle the degenerate strings with the traditional Dynamic Programming algorithm to solve LCSP. In other words, as long as $w \ge |\Sigma|$, we can solve LCSP for degenerate strings with the same running time that is needed to solve LCSP for 'normal' strings. It is also easy to see that this fact is also applicable for all the variants of LCSP we have discussed in this paper.

We note however that the algorithms exploiting the pre-processing steps of Algorithm 1, are excluded from the above discussion. This is because, when we use Algorithm 1 as a pre-processing step to construct the set $\mathcal{M}$, we do not need to use any equality checking later. As a result, in these cases, in order to handle degenerate strings, we just need to modify Algorithm 1. It is easy to see that the only modification that need be done is in Step 2, Step 3 of Algorithm 1. The idea is to ensure that for set of character at $X[i]$ ($Y[i]$) we must add i in $L_X[a]$ ($L_Y[a]$) for all $a \in X[i]$ ($a \in Y[i]$). This can be done $O(|\Sigma|n)$ time as opposed to $O(n)$ in Algorithm 1 because a set of character can be of at most of $|\Sigma|$ cardinality. Again, we emphasize that, due to very small size of $|\Sigma|$ in biological applications, this would not affect the asymptotic behavior of the algorithms.

Finally we conclude this section considering another interesting problem, again motivated by computational biology, namely, Longest Common Substring problem for degenerate string.

Definition 8 (*Substring*). Given a string $X = X[1]X[2]...X[n]$, a substring of X, denoted by $X[i..j]$ is the string $X[i]X[i+1]...X[j]$, $1 \le i \le j \le n$.

Problem 11 (*Longest Common Substring*). Given two strings X and Y each of length n we want to find out a substring of maximum length common to both X and Y.

The longest common substring problem can easily be solved in linear time, for e.g. with the help of generalized suffix tree. However, to the best of our knowledge there has been no algorithms in the literature for solving the same problem for degenerate strings. Fortunately we can solve this problem using our algorithms for FIG. This is because, as can be easily seen, if $K = 0$, then FIG reduces to the longest common substring problem. We note however that this would not be a linear algorithm.

11. Conclusion

In this paper we have introduced several new variants of the LCS problem and presented efficient algorithms to solve them. In particular, we introduced the idea of gap constraint in LCS and defined a new set of problems. For the LCS problem with fixed gap (Problem FIG), we first presented a naive algorithm runs in $O(n^2 + \mathcal{R}(K+1)^2)$ time and then improved the running time to $O(n^2 + \mathcal{R}K + \mathcal{R}\log\log n)$ using some novel techniques. Furthermore, we presented an algorithm that is independent of K and runs in $O(n^2 + \mathcal{R}\log\log n)$ time. These techniques lead to a new $O(n^2 + \mathcal{R}\log\log n)$ algorithm to solve the original LCS problem. Then we modify our algorithms to handle elastic (Problem ELAG) and rigid gaps (Problem RIFIG and RELAG). We also apply the notion of rigidness to the original LCS problem and modify the traditional dynamic programming solution to handle the rigidness (Problem RLCS) presenting an $O(n^2)$ algorithm to solve the problem. Inspired by the solution of the Problem RLCS, we also improve the solution to Problem RIFIG to $O(n^2)$.

We also show that our algorithms can be used to solve some other interesting problems in bioinformatics. We have shown how we can tackle efficiently the degenerate strings in biological applications. This gives us the opportunity to handle degenerate strings in all the variants of the LCS problem, discussed in this paper, virtually without any increase in the asymptotic running time. Moreover we can solve the *Longest Common Substring* problem for degenerate strings, a very common problem in molecular biology simply by setting $K = 0$ in the Problem FIG. To the best of our knowledge there does not exist any algorithm to solve this problem for degenerate strings.

Acknowledgements

The first author was supported by EPSRC and Royal Society grants. The second author was supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).

References

[1] ED'NIMBUS. http://igm.univ-mlv.fr/peterlon/officiel/ednimbus/.

[2] Identification of a src SH3 domain binding motif by screening a random phage display library, J. Biol. Chem. 269 (1994) 24034–24039.

[3] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Meyers, David J. Lipman, Basic local alignment search tool, J. Molecular Biol. 215 (3) (1990) 403–410.

[4] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev, On economic construction of the transitive closure of a directed graph, Soviet Math. Dokl. 11 (1975) 1209–1210. (English translation).

[5] Abdullah N. Arslan, Ömer Egecioglu, Algorithms for the constrained longest common subsequence problems, in: Milan Simánek, Jan Holub (Eds.), The Prague Stringology Conference, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2004, pp. 24–32.

[6] Abdullah N. Arslan, Ömer Egecioglu, Algorithms for the constrained longest common subsequence problems, Int. J. Found. Comput. Sci. 16 (6) (2005) 1099–1109.

[7] Michael A. Bender, Martin Farach-Colton, The lca problem revisited, in: Gaston H. Gonnet, Daniel Panario, Alfredo Viola (Eds.), Latin American Theoretical INformatics, LATIN, in: Lecture Notes in Computer Science, vol. 1776, Springer, 2000, pp. 88–94.

[8] Sergey Bereg, Binhai Zhu, RNA multiple structural alignment with longest common subsequences, in: Lusheng Wang (Ed.), Computing and Combinatorics (COCOON), in: Lecture Notes in Computer Science, vol. 3595, Springer, 2005, pp. 32–41.

[9] Lasse Bergroth, Harri Hakonen, Timo Raita, A survey of longest common subsequence algorithms, in: String Processing and Information Retrieval (SPIRE), IEEE Computer Society, 2000, pp. 39–48.

[10] Guillaume Blin, Guillaume Fertin, Romeo Rizzi, Stéphane Vialette, What makes the arc-preserving subsequence problem hard? in: Vaidy S. Sunderam, G. Dick van Albada, Peter M.A. Sloot, Jack Dongarra (Eds.), International Conference on Computational Science, in: Lecture Notes in Computer Science, vol. 3515, Springer, 2005, pp. 860–868.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to Algorithms, The MIT Press and McGraw-Hill Book Company, 1989.

[12] Maxime Crochemore, Gad M. Landau, Michal Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, in: Symposium of Discrete Algorithms (SODA), 2002, pp. 679–688.

[13] P.A. Evans, Algorithms and complexity for annotated sequence analysis, Ph.D. Thesis, University of Victoria, 1999.

[14] H. Gabow, J. Bentley, R. Tarjan, Scaling and related techniques for geometry problems, in: Symposium on the Theory of Computing, STOC, ACM Press, New York, NY, USA, 1984, pp. 135–143. Chairman-Richard DeMillo.

[15] Dan Gusfield, Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology, Cambridge University Press, 1997.

[16] F. Hadlock, Minimum detour methods for string or sequence comparison, Congr. Numer. 61 (1988) 263–274.

[17] Daniel S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.

[18] James W. Hunt, Thomas G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.

[19] Tao Jiang, Ming Li, On the approximation of shortest common supersequences and longest common subsequences, SIAM J. Comput. 24 (5) (1995) 1122–1139.

[20] Tao Jiang, Guohui Lin, Bin Ma, Kaizhong Zhang, The longest common subsequence problem for arc-annotated sequences, in: Raffaele Giancarlo, David Sankoff (Eds.), Combinatorial Pattern Matching (CPM), in: Lecture Notes in Computer Science, vol. 1848, Springer, 2000, pp. 154–165.

[21] I. Lee, A. Apostolico, C.S. Iliopoulos, K. Park, Finding approximate occurrence of a pattern that contains gaps, in: M. Miller, K. Park (Eds.), Australasian Workshop on Combinatorial Algorithms, AWOCA, 2003, pp. 89–100.

[22] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Probl. Inf. Transm. 1 (1965) 8–17.

[23] Guo-Hui Lin, Zhi-Zhong Chen, Tao Jiang, Jianjun Wen, The longest common subsequence problem for sequences with nested arc annotations, J. Comput. Syst. Sci. 65 (3) (2002) 465–480.

[24] Bin Ma, Kaizhong Zhang, On the longest common rigid subsequence problem, in: Alberto Apostolico, Maxime Crochemore, Kunsoo Park (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 3537, Springer, 2005, pp. 11–20.

[25] David Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.

[26] William J. Masek, Mike Paterson, A faster algorithm computing string edit distances, J. Comput. Syst. Sci. 20 (1) (1980) 18–31.

[27] Eugene W. Myers, An O(ND) difference algorithm and its variations, Algorithmica 1 (2) (1986) 251–266.

[28] Veli Mkinen, Gonzalo Navarro, Esko Ukkonen, Transposition invariant string matching, J. Algorithms 56 (2005) 124–153.

[29] Narao Nakatsu, Yahiko Kambayashi, Shuzo Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inf. 18 (1982) 171–179.

[30] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Natl. Acad. Sci. USA 85 (1988) 2444–2448.

[31] Pierre Peterlongo, Private communication.

[32] Pierre Peterlongo, Nadia Pisanti, Frédéric Boyer, Marie-France Sagot, Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array, in: Mariano P. Consens, Gonzalo Navarro (Eds.), SPIRE, in: Lecture Notes in Computer Science, vol. 3772, Springer, 2005, pp. 179–190.

[33] Yin-Te Tsai, The constrained longest common subsequence problem, Inform. Process. Lett. 88 (4) (2003) 173–176.

[34] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (1977) 80–82.

[35] Robert A. Wagner, Michael J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.