

Towards Optimally Solving the LONGEST COMMON SUBSEQUENCE Problem for Sequences with Nested Arc Annotations in Linear Time

Jochen Alber^{*}, Jens Gramm^{**}, Jiong Guo^{***}, and Rolf Niedermeier

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
{alber, gramm, guo, niedermr}@informatik.uni-tuebingen.de

Abstract. We present exact algorithms for the NP-complete LONGEST COMMON SUBSEQUENCE problem for sequences with nested arc annotations, a problem occurring in structure comparison of RNA. Given two sequences of length at most n and nested arc structure, our algorithm determines (if existent) in time $O(3.31^{k_1+k_2} \cdot n)$ an arc-preserving subsequence of both sequences, which can be obtained by deleting (together with corresponding arcs) k_1 letters from the first and k_2 letters from the second sequence. Thus, the problem is fixed-parameter tractable when parameterized by the number of deletions. This complements known approximation results which give a quadratic time factor-2-approximation for the general and polynomial time approximation schemes for restricted versions of the problem. In addition, we obtain further fixed-parameter tractability results for these restricted versions.

1 Introduction

Given two or more sequences over some fixed alphabet Σ , the LONGEST COMMON SUBSEQUENCE problem (LCS) asks for a maximum length sequence that occurs as a subsequence in all of the given input sequences. This is considered to be a core problem of computer science with many applications, see, e.g., [4, 5, 12, 18, 19]. With the advent of computational biology, structure comparison of RNA and of protein sequences has become a central computational problem, bearing many challenging computer science questions. In this context, the LONGEST ARC PRESERVING COMMON SUBSEQUENCE problem (LAPCS) recently has received considerable attention [6–8, 13, 17]. It is a sound and meaningful mathematical formalization of comparing the secondary structures of molecular sequences: For a sequence S , an *arc annotation* A of S is a set of unordered pairs of positions

^{*} Supported by the Deutsche Forschungsgemeinschaft (DFG), project PEAL (parameterized complexity and exact algorithms), NI 369/1-1,1-2.

^{**} Supported by the Deutsche Forschungsgemeinschaft (DFG), project OPAL (optimal solutions for hard problems in computational biology), NI 369/2-1.

^{***} Partially supported by the Deutsche Forschungsgemeinschaft (DFG), Zentrum für Bioinformatik Tübingen (ZBIT).

in S . Focusing on the case of two given arc-annotated input sequences S_1 and S_2 , LAPCS asks for a maximum length arc-annotated sequence T that forms an arc-annotated subsequence of S_1 as well as S_2 . More precisely, this means that if one deletes k_1 letters (also called *bases*) from S_1 —when deleting a letter at position i , then *all* arcs with endpoint i are also deleted—and one deletes k_2 letters from S_2 , then T and the resulting sequences are the same and also the arc annotations coincide. For related studies concerning algorithmic aspects of (protein) structure comparison using “contact maps,” refer to [10, 15].

Whereas LCS for two sequences without arc annotations is solvable in quadratic time (it becomes NP-complete when allowing for an arbitrary number of input sequences), LAPCS for two sequences is NP-complete [6, 7]. According to Lin *et al.* [17], however, LAPCS for *nested arc annotations* is “generally thought of as the most important variant of the LAPCS problem.” Here, one requires that no two arcs share an endpoint and no two arcs cross each other. This problem is referred to by LAPCS(NESTED,NESTED). Answering an open question of Evans [6], Lin *et al.* [17] showed that LAPCS(NESTED,NESTED) is NP-complete. In addition, they gave polynomial time approximation schemes (PTAS) for (also NP-complete) special cases of LAPCS(NESTED,NESTED). Here, matches between two given input sequences are allowed only in a “local area” (of constant size) w.r.t. matching position numbers (refer to Section 2 for details). As to the general LAPCS(NESTED,NESTED) problem, Jiang *et al.* [13] gave a quadratic time factor-2-approximation algorithm.

By way of contrast, our main result is an exact, fixed-parameter algorithm that solves the general LAPCS(NESTED,NESTED) problem in time $O(3.31^{k_1+k_2} \cdot n)$ where n is the maximum input sequence length. This gives an efficient algorithm in case of reasonably small values for k_1 and k_2 (the numbers of deletions allowed in S_1 and S_2 , respectively), providing an *optimal* solution. Observe that the PTAS algorithms obtaining a good degree of approximation get prohibitive from a practical point of view due to enormous running times. And, the factor-2-approximation might be not good enough for biological and other applications. In summary, in terms of parameterized complexity [1, 5, 9], our result shows that LAPCS(NESTED,NESTED) is fixed-parameter tractable when parameterized by the number of deletions allowed, perhaps the most natural problem parameterization. Moreover, we obtain fixed-parameter tractability results for restricted versions of LAPCS(NESTED,NESTED) as studied by Lin *et al.* [17].¹ In the restricted cases, we *additionally* prove fixed-parameter tractability when taking the length of the common subsequence as the problem parameter. Our algorithms employ the bounded search tree paradigm of parameterized complexity in a highly nontrivial way.

¹ Actually, we show fixed-parameter tractability for more general problems, allowing more complicated (e.g., crossing) arc annotations.

2 Preliminaries and Previous Work

Besides its central importance in computer science and applications, the classical, NP-complete LCS is particularly important from the viewpoint of parameterized complexity [1, 5, 9]. It is used as a key problem for determining the parameterized complexity of many problems from computational biology. In this setting, the input consists of several sequences over an alphabet Σ and a positive integer l , the question is whether there is a sequence from Σ^* of length at least l that is a subsequence of *all* input sequences. For example, with respect to parameter l the problem is known to be *fixed-parameter tractable*, that is, it can be solved in time $f(l) \cdot N^{O(1)}$, where N denotes the complete input size and f can be an arbitrarily fast growing function only depending on l .² The decisive point here is that the combinatorial explosion seemingly unavoidable in optimally solving NP-hard problems can be completely restricted to the *parameter* l . Note, however, that for LCS this requires that $|\Sigma|$ is of constant size. If $|\Sigma|$ is unbounded, then there is concrete indication that LCS is *not* fixed-parameter tractable with respect to parameter l (by way of so-called W[2]-hardness which is well-known in parameterized complexity). We refer to [2, 3, 5] for any details.

Evans [6, 7] initiated classical and parameterized complexity studies for the more general case that the input sequences additionally carry an *arc structure* each, which is motivated by structure comparison problems in computational molecular biology. For a sequence S of length $|S| = n$, an *arc annotation* (or *arc set*) A of S is a set of unordered pairs of numbers from $\{1, 2, \dots, n\}$. Each pair (i, j) connects the two *bases* $S[i]$ and $S[j]$ at positions i and j in S by an arc. Since LAPCS is NP-complete even for two input sequences, here and in the literature attention is focused on this case. Let S_1 and S_2 be two sequences with arc sets A_1 and A_2 , respectively, and let i_1, i_2, j_1, j_2 be positive integers. If $S_1[i_1] = S_2[j_1]$, we refer to this as *base match* and if $S_1[i_1] = S_2[j_1]$, $S_1[i_2] = S_2[j_2]$, $(i_1, i_2) \in A_1$, and $(j_1, j_2) \in A_2$, we refer to this as *arc match*. A common subsequence T of S_1 and S_2 induces a one-to-one mapping M from a subset of $\{1, 2, \dots, n_1\}$ to a subset of $\{1, 2, \dots, n_2\}$, where $n_1 = |S_1|$ and $n_2 = |S_2|$. Suppose that $M = \{(i_r, j_r) \mid 1 \leq r \leq |T|\}$. Then T *induces* the base matches $\langle i_r, j_r \rangle$, $1 \leq r \leq |T|$. We say that T is an *arc-preserving subsequence (aps)* if the arcs induced by M are preserved, i.e., for all $\langle i_{r_1}, j_{r_1} \rangle, \langle i_{r_2}, j_{r_2} \rangle \in M$:

$$(i_{r_1}, i_{r_2}) \in A_1 \iff (j_{r_1}, j_{r_2}) \in A_2.$$

Depending on the arc structures allowed, various subproblems of LAPCS can be defined. We focus here on the NP-complete LAPCS(NESTED,NESTED) [13, 17], where for both input sequences it is required that no two arcs share an endpoint and that no two arcs cross each other, i.e., for all $(i_1, i_2), (i_3, i_4)$ with $i_1 < i_2$ and $i_3 < i_4$ part of the arc annotation of a sequence, it holds that $i_3 < i_1 < i_4$ iff $i_3 < i_2 < i_4$. We will also consider the less restrictive *crossing* arc structure, where the only requirement is that no two arcs share an endpoint. Lin *et al.* [17] furthermore introduced the following special cases of LAPCS(NESTED,NESTED),

² Actually, in case of LCS running time $O(|\Sigma|^l \cdot N)$ is easily achieved.

motivated from biological applications [12, 16]. For any positive integer c , in c -DIAGONAL LAPCS(NESTED,NESTED) base $S_1[i]$ ($S_2[j]$, respectively) is allowed to match only bases in the range $S_2[i - c, i + c]$ ($S_1[j - c, j + c]$, respectively). Herein, for $i_1 \leq i_2$, $S[i_1, i_2]$ denotes the substring of sequence S starting at position i_1 and ending at position i_2 and $S[i_1, +]$ denotes the suffix of S starting at position i_1 . Similarly, in c -FRAGMENT LAPCS(NESTED,NESTED) all base matches induced by a *longest arc-preserving common subsequence (lapcs)* are required to be between “corresponding” fragments of size c of S_1 and S_2 . Lin *et al.* [17] showed that c -DIAGONAL LAPCS(NESTED,NESTED) is NP-complete for $c \geq 1$ and c -FRAGMENT LAPCS(NESTED,NESTED) is NP-complete for $c \geq 2$.

3 Search Tree Algorithm for LAPCS(NESTED,NESTED)

In this section, we describe and analyze Algorithm LAPCS which solves the LAPCS(NESTED, NESTED) problem in time $O(3.31^{k_1+k_2} \cdot n)$, where n is the maximum length of the input sequences. It is a search tree algorithm and, for sake of clarity, we choose the presentation in a recursive style: Based on the current instance, we make a case distinction, branch into one or more subcases of somehow simplified instances and invoke the algorithm recursively on each of these subcases. Note, however, that we require to traverse the resulting search tree in breadth-first manner, which will be important in the running time analysis. Before presenting the algorithm, we define the employed notation.

Recall that the considered sequences are seen as arc-annotated sequences; a comparison $S_1 = S_2$ includes the comparison of arc structures. Additionally, we use a modified comparison $S_1 \approx_{i,j} S_2$ that is satisfied when $S_1 = S_2$ after deleting at most i bases in S_1 and at most j bases in S_2 . Note that we can check whether $S_1 \approx_{1,0} S_2$ or whether $S_1 \approx_{0,1} S_2$ in linear time. The subsequence obtained from an arc-annotated sequence S by deleting $S[i]$ is denoted by $S - S[i]$. When branching into the case of a simplified sequence $S - S[i]$ (or $S[2, n]$, resp.), the input for the recursive call is $S_{new} := S - S[i]$ (or $S_{new} := S[2, n]$, resp.)—hence, $|S_{new}| = |S| - 1$ —and, therefore, $S_{new}[i] = S[i + 1]$. For handling branches in which no solution is found, we use a modified addition operator “ $\dot{+}$ ” defined as follows: $a \dot{+} b := a + b$ if $a \geq 0$ and $b \geq 0$, and $a \dot{+} b := -1$, otherwise. We abbreviate $n_1 := |S_1|$ and $n_2 := |S_2|$.

The most involved case in the algorithm is Case (2.5), which will also determine our upper bound on the search tree size. The focus of our analysis will, in particular, be on Subcase (2.5.3). For sake of clarity, we, firstly, give an overview of the algorithm which omits the details of Case (2.5), and, then, present Case (2.5) in detail separately. Although the algorithm as given reports only the length of an lapcs, it can easily be extended to compute the lapcs itself within the same running time.

Algorithm LAPCS(S_1, S_2, k_1, k_2)

Input: Arc-annotated sequences S_1 and S_2 , positive integers k_1 and k_2 .

Return value: Integer denoting the length of an lapcs of S_1 and S_2 which can

be obtained by deleting at most k_1 symbols in S_1 and at most k_2 symbols in S_2 .
Return value -1 if no such subsequence exists.

(Case 0) */* Recursion ends. */*
 If $k_1 < 0$ or $k_2 < 0$ then return -1 . */* No solution found. */*
 If $|S_1| = 0$ and $|S_2| = 0$, then return 0 . */* Success! Solution found. */*
 If $|S_1| = 0$ and $|S_2| > 0$, then */* One sequence done... */*
 if $k_2 \geq |S_2|$, then return 0 , else return -1 . */* ...but not the other. */*
 If $|S_1| > 0$ and $|S_2| = 0$, then */* ditto */*
 if $k_1 \geq |S_1|$, then return 0 , else return -1 .

(Case 1) */* Non-matching bases. */*
 If $S_1[1] \neq S_2[1]$, then return the maximum of the following values:
 • $\text{LAPCS}(S_1[2, n_1], S_2, k_1 - 1, k_2)$ */* delete $S_1[1]$ */*
 • $\text{LAPCS}(S_1, S_2[2, n_2], k_1, k_2 - 1)$ */* delete $S_2[1]$ */*.

(Case 2) */* Matching bases */*
 If $S_1[1] = S_2[1]$, then
 (2.1) */* No arcs involved. */*
 If both $S_1[1]$ and $S_2[1]$ are not endpoints of arcs, then return
 $1 + \text{LAPCS}(S_1[2, n_1], S_2[2, n_2], k_1, k_2)$.
 / Since no arcs are involved, it is safe to match the bases. */*
 (2.2) */* Only one arc. */*
 If $S_1[1]$ is left endpoint of an arc $(1, i)$ but $S_2[1]$ is not endpoint of an arc,
 then return the maximum of the following values:
 • $\text{LAPCS}(S_1[2, n_1], S_2, k_1 - 1, k_2)$ */* delete $S_1[1]$ */*,
 • $\text{LAPCS}(S_1, S_2[2, n_2], k_1, k_2 - 1)$ */* delete $S_2[1]$ */*, and
 • $1 + \text{LAPCS}((S_1 - S_1[i])[2, +], S_2[2, n_2], k_1 - 1, k_2)$ */* match */*.
 / Since there is an arc in one sequence only, $S_1[1]$ and $S_2[1]$ can be
 matched only if $S_1[i]$ is deleted. */*
 (2.3) */* Only one arc. */*
 If $S_2[1]$ is left endpoint of an arc $(1, j)$ but $S_1[1]$ is not endpoint of an arc,
 then proceed analogously as in (2.2).
 (2.4) */* Non-matching arcs. */*
 If $S_1[1]$ is left endpoint of an arc $(1, i)$, $S_2[1]$ is left endpoint of an arc $(1, j)$
 and $S_1[i] \neq S_2[j]$, then return the maximum of the following values:
 • $\text{LAPCS}(S_1[2, n_1], S_2, k_1 - 1, k_2)$ */* delete $S_1[1]$ */*,
 • $\text{LAPCS}(S_1, S_2[2, n_2], k_1, k_2 - 1)$ */* delete $S_2[1]$ */*, and
 • $1 + \text{LAPCS}((S_1 - S_1[i])[2, +], (S_2 - S_2[j])[2, +], k_1 - 1, k_2 - 1)$ */* match */*.
 / Since the arcs cannot be matched, $S_1[1]$ and $S_2[1]$ can be matched only
 if $S_1[i]$ and $S_2[j]$ are deleted. */*
 (2.5) */* An arc match is possible. */*
 If $S_1[1]$ is left endpoint of an arc $(1, i)$, $S_2[1]$ is left endpoint of an arc $(1, j)$,
 and $S_1[i] = S_2[j]$, then go through Cases (2.5.1), (2.5.2), and (2.5.3) which
 are presented below (one of them will apply and will return the length of
 the lapcs of S_1 and S_2 , if such an lapcs can be obtained with k_1 deletions
 in S_1 and k_2 deletions in S_2 , or will return -1 , otherwise).

In Case (2.5), it is possible to match arcs $(1, i)$ in S_1 and $(1, j)$ in S_2 since $S_1[1] = S_2[1]$ and $S_1[i] = S_2[j]$. Our first observation is that, if $S_1[2, i - 1] = S_2[2, j - 1]$ (which will be handled in Case (2.5.1)) or if $S_1[i + 1, n_1] = S_2[j + 1, n_2]$ (which will be handled in Case (2.5.2)), it is safe to match arc $(1, i)$ with arc $(1, j)$: no longer apcs would be possible when not matching them. We match the equal parts of the sequences (either those inside arcs or those following the arcs) and call Algorithm LAPCS recursively only on the remaining subsequences. These cases only simplify the instance and do not require to branch into several subcases:

- (2.5.1)** */* Sequences inside the arcs match. */*
 If $S_1[2, i - 1] = S_2[2, j - 1]$, then return
 $i \dot{+} \text{LAPCS}(S_1[i + 1, n_1], S_2[j + 1, n_2], k_1, k_2)$.
- (2.5.2)** */* Sequences following the arcs match. */*
 If $S_1[i + 1, n_1] = S_2[j + 1, n_2]$, then return
 $2 \dot{+} (n_1 - i) \dot{+} \text{LAPCS}(S_1[2, i - 1], S_2[2, j - 1], k_1, k_2)$.

If neither Case (2.5.1) nor Case (2.5.2) applies, this is handled by Case (2.5.3), which branches into four recursive calls: we have to consider breaking at least one of the arcs (handled by the first three recursive calls in (2.5.3)) or to match the arcs (handled by the fourth recursive call in (2.5.3)):

(2.5.3) Return the maximum of the following four values:

- $\text{LAPCS}(S_1[2, n_1], S_2, k_1 - 1, k_2)$ */* delete $S_1[1]$. */*,
- $\text{LAPCS}(S_1, S_2[2, n_2], k_1, k_2 - 1)$ */* delete $S_2[1]$. */*,
- $1 \dot{+} \text{LAPCS}((S_1 - S_1[i])[2, +], (S_2 - S_2[j])[2, +], k_1 - 1, k_2 - 1)$
/ match $S_1[1]$ and $S_2[1]$, but do not match arcs $(1, i)$ and $(1, j)$; this implies the deletion of $S_1[i]$, $S_2[j]$, and the incident arcs. */*,
- l (computed as given below) */* match the arcs. */*

Value l denotes the length of the lapcs of S_1 and S_2 in case of matching arc $(1, i)$ with arc $(1, j)$. It can be computed as the sum of the lengths l' , denoting the length of an lapcs of $S_1[2, i - 1]$ and $S_2[2, j - 1]$, and l'' , denoting the length of an lapcs of $S_1[i + 1, n_1]$ and $S_2[j + 1, n_2]$; each of l' and l'' can be computed by one recursive call. Remember that we already excluded $S_1[2, i - 1] = S_2[2, j - 1]$ (by Case (2.5.1)) and $S_1[i + 1, n_1] = S_2[j + 1, n_2]$ (by Case (2.5.2)). For the running time analysis, however, we will require that the deletion parameters k_1 and k_2 will be decreased by two in both recursive calls computing l' and l'' . Therefore, we will further exclude those special cases in which l' or l'' can be found by exactly one deletion, either in S_1 or in S_2 (this can be checked in linear time); then, we need only one recursive call to compute l . Only if this is not possible, we will invoke the two calls for l' and l'' . Therefore, l is computed as follows:

$$l := \begin{cases} j \dot{+} \text{LAPCS}(S_1[i+1, n_1], S_2[j+1, n_2], k_1-1, k_2) & \text{if } S_1[2, i-1] \approx_{1,0} S_2[2, j-1], \\ i \dot{+} \text{LAPCS}(S_1[i+1, n_1], S_2[j+1, n_2], k_1, k_2-1) & \text{if } S_1[2, i-1] \approx_{0,1} S_2[2, j-1], \\ 2 \dot{+} (n_2 - j) \dot{+} \text{LAPCS}(S_1[2, i-1], S_2[2, j-1], k_1-1, k_2) & \text{if } S_1[i+1, n_1] \approx_{1,0} S_2[j+1, n_2], \\ 2 \dot{+} (n_1 - i) \dot{+} \text{LAPCS}(S_1[2, i-1], S_2[2, j-1], k_1, k_2-1) & \text{if } S_1[i+1, n_1] \approx_{0,1} S_2[j+1, n_2], \\ 2 \dot{+} l' \dot{+} l'' \text{ (defined below) otherwise.} \end{cases}$$

Computing l' , we credit the two deletions that will certainly be needed when computing l'' . Depending on the length of $S_1[i+1, n_1]$ and $S_2[j+1, n_2]$, we have to decide which parameter to decrease: If $|S_1[i+1, n_1]| > |S_2[j+1, n_2]|$, we will certainly need at least two deletions in $S_1[i+1, n_1]$, and can start the recursive call with parameter $k_1 - 2$ (and, analogously, with $k_2 - 2$ if $|S_1[i+1, n_1]| < |S_2[j+1, n_2]|$ and both $k_1 - 1$ and $k_2 - 1$ if $S_1[i+1, n_1]$ and $S_2[j+1, n_2]$ are of same length):

$$l' := \begin{cases} \text{LAPCS}(S_1[2, i-1], S_2[2, j-1], k_1-2, k_2) & \text{if } n_1 - i > n_2 - j, \\ \text{LAPCS}(S_1[2, i-1], S_2[2, j-1], k_1, k_2-2) & \text{if } n_1 - i < n_2 - j, \\ \text{LAPCS}(S_1[2, i-1], S_2[2, j-1], k_1-1, k_2-1) & \text{if } n_1 - i = n_2 - j. \end{cases}$$

Computing l'' , we decrease k_1 and k_2 by the deletions already spent when computing l' , where $k'_{1,1} := i - 2 - l'$ denotes the number of deletions spent in $S_1[1, i]$ and $k'_{2,1} := j - 2 - l'$ denotes the number of deletions spent in $S_2[1, j]$:

$$l'' := \text{LAPCS}(S_1[i+1, n_1], S_2[j+1, n_2], k_1 - k'_{1,1}, k_2 - k'_{2,1}).$$

Correctness of Algorithm LAPCS. To show the correctness, we have to make sure that, if an lapcs with the specified properties exists, then the algorithm finds one; the reverse can be seen by checking, for every case of the above algorithm, that we only make matches when they extend the lapcs and that the bookkeeping of the “mismatch counters” k_1 and k_2 is correct. In the following, we omit the details for the easier cases of our search tree algorithm and, instead, focus on the most involved situation, Case (2.5).

In Case (2.5), $S_1[1] = S_2[1]$, there is an arc $(1, i)$ in S_1 and an arc $(1, j)$ in S_2 , and $S_1[i] = S_2[j]$. In Cases (2.5.1) and (2.5.2), we handled the special situation that $S_1[1, i] = S_2[1, j]$ or that $S_1[i+1, n_1] = S_2[j+1, n_2]$. Observe that, if we decide to match the arcs (Case (2.5.3)), we can divide the current instance into two subinstances: bases from $S_1[2, i-1]$ can only be matched to bases from $S_2[2, j-1]$ and bases from $S_1[i+1, n_1]$ can only be matched to bases from $S_2[j+1, n_2]$. We will, in the following, denote the subinstance given by $S_1[2, i-1]$ and $S_2[2, j-1]$ as part 1 of the instance and the one given by $S_1[i+1, n_1]$ and $S_2[j+1, n_2]$ as part 2 of the instance. We have the choice of breaking at least one of the arcs $(1, i)$ and $(1, j)$ or to match them.

We distinguish two cases. Firstly, suppose that we want to break at least one arc. This can be achieved by either deleting $S_1[1]$ or $S_2[1]$. If we do not delete either of

these bases, we obtain a base match. But, in addition, we must delete both $S_1[i]$ and $S_2[j]$, since otherwise we cannot maintain the arc-preserving property. Secondly, we can match the arcs $(1, i)$ and $(1, j)$. Then, we know, since neither Case (2.5.1) nor (2.5.2) applies, that an optimal solution will require at least one deletion in part 1 and will also require at least one deletion in part 2. We can further compute, in linear time, whether part 1 (or part 2, resp.) can be handled by exactly one deletion and start the algorithm recursively only on part 2 (part 1, resp.), decreasing one of k_1 or k_2 by the deletion already spent. In the remaining case, we start the algorithm recursively first on part 1 (to compute l') and, then, on part 2 (to compute l''). At this point we know, however, that an optimal solution will require at least two deletions in part 1 and will also require at least two deletions in part 2. Thus, when starting the algorithm on part 1, we can “spare” two of the $k_1 + k_2$ deletions for part 2, depending on part 2 (as outlined above). Having, thus, found an optimal solution of length l' for part 1, the number of allowed deletions remaining for part 2 is determined: we have, in part 1, already spent $k'_{1,1} := i - 2 - l'$ deletions in $S_1[2, i - 1]$ and $k'_{2,1} := j - 2 - l'$ deletions in $S_2[2, j - 1]$. Thus, there remain, for part 2, $k_1 - k'_{1,1}$ deletions for $S_1[i + 1, n_1]$ and $k_2 - k'_{2,1}$ deletions for $S_2[j + 1, n_2]$. This discussion showed that, in Case (2.5.3), our case distinction covers all subcases in which we can find an optimal solution and, hence, Case (2.5.3) is correct.

Running time of Algorithm LAPCS.

Lemma 1. *Given two arc-annotated sequences S_1 and S_2 , suppose that we have to delete k'_1 symbols in S_1 and k'_2 symbols in S_2 in order to obtain an lapcs.³ Then, the search tree size (i.e., the number of the nodes in the search tree) for a call $LAPCS(S_1, S_2, k'_1, k'_2)$ is upperbounded by $3.31^{k'_1+k'_2}$.*

Proof. Algorithm LAPCS constructs a search tree. In each of the given cases, we do a branching and we perform a recursive call of LAPCS with a smaller value of the sum of the parameters in each of the branches. We now discuss some cases which, in some sense, have a special branching structure. Firstly, Cases (2.1), (2.5.1), and (2.5.2) do not cause any branching of the recursion. Secondly, for Case (2.5.3), in total, we perform five recursive calls. For the following running time analysis, the last two recursive calls of this case (i.e., the ones needed to evaluate l' and l'') will be treated together. More precisely, we treat Case (2.5.3) as if it were a branching into four subcases, where, in each of the first three branches we have *one* recursive call and in the fourth branch we have *two* recursive calls.

In a search tree produced by Algorithm LAPCS, every search tree node corresponds to one of the cases mentioned in the algorithm. Let m be the number of nodes corresponding to Case (2.5.3) that appear in such a search tree. We prove the claim on the search tree size by induction on the number m .

³ Note that there might be several lapcs for two given sequences S_1 and S_2 . The length ℓ of such an lapcs, however, is uniquely defined. Since, clearly, $k'_1 = |S_1| - \ell$ and $k'_2 = |S_2| - \ell$, the values k'_1 and k'_2 also are uniquely defined for given S_1 and S_2 .

For $m = 0$, we do not have to deal with Case (2.5.3). Hence, we can determine the search tree size by the so-called branching vectors (for details of this type of analysis we refer to [14]): Suppose that in one search tree node with current sequences S_1 , S_2 and parameters k'_1 , k'_2 , we have q branches. We analyze the size of the search tree in terms of the sum $k' := k'_1 + k'_2$ of the two parameters. Suppose that in branch t , $1 \leq t \leq q$, we call LAPCS recursively with new parameter values $k'_{1,t}$ and $k'_{2,t}$, i.e., with a sum $k'(t) := k'_{1,t} + k'_{2,t}$. Letting $p_t := k' - k'(t)$, $1 \leq t \leq q$, the vector (p_1, \dots, p_q) is called the *branching vector* for this branch. It corresponds to the recurrence

$$T_{k'} = T_{k'-p_1} + \dots + T_{k'-p_q},$$

where T_i denotes the size of the search tree for parameter value i . The characteristic polynomial of this recurrence is

$$z^p - z^{p-p_1} - \dots - z^{p-p_q}, \tag{1}$$

where $p = \max\{p_1, \dots, p_q\}$. If c is a root of (1) with maximum absolute value, then $T_{k'}$ is $c^{k'}$ up to a polynomial factor and c is called the *branching number* that corresponds to the branching vector (p_1, \dots, p_q) . Moreover, if c is a single root, then even $T_{k'} = O(c^{k'})$.⁴ The branching vectors which appear in our search tree are (1, 1) (Case 1), (1, 1, 1) (Cases 2.2, 2.3), (1, 1, 2) (Case 2.4), (1, 1, 2) (Case 2.5.3 with $m = 0$). The worst-case branching number c for these branching vectors is given for (1, 1, 1) with $c = 3 \leq 3.31$.

Now suppose that the claim is true for all values $m' \leq m - 1$. In order to prove the claim for m , we have to, for a given search tree, analyze a search tree node corresponding to Case (2.5). Suppose that the current sequences in this node are S_1 and S_2 with lengths n_1 and n_2 and that the optimal parameter values are k'_1 and k'_2 . Our goal is to show that the branching of the recursion for Case (2.5.3) has branching vector (1, 1, 2, 1) which corresponds to a branching number $c = 3.31$ (which can be easily verified using the corresponding recurrence). As discussed above, for the first three branches of Case (2.5.3), we only need one recursive call of the algorithm. The fourth branch is more involved. We will have a closer look at this fourth subcase of (2.5.3) in the following. Let us evaluate the search tree size for a call of this fourth subcase. It is clear that the optimal parameter values for the subsequences $S_1[2, i - 1]$ and $S_2[2, j - 1]$ are $k'_{1,1} = (i - 2) - l'$ and $k'_{2,1} = (j - 2) - l'$. Moreover, the optimal parameter values for the subsequences $S_1[i + 1, n_1]$ and $S_2[j + 1, n_2]$ are $k'_{1,2} = (n_1 - i) - l''$ and $k'_{2,2} = (n_2 - j) - l''$. Since, by Cases (2.5.1) and (2.5.2) and by the first four cases in the fourth branch of Case (2.5.3), the cases where $k'_{1,1} + k'_{2,1} \leq 1$ or $k'_{1,2} + k'_{2,2} \leq 1$ are already considered, we may assume that we have $k'_{1,1} + k'_{2,1}, k'_{1,2} + k'_{2,2} \geq 2$. Hence, by induction hypothesis, the search tree size for the computation of l' is $3.31^{k'_{1,1} + k'_{2,1}}$, and the computation of l'' needs a search tree of size $3.31^{k'_{1,2} + k'_{2,2}}$.

⁴ For the branching vectors that appear in our setting, c is always real and will always be a single root.

This means that the total search tree size for this fourth subcase is upperbounded by

$$3.31^{k'_{1,1}+k'_{2,1}} + 3.31^{k'_{1,2}+k'_{2,2}}. \quad (2)$$

Note that, since k'_t is assumed to be the optimal value, we have

$$k'_t = n_t - l' - l'' - 2 \quad \text{for } t = 1, 2,$$

and, hence, an easy computation shows that

$$k'_{t,1} + k'_{t,2} = k'_t \quad \text{for } t = 1, 2.$$

From this we conclude that,

$$3.31^{k'_{1,1}+k'_{2,1}} + 3.31^{k'_{1,2}+k'_{2,2}} \leq 3.31^{k'_1+k'_2-1}. \quad (3)$$

Inequality (3) holds true since, by assumption, $k'_{1,1}+k'_{2,1}, k'_{1,2}+k'_{2,2} \geq 2$. Plugging Inequality (3) into Expression (2), we see that the search tree size for this fourth case of (2.5.3) is upperbounded by $3.31^{k'_1+k'_2-1}$. Besides, by induction hypothesis, the search trees for the first and the second branch of Case (2.5.3) also have size upperbounded by $3.31^{k'_1+k'_2-1}$ and the search tree for the third branch of Case (2.5.3) has size upperbounded by $3.31^{k'_1+k'_2-2}$.

Hence, the overall computations for Case (2.5.3) can be treated with branching vector $(1, 1, 2, 1)$. The corresponding branching number c of this branching vector is 3.31 (being the root of (1) with maximal absolute value for the branching vector $(1, 1, 2, 1)$). This is the worst-case branching number among all branchings. Hence, the full search tree has size $3.31^{k'_1+k'_2}$. \square

Now, suppose that we run algorithm LAPCS with sequences S_1, S_2 and parameters k_1, k_2 . As before let k'_1 and k'_2 be the number of deletions in S_1 and S_2 needed to find an lapcs. As pointed out at the beginning of this section, the search tree will be traversed in breadth-first manner. Hence, on the one hand, we may stop the computation if at some search tree node an lapcs is found (even though the current parameters at this node may be non-zero). On the other hand, if it is not possible to find an lapcs with k_1 and k_2 deletions, then the algorithm terminates automatically by Case (0). Observe that the time needed in each search tree node is upperbounded by $O(n)$ if both sequences S_1 and S_2 have length at most n . This gives a total running time of $O(3.31^{k_1+k_2} \cdot n)$ for the algorithm. The following theorem summarizes the results of this section.

Theorem 1 *The problem LAPCS(NESTED, NESTED) for two sequences S_1 and S_2 with $|S_1|, |S_2| \leq n$ can be solved in time $O(3.31^{k_1+k_2} \cdot n)$ where k_1 and k_2 are the number of deletions needed in S_1 and S_2 . \square*

4 Algorithms for Restricted Versions

In this section, we investigate the so-called c -FRAGMENT LAPCS(CROSSING, CROSSING) and the c -DIAGONAL LAPCS(CROSSING, CROSSING) problems.

In the setting of c -FRAGMENT, the sequences S_1 and S_2 are divided into fragments $S_1^{(1)}, \dots, S_1^{(\lceil |S_1|/c \rceil)}$ and $S_2^{(1)}, \dots, S_2^{(\lceil |S_2|/c \rceil)}$, respectively, each fragment of lengths exactly c (the last fragment may have length less than c) and we do not allow matches between distinct fragments, i.e., between bases in $S_1^{(t)}$ and $S_2^{(t')}$ with $t \neq t'$.

In the setting of c -DIAGONAL, a base $S_1[i]$ ($1 \leq i \leq |S_1|$) is only allowed to match bases in the range $S_2[i - c, i + c]$.

We give algorithms for these problems when parameterized by the length l of the desired subsequence. The versions c -DIAGONAL and c -FRAGMENT of LAPCS (NESTED, NESTED) were already treated by Lin *et al.* [17]. They gave PTAS's for these problems. The running times for the following algorithms are based on worst-case analysis. The algorithms are expected to perform much better in practice.

The problem c -FRAGMENT LAPCS(CROSSING, CROSSING). The algorithm presented here extends an idea which was used for the 1-FRAGMENT case by Lin *et al.* [17]. We translate an instance of the c -FRAGMENT LAPCS problem into an instance of an INDEPENDENT SET problem on a graph $G = (V, E)$ of bounded degree. Since INDEPENDENT SET on graphs of bounded degree is fixed-parameter tractable, we also obtain a fixed-parameter algorithm for c -FRAGMENT LAPCS. The following lemma uses a straightforward search tree algorithm.

Lemma 2. *Let G be a graph of degree bounded by B . Then, an independent set of size k can be found in time $O((B + 1)^k B + |G|)$. \square*

The graph G which is obtained when translating an instance of the c -FRAGMENT LAPCS(CROSSING, CROSSING) problem has degree bounded by $B = c^2 + 2c - 1$, which gives us the following result.

Proposition 1 *The c -FRAGMENT LAPCS(CROSSING, CROSSING) problem parameterized by the length l of the desired subsequence can be solved in time $O((B + 1)^l B + c^3 n)$, where $B = c^2 + 2c - 1$.*

Proof. Let (S_1, A_1) and (S_2, A_2) be an instance of c -FRAGMENT LAPCS (CROSSING, CROSSING), where S_1 and S_2 are over a fixed alphabet Σ .⁵

We construct a graph $G = (V, E)$ as follows. The set of vertices of G corresponds to all possible matches, i.e., we define

$$V := \{v_{i,j} \mid S_1[i] = S_2[j] \text{ and } \lceil i/c \rceil = \lceil j/c \rceil\}.$$

Note that for the c -FRAGMENT problem we are only allowed to match two symbols which are in the same fragment.

Since we want to translate the LAPCS instance into an instance of an INDEPENDENT SET problem on G , the edges of G will represent all conflicting

⁵ We assume that both sequences have the same length, i.e., $n = |S_1| = |S_2|$. If the sequences do not have the same length, we extend the shorter sequence by adding a sequence of a letter not in the alphabet at its end.

matches. Since such a conflict may arise from three different situations, we let $E := E_1 \cup E_2 \cup E_3$, where

$$E_1 := \{\{v_{i_1, j_1}, v_{i_2, j_2}\} \mid (i_1 \neq i_2 \wedge j_1 = j_2) \vee (i_1 = i_2 \wedge j_1 \neq j_2)\} \quad (4)$$

(the two matches represented by v_{i_1, j_1} and v_{i_2, j_2} share a common endpoint),

$$E_2 := \{\{v_{i_1, j_1}, v_{i_2, j_2}\} \mid ((i_1 < i_2) \wedge (j_1 > j_2)) \vee ((i_1 > i_2) \wedge (j_1 < j_2))\} \quad (5)$$

(the two matches represented by v_{i_1, j_1} and v_{i_2, j_2} are not order-preserving), and

$$E_3 := \left\{ \{v_{i_1, j_1}, v_{i_2, j_2}\} \mid \begin{array}{l} ((i_1, i_2) \in A_1 \wedge (j_1, j_2) \notin A_2) \vee \\ ((i_1, i_2) \notin A_1 \wedge (j_1, j_2) \in A_2) \end{array} \right\} \quad (6)$$

(the two matches represented by v_{i_1, j_1} and v_{i_2, j_2} are not arc-preserving).

By construction, it is clear that all lapcs's of length l which match positions $Q_1 = \{i_1, \dots, i_l\}$ in S_1 to positions $Q_2 = \{j_1, \dots, j_l\}$ in S_2 one-to-one correspond to independent sets of the form $V' = \{v_{i_t, j_t} \mid t = 1, \dots, l\}$ in the graph G . We then use the algorithm from Lemma 2 to determine an independent set in G of size l and, hence, an lapcs of length l for $(S_1, A_1), (S_2, A_2)$.

For the running time analysis of this algorithm, note that there can be up to c^2 vertices in G for each fragment. Hence, we can have a total of cn vertices.

Each vertex $v_{i,j}$ in G can have at most $c^2 + 2c - 1$ adjacent edges:

- If a base match $\langle i, j_1 \rangle$ shares with another base match $\langle i, j_2 \rangle$ the same base $S_1[i]$, then an edge must be imposed between the vertices v_{i, j_1} and v_{i, j_2} . There can be at most $c - 1$ such base matches, which share $S_1[i]$ with $\langle i, j \rangle$, and at most $c - 1$ base matches, which share $S_2[j]$ with $\langle i, j \rangle$. Thus, $v_{i, j}$ can have at most $2(c - 1)$ adjacent edges from the set E_1 .
- If $S_1[i]$ is the first base in one fragment of S_1 and $S_2[j]$ is the last base in the same fragment of S_2 , then the base match $\langle i, j \rangle$ can violate the order of the original sequences with at most $(c - 1)^2$ other base matches. Thus, at most $(c - 1)^2$ edges from E_2 will be imposed on vertex $v_{i, j}$.
- If $S_1[i]$ and $S_2[j]$ both are endpoints of arcs (i, i') and (j, j') , then all base matches involving $S_1[i']$ or $S_2[j']$ (but not both) with base match $\langle i, j \rangle$ cannot be arc-preserving. Since $S_1[i']$ and $S_2[j']$ can be in two different fragments and each of them has at most c matched bases, the edges from the set E_3 adjacent to vertex $v_{i, j}$ can amount to $2c$.

Thus, the resulting graph G has a vertex degree bounded by $B = c^2 + 2c - 1$. According to Lemma 2, we can find an independent set in G of size l in time $O((B + 1)^l B + |G|)$. Moreover, since we have cn vertices, G can have at most $O(c^3 n)$ edges. The construction of G can be carried out in time $O(c^3 n)$. Hence, the c -FRAGMENT LAPCS(CROSSING, CROSSING) problem can be solved in time $O((B + 1)^l B + c^3 n)$, where $B = c^2 + 2c - 1$. \square

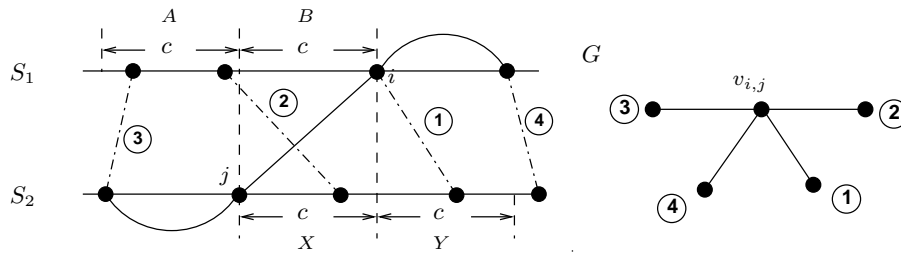


Fig. 1. c -diagonal LAPCS. Vertex $v_{i,j}$ is created for the base match $\langle i, j \rangle$. The lines 1, 2, 3, and 4 represent other four base matches, each of them corresponds to a vertex in G . Since the base match (1) shares base $S_1[i]$ with $\langle i, j \rangle$, an edge is imposed between vertices $v_{i,j}$ and vertex (1). Base matches (2) and $\langle i, j \rangle$ cross each other. Hence, an edge is also imposed between their corresponding vertices. It is clear that neither the pair of base matches (3) and $\langle i, j \rangle$ nor (4) and $\langle i, j \rangle$ is arc-preserving. Two edges are added to the graph G . Note that $A, B, X,$ and Y are all substrings of length c .

The problem c -DIAGONAL LAPCS(CROSSING, CROSSING). A similar approach as the one for c -FRAGMENT LAPCS(CROSSING, CROSSING) can be used to obtain a result for the c -DIAGONAL case. It can be shown that, in this case, that graph G for which we have to find an independent set has degree at most $B = 2c^2 + 7c + 2$.

Proposition 2 *The c -DIAGONAL LAPCS(CROSSING, CROSSING) problem parameterized by the length l of the desired subsequence can be solved in time $O((B + 1)^l B + c^3 n)$, where $B = 2c^2 + 7c + 2$.*

Proof. Again, we translate the problem into an INDEPENDENT SET problem on a graph $G = (V, E)$ with bounded degree and use Lemma 2. For given arc-annotated sequences $(S_1, A_1), (S_2, A_2)$, the set of vertices now becomes

$$V := \{v_{i,j} \mid S_1[i] = S_2[j] \text{ and } j \in [i - c, i + c]\},$$

since each position i in sequence S_1 can only be matched to positions $j \in [i - c, i + c]$ of S_2 . The definition of the edge set E can be adapted from the case for c -FRAGMENT (cf. Equations (4), (5), and (6)). We put an edge $\{v_{i_1, j_1}, v_{i_2, j_2}\}$ iff the corresponding matches $\langle i_1, j_1 \rangle$ and $\langle i_2, j_2 \rangle$ (1) share a common endpoint, (2) are not order-preserving, or (3) are not arc-preserving. Fig. 1 illustrates this construction. Obviously, $|V| \leq (2c + 1) \cdot n$. In the following, we argue that the degree of $G = (V, E)$ is upper-bounded by $B = 2c^2 + 7c + 2$:

- Because a base can be matched to at most $2c + 1$ bases in another sequence, a base match can have common bases with up to $2c + 2c = 4c$ other base matches. E.g., in Fig. 1, base match $\langle i, j \rangle$ shares base $S_1[i]$ with the base match denoted by (1).
- We can observe in Fig. 1 that a vertex in G has a maximum number of edges from E_2 , if the distance between the bases involved in its corresponding base

match is equal to c . Consider, e.g., base match $\langle i, j \rangle$ in Fig. 1. There, a base match crossing $\langle i, j \rangle$ must be from one of the following sets: $M_1 = \{ \langle i_1, j_1 \rangle \mid S_1[i_1] \text{ is in substring } B, S_2[j_1] \text{ is in substring } X \}$, $M_2 = \{ \langle i_2, j_2 \rangle \mid S_1[i_2] \text{ is in substring } B, S_2[j_2] \text{ is in substring } Y, \text{ and } j_2 - i_2 \leq c \}$, or $M_3 = \{ \langle i_3, j_3 \rangle \mid S_1[i_3] \text{ is in substring } A, S_2[j_3] \text{ is in substring } X, \text{ and } j_3 - i_3 \leq c \}$. The set M_1 can have at most c^2 elements. The number of elements of the other two sets can amount to $c^2 - c$. Therefore, each vertex in V can have at most $2c^2 - c$ edges which are imposed to guarantee the order-preserving property.

- If the two bases, which form a base match, both are endpoints of two arcs, like the base match $\langle i, j \rangle$ in Fig. 1, then this base match cannot be in an arc-preserving subsequence with base matches, which involve only one of the other endpoints of the arcs. Two such base matches are marked in Fig. 1 with (3) and (4). Those base matches can amount to $4c + 2$.

Consequently, the graph G has degree bounded by $B = 2c^2 + 7c + 2$. With $(2c+1)n$ vertices, G has at most $O(c^3n)$ edges. The construction of G can be done in time $O(c^3n)$. Hence, the c -DIAGONAL LAPCS(CROSSING, CROSSING) problem can be solved in time $O((B + 1)^l B + c^2n)$, where $B = 2c^2 + 7c + 2$. \square

The problems c -FRAGMENT (c -DIAGONAL) LAPCS(UNLIMITED, UNLIMITED).

Note that the observation that the graph $G = (V, E)$ above has bounded degree heavily depends on the fact that the two underlying sequences have CROSSING arc structure. Hence, the same method does not directly apply for c -FRAGMENTED (c -DIAGONAL) LAPCS(UNLIMITED, UNLIMITED). However, if the “degree of a sequence” is bounded, we can upperbound the degree of G . The *degree* of an arc-annotated sequence (S, A) with UNLIMITED arc structure is the maximum number of arcs from A that are incident to a base in S . Clearly, the so-called *cutwidth* of an arc-annotated sequence (see [6]) is an upper bound on the degree.

Proposition 3 *The c -FRAGMENT (and c -DIAGONAL, respectively) LAPCS (UNLIMITED, UNLIMITED) problem with bounded degree b for its sequences, parameterized by the length l of the desired subsequence, can be solved in time $O((B + 1)^l B + c^3n)$ with $B = c^2 + 2bc - 1$ (and in time $O((B' + 1)^l B' + c^3n)$ with $B' = 2c^2 + (4b + c)c + 2b$, respectively).*

Proof. The c -FRAGMENT (c -DIAGONAL) LAPCS(UNLIMITED, UNLIMITED) problem can also be translated into an INDEPENDENT SET problem on a graph $G = (V, E)$ with bounded degree using a similar construction as shown above. The number of vertices in the resulting graph is equal to the one obtained in the proofs of Proposition 1 and 2, but the bound on the degree changes. In the constructions in the proofs, we added three sets of edges to G . Since the first two sets have nothing to do with arcs, these edges remain in the graph for UNLIMITED arc structure. In the constructions above, $2c$ edges for c -FRAGMENT and $4c + 2$ edges for c -DIAGONAL are added into E_3 for a base match $\langle i, j \rangle$ with two arc endpoints, $(i, i_1) \in A_1$ and $(j, j_1) \in A_2$. These edges are between vertex $v_{i,j}$ and the vertices, which correspond to the base matches involving one of $S_1[i_1]$ and $S_2[j_1]$. In

UNLIMITED arc structure with bounded degree b , a base $S_1[i]$ can be endpoint of at most b arcs, we denote them by $(i, i_1), (i, i_2), \dots, (i, i_b)$. The third set of edges must be extended to include the edges between (i, j) and all vertices, which correspond to base matches involving one of $S_1[2], \dots, S_1[b], S_2[2], \dots, S_2[b]$. The amount of edges in this set can increase to $b(2c)$ for c -FRAGMENT and to $b(4c + 2)$ for c -DIAGONAL LAPCS(UNLIMITED, UNLIMITED). The degree of the resulting graph for c -FRAGMENT is then bounded by $B = c^2 + 2bc - 1$, and the one for c -DIAGONAL by $B = 2c^2 + (4b + 3)c + 2b$. \square

5 Conclusion and Future Work

Adopting a parameterized point of view [1, 5, 9], we have shed new light on the algorithmic tractability of the NP-complete LONGEST COMMON SUBSEQUENCE problem with nested arc annotations, an important problem in biologically motivated structure comparison. Immediate questions arising from our work are to further improve the exponential terms of the running times of our exact algorithms and to determine the parameterized complexity of (non-restricted) LAPCS(NESTED,NESTED) when parameterized by subsequence length instead of number of deletions. Depending on what (relative) length of the longest common subsequence is to be expected, one or the other parameterization might be more appropriate. In the case of a “short” lcs, the subsequence length is more useful as a parameter, and in the case of “long” lcs, the number of deletions is the preferable parameter. Our complexity analyses are worst-case, however, and it is a topic of future investigations to study the practical usefulness of our algorithms by implementations and experiments. In this context, it is also meaningful to take a closer look at the (special case) problem “APS(NESTED,NESTED),” where one asks whether a given sequence forms an arc-preserving subsequence of another. Results in [13] make us confident that this is doable in polynomial time, but, if so, to get the degree of the polynomial as small as possible remains a research issue of theoretical as well as practical importance, see [11] for more details.

References

1. J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229: 3–27, 2001.
2. H. L. Bodlaender, R. G. Downey, M. R. Fellows, M. T. Hallett, and H. T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11: 49–57, 1995.
3. H. L. Bodlaender, R. G. Downey, M. R. Fellows, and H. T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147:31–54, 1995.
4. P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110:13–24, 2001.
5. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer. 1999.

6. P. A. Evans. *Algorithms and Complexity for Annotated Sequence Analysis*. PhD thesis, University of Victoria, Canada. 1999.
7. P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proc. of 10th CPM*, number 1645 in LNCS, pages 270–280, 1999. Springer.
8. P. A. Evans and H. T. Wareham. Exact algorithms for computing pairwise alignments and 3-medians from structure-annotated sequences. In *Proc. of Pacific Symposium on Biocomputing*, pages 559–570, 2001.
9. M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proc. of 12th ISAAC*, number 2223 in LNCS, pages 291–307, 2001. Springer.
10. D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proc. of 40th IEEE FOCS*, pages 512–521, 1999.
11. J. Guo. *Exact Algorithms for the Longest Common Subsequence Problem for Arc-Annotated Sequences*. Diploma thesis, Universität Tübingen, Fed. Rep. of Germany. February 2002.
12. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press. 1997.
13. T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In *Proc. of 11th CPM*, number 1848 in LNCS, pages 154–165, 2000. Springer. Full paper accepted by *Journal of Discrete Algorithms*.
14. O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223: 1–72, 1999.
15. G. Lancia, R. Carr, B. Walenz, and S. Istrail. 101 optimal PDB structure alignments: a branch-and-cut algorithm for the maximum contact map overlap problem. In *Proc. of 5th ACM RECOMB*, pages 193–202, 2001.
16. M. Li, B. Ma, and L. Wang. Near optimal multiple alignment within a band in polynomial time. In *Proc. of 32nd ACM STOC*, pages 425–434, 2000.
17. G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. In *Proc. of 28th ICALP*, number 2076 in LNCS, pages 444–455, 2001. Springer.
18. M. Paterson and V. Dancik. Longest common subsequences. In *Proc. of 19th MFCS*, number 841 in LNCS, pages 127–142, 1994. Springer.
19. D. Sankoff and J. Kruskal (eds.). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley. 1983. Reprinted in 1999 by CSLI Publications.