Computing the Similarity of Two Sequences with Nested Arc Annotations [1]

Jochen Alber [2] Jens Gramm [3] Jiong Guo [4] Rolf Niedermeier [4]

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, one of our algorithms determines (if existent) in $O(3.31^{k_1+k_2} \cdot n)$ time 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. A second algorithm shows that (in case of a four letter alphabet) we can find a length l arc-annotated subsequence in $O(12^l \cdot l \cdot n)$ time. This means that the problem is fixed-parameter tractable when parameterized by the number of deletions as well as when parameterized by the subsequence length. Our findings complement 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.

Key words: arc-annotated sequences, RNA secondary structure, search tree, NP-completeness, fixed-parameter tractability, longest common subsequence.

[1] A preliminary version of this paper titled "Towards optimally solving the Longest Common Subsequence problem for sequences with nested arc annotations in linear time" was presented at the *13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002),* Springer-Verlag, LNCS 2373, pages 99–114, held in Fukuoka, Japan, July 2002.
[2] Supported by the Deutsche Forschungsgemeinschaft (DFG), research project PEAL (parameterized complexity and exact algorithms), NI 369/1.
[3] Supported by the Deutsche Forschungsgemeinschaft (DFG), research project OPAL (optimal solutions for hard problems in computational biology), NI 369/2.
[4] Supported by the Deutsche Forschungsgemeinschaft (DFG), junior research group PIAF (fixed-parameter algorithms), NI 369/4.

1 Introduction

Basic motivation. 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., [5,7,16,26,28]. 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 [8–10,18,24]. It is a sound and meaningful mathematical formalization of comparing the secondary structures of molecular sequences. A similar but more restrictive model was studied by Ma *et al.* [25].

Problem description. For a sequence S, an *arc annotation* A of S is a set of unordered pairs of positions 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 $k_1 := |S_1| - |T|$ 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 $k_2 := |S_2| - |T|$ letters from S_2, then T and the resulting sequences are the same and also their arc annotations coincide. In this paper, we mainly restrict ourselves to sequences with *nested* arc annotations, where one requires that no two arcs share an endpoint and no two arcs cross each other. This variant of the problem is referred to as LAPCS(nested,nested). According to Lin *et al.* [24], LAPCS for nested arc annotations is "generally thought of as the most important variant of the LAPCS problem." An example of a longest common subsequence for two sequences with nested arc annotations is given in Fig. 1.

Previous results. Whereas LCS for two sequences without arc annotations is easily solvable in quadratic time (it becomes NP-complete when allowing for an arbitrary number of input sequences), LAPCS for two sequences is NP-complete [8,9]. Answering an open question of Evans [8], Lin *et al.* [24] showed that already LAPCS(nested,nested) is NP-complete. Observe that the corresponding problem in the easier model of Ma *et al.* [25] is solvable in polynomial time. In addition, Lin *et al.* gave polynomial time approximation schemes (PTAS) for some (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) with respect to matching position numbers (refer to Section 2 for details). As to the general LAPCS(nested,nested) problem, Jiang *et al.* [18] gave a quadratic time factor-2-approximation algorithm. For related studies concerning algo-

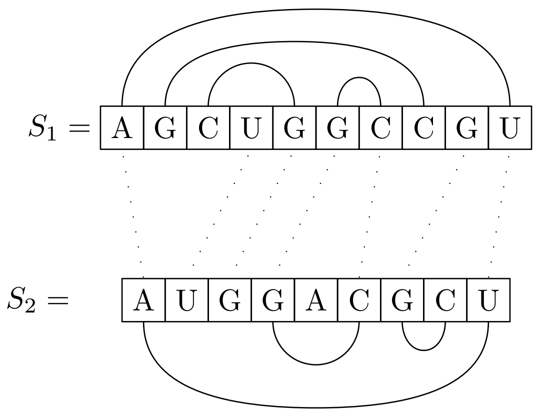

Fig. 1. Example of a longest arc-preserving common subsequence for two arc-annotated input sequences S_1 and S_2. The common subsequence is obtained by deleting three bases in S_1 and two bases in S_2

rithmic aspects of (protein) structure comparison using "contact maps," refer to [12,22].

New results. Our main results are two exact algorithms, i.e., providing *optimal* solutions, for LAPCS(NESTED,NESTED): one that runs in linear time when we only need a constant number of base deletions to obtain the common subsequence from the input sequences, and one that runs in linear time when the length of the common subsequence is constant.

More precisely, we give an exact algorithm that solves LAPCS(NESTED,NESTED) in $O(3.31^{k_1+k_2} \cdot n)$ time 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). Our algorithm employs the bounded search tree paradigm of parameterized complexity in a nontrivial way and introduces a novel analysis technique to handle the bottleneck case in our analysis of the search tree sizes; namely, we estimate the sum of sizes for two mutually dependent search trees based on the fact that the sum of their heights is bounded.

Moreover, we give an enumerative algorithm solving LAPCS(NESTED,NESTED) in $O((3 \cdot |\Sigma|)^l \cdot l \cdot n)$ time where l is the length of the common subsequence and $|\Sigma|$ is the size of the underlying alphabet. In both these fixed-parameter algorithms we apply a dynamic programming algorithm [14] that determines, given two arc-annotated sequences S_1 and S_2, in $O(|S_1| \cdot |S_2|)$ time whether S_2 is an arc-preserving subsequence of S_1.

In order to compare our exact algorithms to the approximation algorithms for LAPCS(NESTED,NESTED), observe that the PTAS algorithms (which only work for special cases of LAPCS(NESTED,NESTED)) obtaining a good degree of approximation get prohibitive from a practical point of view due to enormous running times. The factor-2-approximation might be not good enough for biological and other applications. In summary, in terms of parameterized complexity [1,7,11], our results show that LAPCS(nested,nested) is fixed-parameter tractable when parameterized by each of two dual parameters, the number of deletions allowed and, for fixed alphabet size, the length of the common subsequence. This results in a useful pair of algorithms: One is especially practical for the comparison of non-similar sequences, i.e., when the length of the common subsequence is small, and one is particularly suitable in the remaining case, i.e., when the input sequences are similar. Note that our results give also a counter-example to the conjecture proposed by Khot and Raman [20], "... typically parametric dual problems have complimentary parameterized complexity." This conjecture means that if a problem is fixed-parameter tractable with respect to one of two dual parameters then it is supposed to be fixed-parameter intractable with respect to the other.

Finally, we obtain fixed-parameter tractability results for modified versions of LAPCS as studied by Lin *et al.* [24] for arc-annotations which are more general than nested (e.g., crossing).

Structure of the paper. In Section 2, we formally define the problems and introduce the notation used throughout the work. Section 3 contains the enumeration based approach to solve LAPCS(nested,nested), leading to an algorithm which runs in $O((3\cdot|\Sigma|)^l\cdot l\cdot n)$ time. By way of contrast, Section 4 presents the more involved and probably more practical algorithm that solves LAPCS(nested,nested) by a search tree based method with $O(3.31^{k_1+k_2}\cdot n)$ running time. Section 5 contains exact algorithms for modified versions of LAPCS(nested,nested) that were introduced in [24]. We conclude with some open questions in Section 6.

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 [1,6,7,11,27]. 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*, i.e., 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. [5] The decisive point here is that the combinatorial explosion seemingly unavoid-

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

able when 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 [3,4,7,27] for any details.

Evans [8,9] 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.

Definition 1 *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 $\{1, 2, \ldots, n\}$. Each pair (i, j) connects the two* bases *$S[i]$ and $S[j]$ at positions i and j in S by an arc.*

Besides, for $i \leq j \leq n$, we use $S[i, j]$ to denote the substring of the sequence S starting at position i and ending at position j and we use $S[i, +]$ to denote the suffix of S starting at position i.

We focus on two sorts of restrictions of the unlimited arc structures, namely *crossing* and *nested*.

Definition 2 *An arc annotation A of a sequence S is said to have* crossing *structure if no two arcs in A share an endpoint, i.e., if for two distinct arcs $(i_1, i_2), (i_3, i_4)$ from A we have $i_1, i_2 \notin \{i_3, i_4\}$.*

The arc annotation A is said to have nested *structure, if no two arcs share an endpoint and, in addition, no two arcs cross each other which means that for all (i_1, i_2), (i_3, i_4) with $i_1 < i_2$ and $i_3 < i_4$ it holds that $i_3 < i_1 < i_4$ iff $i_3 < i_2 < i_4$.*

Let S_1 and S_2 be two sequences with arc sets A_1 and A_2, respectively, and let $i_1, i_2 \in \{1, \ldots, |S_1|\}$ and $j_1, j_2 \in \{1, \ldots, |S_2|\}$ be positive integers. If $S_1[i_1] = S_2[j_1]$, we refer to this as a *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 an *arc match*.

We now introduce the notion of an arc-preserving subsequence. Note that a common subsequence T of S_1 and S_2 induces a one-to-one mapping M_T from a subset of $\{1, 2, \ldots, |S_1|\}$ to a subset of $\{1, 2, \ldots, |S_2|\}$.

Definition 3 *We say that a common subsequence T of S_1 and S_2 is an* arc-preserving common subsequence *if the arcs induced by M_T are preserved, i.e.,* *for all* $\langle i_{r_1}, j_{r_1}\rangle, \langle i_{r_2}, j_{r_2}\rangle \in M_T$*:*

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

The Longest Arc-Preserving Common Subsequence (LAPCS) problem is defined as follows:

Input: Two arc-annotated sequences S_1 and S_2 over some alphabet Σ.

Task: Find a maximum length arc-preserving subsequence for S_1 and S_2.

We refer to a maximum length arc-preserving subsequence for two sequences as a *longest arc-preserving common subsequence*, which will be abbreviated by *lapcs*. Since LAPCS is NP-complete even for two input sequences [8], here and in the literature attention is focused on this case. Depending on the arc structures allowed in S_1 and S_2, various subproblems of LAPCS can be defined. We focus here on the NP-complete LAPCS(nested,nested) [18,24], where both sequences S_1 and S_2 are required to have nested arc annotations.

Motivated from biological applications [16,23], Lin *et al.* [24] furthermore introduced the following special cases of LAPCS(nested,nested). 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). Similarly, in c-fragment LAPCS(nested,nested) all base matches induced by a longest arc-preserving common subsequence have to be between "corresponding" fragments of size c of S_1 and S_2. Lin *et al.* [24] 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 Enumerative Algorithm for LAPCS(nested,nested)

In this section, we solve LAPCS(nested,nested) by an enumerative approach based on a recent result for pattern matching for arc-annotated sequences [14]. This leads to a fixed-parameter algorithm where the parameter is the length l of the common arc-annotated subsequence. The result reads as follows.

Theorem 1 *The problem* LAPCS(nested,nested) *for two sequences* S_1 *and* S_2 *over alphabet* Σ *with* $|S_1|, |S_2| \leq n$ *can be solved in* $O((3 \cdot |\Sigma|)^l \cdot l \cdot n)$ *time where* l *is the length of the common subsequence searched for.*

PROOF. In a recent paper [14] it was shown that for two sequences S_1 and S_2 with nested arc annotations it can be decided in $O(|S_1| \cdot |S_2|)$ time whether or not S_2 occurs as an arc-preserving subsequence in S_1. That is, this algorithm solves the arc-annotated pattern matching problem in case of nested arc structures in quadratic time. To decide whether there is an arc-preserving common subsequence of length l of the two given arc-annotated sequences S_1 and S_2, the following simple enumerative approach applies.

- Generate all possible length-l sequences with all sorts of nested arc annotations, and
- for each of these arc-annotated candidate sequences, check—using the above mentioned pattern matching algorithm twice—whether or not it occurs as a pattern in both S_1 and S_2.

This algorithm directly leads to a solution of LAPCS(NESTED,NESTED), its correctness being straightforward.

What is the time complexity of this algorithm? To check for a given arc-annotated sequence of length l whether it occurs as a pattern both in S_1 and S_2 can be done in $O(l \cdot n)$ time [14]. Thus, to achieve the claimed overall running time of $O((3 \cdot |\Sigma|)^l \cdot l \cdot n)$ it remains to be shown that there are at most $(3 \cdot |\Sigma|)^l$ of these length-l candidate patterns. Firstly, note that on the "pure sequence level," there are exactly $|\Sigma|^l$ different length-l strings as candidates. Secondly, we need to analyze the more subtle point how many different nested arc structures there are which can be associated with each such string. It is crucial here to observe that there is a one-to-one correspondence between nested arc structures and Dyck languages well-known from formal language theory (cf., e.g., [17]). More precisely, each arc structure corresponds to a Dyck word over a two-letter alphabet, namely opening and closing brackets. The number of ways to parenthesize is exactly given by the Catalan numbers (cf., e.g., [13]). Thus, $D(i)$, (for even i) denoting the number of different Dyck words of length i, can be determined by

$$\binom{i}{i/2}/(i+1) < 2^i.$$

Without loss of generality assume that l is an even number. Then, taking into account that a length-l string may have bases without arcs and that each base may be the starting or end point of at most one arc, we obtain the following expression that upperbounds the number of different arc structures to be associated with one particular length-l string:

$$D(l) + \binom{l}{2} \cdot D(l-2) + \ldots + \binom{l}{l-2} \cdot D(2) + 1.$$

Note that there only can be even numbers of bases without arcs. Using the upper bound 2^i for $D(i)$, the above sum can be upperbounded (also counting non-occurring odd values i for $D(i)$) by

$$\sum_{i=0}^{l} \binom{l}{i} \cdot 2^i \leq (1+2)^l = 3^l.$$ [6]

In total, we thus have $|\Sigma|^l$ possible strings, multiplied with 3^l possible arc structures, which leads to the upper bound of $(3 \cdot |\Sigma|)^l$ candidate patterns to be tested. Clearly, these can be generated in $O((3 \cdot |\Sigma|)^l \cdot l)$ time. □

For fixed alphabet size, Theorem 1 means that LAPCS(NESTED,NESTED) is fixed-parameter tractable with respect to parameter l. In case of RNA with alphabet size four, we obtain:

Corollary 2 *The problem* LAPCS(NESTED,NESTED) *for two RNA sequences of length at most n can be solved in $O(12^l \cdot l \cdot n)$ time, l being the length of the common subsequence searched for.* □

Clearly, the above presented result only leads to an efficient exact algorithm if parameter l (subsequence length) is small. The next section presents an exact algorithm that is preferable in the (practically more relevant) case that the common subsequence is expected to be large. Note, however, that the above given time bounds clearly are worst-case bounds and that by adding further heuristics and making use of special properties of practical problem instances (which might lead us to having to consider significantly less than all theoretically possible candidate patterns) should enable significant accelerations of the above algorithm in applications. A good example of the practical success of such an enumerative approach in computational molecular biology is the recent result of Blanchette *et al.* [2] in the context of motif search for DNA sequences.

4 Search Tree Algorithm for LAPCS(NESTED,NESTED)

In this section, we describe and analyze Algorithm STA [7] which solves the LAPCS(NESTED, NESTED) problem in $O(3.31^{k_1+k_2} \cdot n)$ time, where n is the maximum length of the input sequences, and k_1 and k_2 denote the number of letters that have to be deleted from S_1 and S_2, respectively, in order to obtain

[6] We remark that this analysis is not tight. Using generating functions, it could be somewhat improved. For the sake of simplicity (and because the analysis is purely worst-case anyway) we do not go into the details here.
[7] The name STA stands for search tree algorithm.

a longest arc-preserving common subsequence. 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.

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 $n_2 := |S_2|$.

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. We can check whether $S_1 \approx_{1,0} S_2$ or whether $S_1 \approx_{0,1} S_2$ in linear time, as is sketched in the following. Assume that we want to check whether $S_1 \approx_{1,0} S_2$. If $n_1 < n_2$, this cannot hold and if $n_1 = n_2$ then we only have to test whether S_1 and S_2 are equal. Therefore, we assume w.l.o.g. that $n_1 > n_2$, more precisely that $n_1 = n_2 + 1$ (if $n_1 > n_2 + 1$ then $S_1 \approx_{1,0} S_2$ is not possible). Then, we process S_1 and S_2 from left to right, trying to match $S_1[i]$ with $S_2[i]$ (also taking into account the arcs) for $i = 1, \ldots, n_1$, until we encounter a "mismatch," i.e. a position i_m such that $S_1[i_m] \neq S_2[i_m]$ or $S_1[i_m]$ is endpoint of an arc, but $S_2[i_m]$ is not endpoint of an arc. In case of such a mismatch, we delete $S_1[i_m]$ and continue to match $S_1[i + 1]$ with $S_2[i]$ for $i = i_m, \ldots, n_1$. Only if all these positions match, we have $S_1 \approx_{1,0} S_2$; if, otherwise, there is a second mismatch, then $S_1 \approx_{1,0} S_2$ does not hold. This test can clearly be done in linear time.

In the following subsection, we give a detailed description of Algorithm STA. It processes the input sequences S_1 and S_2 from left to right, deleting already treated bases of the sequences (when deleting a base adjacent arcs are also deleted). By a case distinction depending on the bases at the currently first positions in S_1 and S_2, we decide how to continue recursively. This case distinction also takes into account possible arcs which are adjacent to these bases. Note however, that the first positions in S_1 and S_2 can only be left endpoints of an arc. The most involved case in the case distinction of 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.

4.1 Description of the Algorithm

Recursive Procedure STA(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 but not the... */
        if $k_2 \geq |S_2|$ then return 0 else return $-1$            /* ...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:
   • STA($S_1[2, n_1], S_2, k_1 - 1, k_2$) /* delete $S_1[1]$ */
   • STA($S_1, S_2[2, n_2], k_1, k_2 - 1$) /* delete $S_2[1]$ */.
(Case 2)  /* Matching bases. */
  if $S_1[1] = S_2[1]$ then
    (2.1)  /* No arcs involved $\Longrightarrow$ It is safe to match $S_1[1]$ with $S_2[1]$.*/
      if both $S_1[1]$ and $S_2[1]$ are not endpoints of arcs then return
      $1 \dot{+}$STA($S_1[2, n_1], S_2[2, n_2], k_1, k_2$).

    (2.2)  /* Only an arc in $S_1$. */
      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:
       • STA($S_1[2, n_1], S_2, k_1 - 1, k_2$) /* delete $S_1[1]$ */,
       • STA($S_1, S_2[2, n_2], k_1, k_2 - 1$) /* delete $S_2[1]$ */, and
       • $1 \dot{+}$STA($(S_1 - S_1[i])[2, +], S_2[2, n_2], k_1 - 1, k_2$) /* match */.
      /* Since there is an arc in $S_1$ only, $S_1[1]$ and $S_2[1]$ can be matched only
      if $S_1[i]$ is deleted. */
    (2.3)  /* Only an arc in $S_2$. */
      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:
- STA($S_1[2, n_1], S_2, k_1 - 1, k_2$) */* delete* $S_1[1]$ **/*,
- STA($S_1, S_2[2, n_2], k_1, k_2 - 1$) */* delete* $S_2[1]$ **/*, and
- $1 \dot{+}$STA($(S_1 - S_1[i])[2, +], (S_2 - S_2[j])[2, +], k_1 - 1, k_2 - 1$) */* 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 STA 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**
$i \dot{+}$STA($S_1[i+1, n_1], S_2[j+1, n_2], k_1, k_2$).
(2.5.2) */* Sequences following the arcs match. */*
if $S_1[i+1, n_1] = S_2[j+1, n_2]$ **then return**
$2 \dot{+} (n_1 - i) \dot{+}$STA($S_1[2, i-1], S_2[2, j-1], k_1, k_2$).

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:

- STA($S_1[2, n_1], S_2, k_1 - 1, k_2$) */* delete* $S_1[1]$. **/*,
- STA($S_1, S_2[2, n_2], k_1, k_2 - 1$) */* delete* $S_2[1]$. **/*,
- $1 \dot{+}$STA($(S_1 - S_1[i])[2, +], (S_2 - S_2[j])[2, +], k_1 - 1, k_2 - 1$)
  / 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:

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

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):

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

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 $S_2[1, j]$:

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

4.2 Correctness of Algorithm STA.

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 $S_1[i + 1, n_1] = S_2[j + 1, n_2]$. For both cases, the best choice is to match the arcs. In Case (2.5.3), we have the choice of breaking at least one of the arcs $(1, i)$ and $(1, j)$ or to match them. Observe that, if we decide to match the arcs, 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 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.

4.3 Running time of Algorithm STA.

Lemma 3 *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.*[8] *Then, the search tree size (i.e., the number of the nodes in the search tree) for a call STA(S_1, S_2, k'_1, k'_2) is upperbounded by $3.31^{k'_1+k'_2}$.*

PROOF. Algorithm STA constructs a search tree, where every search tree node corresponds to one of the cases mentioned in the algorithm. We observe that in Cases (2.1), (2.5.1), and (2.5.2), we have a recursive call of STA with smaller remaining sequences, but we do not invoke more than one recursive call, i.e., we do not branch in the search tree. Case (0) is used for the termination. In all other cases, i.e., Case (1), Cases (2.2), (2.3), (2.4), and (2.5.3), we do a branching and we perform a recursive call of STA with a smaller value of the sum of the parameters in each of the branches. In particular, 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.

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.

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 [21]): 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 STA recursively with new

[8] 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, $k'_1 = |S_1| - l$ and $k'_2 = |S_2| - l$, the values k'_1 and k'_2 also are uniquely defined for given S_1 and S_2.

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 $(p_1, \ldots, p_q)$ is called the *branching vector* for this branch. It corresponds to the recurrence

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

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

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

where $p = \max\{p_1, \ldots, p_q\}$. 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 $(p_1, \ldots, p_q)$. Moreover, if c is a single root, then even $T_{k'} = O(c^{k'})$.[9] 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 (2.5.3). 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 the induction hypothesis and since we traverse the search tree in breadth-first manner, 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 $3.31^{k'_{1,2}+k'_{2,2}}$. This means that the total search tree size for this fourth subcase is upper-

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

bounded by

$$3.31^{k'_{1,1}+k'_{2,1}} + 3.31^{k'_{1,2}+k'_{2,2}}. \tag{2}$$

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,

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

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 STA 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 4 *The problem* LAPCS(NESTED, NESTED) *for two sequences S_1 and S_2 with $|S_1|, |S_2| \leq n$ can be solved in $O(3.31^{k_1+k_2} \cdot n)$ time where k_1 and k_2 are the number of deletions needed in S_1 and S_2.* $\square$

5 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. Evans [8] showed that LAPCS(crossing,crossing) appears to be fixed-parameter intractable (more precisely, it is W[1]-hard (cf. [7]) with respect to parameter l (subsequence length)). By way of contrast, we show that the restricted versions as considered by Lin et al. [24] are fixed-parameter tractable.

In the setting of c-fragment, the sequences S_1 and S_2 are divided into fragments $S_1^{(1)}, \ldots, S_1^{(\lceil |S_1|/c \rceil)}$ and $S_2^{(1)}, \ldots, S_2^{(\lceil |S_2|/c \rceil)}$, respectively, each fragment of length 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.* [24]. They gave PTAS's for these problems. The running times for the following exact algorithms are based on worst-case analysis. The algorithms are expected to perform much better in practice.

5.1 c-fragment LAPCS(crossing, crossing)

The algorithm presented here extends an idea which was used for the 1-fragment case by Lin *et al.* [24]. 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 5 *Let G be a graph of degree bounded by B. Then, an independent set of size k can be found in $O((B+1)^k B + |G|)$ time.*

PROOF. We describe how to solve this question using a search tree of height k in which each node of the search tree has at most $(B+1)$ children. We use a representation of the graph allowing to label each vertex, thereby marking it as "done." Initially, in the root of the search tree, all vertices of the graph are unmarked and the independent set is empty. Then, the recursive search tree procedure works as follows. First, we find an unmarked vertex u (if all vertices are marked as done, then we have already found a solution). Note that u has at most B neighbors. Any maximum independent set of G contains either u or at least one of its neighbors. Thus, we arrive at modified instances which are handled recursively: In one case, we put u into the independent set and mark u and all its neighbors as done. In all other cases—one case for every neighbor v of u—we put v into the independent set and mark v and all its neighbors as done. On each of these modified instances, we invoke the search tree procedure recursively. The recursion stops when we find a solution (all vertices are marked as done while the independent set is of size at most k) or when already k vertices are in the independent set while not all vertices are marked as done (no solution in this branch of the search tree). This results in a search tree of size at most $(B+1)^k$.

In each search tree node, we only need linear time: The unmarked vertices in the graph can be administrated in a doubly linked list such that the next unmarked vertex can be found in constant time and, when marking a vertex as done, we can delete it from this list in constant time. Then, marking a vertex and all its neighbors as done needs only $O(B)$ time. These data structures can be initialized in $O(|G|)$ time. Therefore, this algorithm needs $O((B+1)^k B + |G|)$ time. $\square$

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 6 *The c-*fragment *LAPCS*(crossing, crossing) *problem parameterized by the length l of the desired subsequence can be solved in $O((B+1)^l B + c^3 n)$ time, 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. [10]

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 of length c.

[10] 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 a sequence of a letter not in the alphabet at its end.

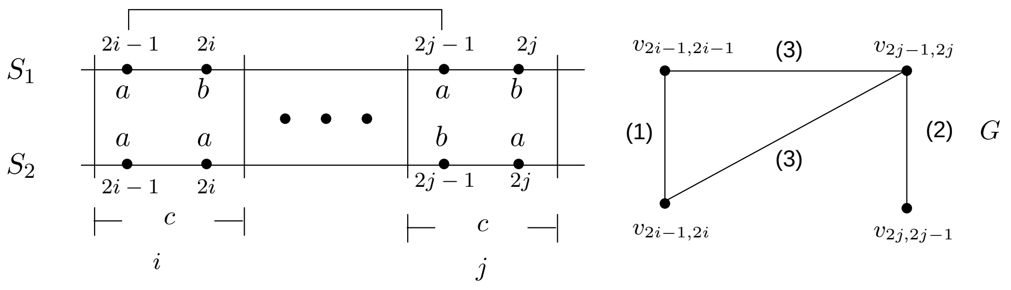

Fig. 2. 2-fragment LAPCS. There are four base matches in the two fragments shown in this figure. They correspond to the four vertices in G. The edge marked by (1) in G is imposed since base matches $\langle 2i-1, 2i-1\rangle$ and $\langle 2i-1, 2i\rangle$ share base $S_1[2i-1]$, i.e., $\{v_{2i-1,2i-1}, v_{2i-1,2i}\} \in E_1$. Since base matches $\langle 2j, 2j-1\rangle$ and $\langle 2j-1, 2j\rangle$ cross each other, their corresponding vertices are joined by an edge marked by (2), i.e., $\{v_{2j-1,2j}, v_{2j,2j-1}\} \in E_2$. While an arc joins bases $S_1[2i-1]$ and $S_1[2j-1]$, there is no arc in S_2 between the ith and the jth fragments. Two edges are imposed between the vertices which correspond to the base matches involving the endpoints of arc $(2i-1, 2j-1)$. These edges are marked with (3), i.e., $\{v_{2i-1,2i-1}, v_{2j-1,2j}\}, \{v_{2i-1,2i}, v_{2j-1,2j}\} \in E_3$

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

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

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

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

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

$$E_3 := \left\{ \{v_{i_1,j_1}, v_{i_2,j_2}\} \;\middle|\; \begin{array}{c} ((i_1,i_2) \in A_1 \wedge (j_1,j_2) \notin A_2) \vee \\ ((i_1,i_2) \notin A_1 \wedge (j_1,j_2) \in A_2) \end{array} \right\} \tag{6}$$

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

Fig. 2 illustrates this construction for 2-fragment LAPCS.

By construction, it is clear that all lapcs's of length l which match positions $Q_1 = \{i_1, \ldots, i_l\}$ in S_1 to positions $Q_2 = \{j_1, \ldots, j_l\}$ in S_2 one-to-one correspond to independent sets of the form $V' = \{v_{i_t,j_t} \mid t = 1, \ldots, l\}$ in the graph G. We then use the algorithm from Lemma 5 to determine an independent set in G of size l and, hence, an lapcs of length l for (S_1, A_1) and (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 $c^2 \cdot \frac{n}{c} = c \cdot n$ vertices in G.

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 5, we can find an independent set in G of size l in $O((B+1)^l B + |G|)$ time. Moreover, since we have $c \cdot n$ vertices, G can have at most $O(c^3 n)$ edges. The construction of G can be carried out in $O(c^3 n)$ time. Hence, the c-fragment LAPCS(crossing, crossing) problem can be solved in $O((B+1)^l B + c^3 n)$ time, where $B = c^2 + 2c - 1$. □

5.2 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, graph G for which we have to find an independent set has degree at most $B = 2c^2 + 7c + 2$.

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

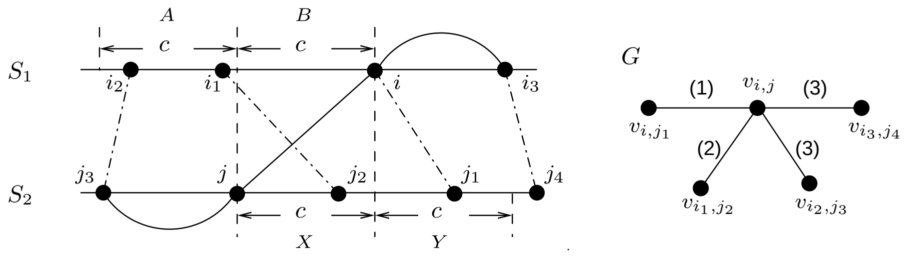

Fig. 3. c-DIAGONAL LAPCS. Vertex $v_{i,j}$ is created for the base match $\langle i, j\rangle$. The other four base matches denoted by dashed lines correspond to the vertices v_{i,j_1}, v_{i_1,j_2}, v_{i_2,j_3}, and v_{i_3,j_4} in G. Since the base match $\langle i, j_1\rangle$ shares base $S_1[i]$ with $\langle i, j\rangle$, an edge marked by (1) is imposed between vertex $v_{i,j}$ and vertex v_{i,j_1}. Base matches $\langle i_1, j_2\rangle$ and $\langle i, j\rangle$ cross each other. Hence, an edge marked by (2) is imposed between their corresponding vertices. It is clear that neither the pair of base matches $\langle i_2, j_3\rangle$ and $\langle i, j\rangle$ nor $\langle i_3, j_4\rangle$ and $\langle i, j\rangle$ is arc-preserving. Two edges marked by (3) are added to the graph G. Note that A, B, X, and Y are all substrings of length c.

PROOF. Again, we translate the problem into an INDEPENDENT SET problem on a graph $G = (V, E)$ with bounded degree and use Lemma 5. 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 impose an edge $\{v_{i,j}, v_{i',j'}\}$ iff the corresponding matches $\langle i, j\rangle$ and $\langle i', j'\rangle$ (1) share a common endpoint, (2) are not order-preserving, or (3) are not arc-preserving. Fig. 3 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. 3, base match $\langle i, j\rangle$ shares base $S_1[i]$ with the base match $\langle i, j_1\rangle$.
- We can observe in Fig. 3 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. 3. There, a base match crossing $\langle i, j\rangle$ must be from one of the following sets: $M_1 = \{\, \langle i', j'\rangle \mid S_1[i']$ is in substring B, $S_2[j']$ is in substring $X\,\}$, $M_2 = \{\, \langle i'', j''\rangle \mid S_1[i'']$ is in substring B, $S_2[j'']$ is in substring Y, and $j'' - i'' \leq c\,\}$, or $M_3 = \{\, \langle i''', j'''\rangle \mid S_1[i''']$ is in substring A, $S_2[j''']$ is in substring X, and $j''' - i''' \leq c\,\}$. E.g., in Fig. 3, base match $\langle i_1, j_2\rangle$ is in M_3. 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. 3, 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. There are two such base matches in Fig. 3, $\langle i_2, j_3\rangle$ and $\langle i_3, j_4\rangle$. 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 $O(c^3 n)$ edges. The construction of G can be done in $O(c^3 n)$ time. Hence, the c-diagonal LAPCS(crossing, crossing) problem can be solved in $O((B+1)^l B + c^3 n)$ time, where $B = 2c^2 + 7c + 2$. □

5.3 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 no two arcs of the two underlying sequences share an endpoint. Thus, 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. The so-called *cutwidth* of an arc-annotated sequence (see [8]) is an upper bound on the degree.

Proposition 8 *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* $O((B+1)^l B + c^3 n)$ *time with* $B = c^2 + 2bc - 1$ *(and in* $O((B'+1)^l B' + c^3 n)$ *time 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 6 and 7, 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 $(i, i_1), (i, i_2), \ldots, (i, i_b)$. The third set of edges must be extended to include the edges between $v_{i,j}$ and all vertices, which correspond to base matches involving one of $S_1[i_2], \ldots, S_1[i_b], S_2[j_2], \ldots, S_2[j_b]$. 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 $B = 2c^2 + (4b+3)c + 2b$. $\square$

6 Conclusion

Adopting a parameterized point of view [1,6,7,11], 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 open questions arising from our work are to significantly improve the exponential terms of the running times of our exact algorithms. Depending on what (relative) length of the longest common subsequence is to be expected, one or the other parameterization discussed in the paper might be more appropriate: In the case of a "short" lapcs, the subsequence length is more useful as a parameter, and in the case of a "long" lapcs, the number of deletions is the preferable parameter. In both cases, our results yield fixed-parameter tractability, making exact solutions of this NP-complete problem feasible when the parameter values are small. Our complexity analyses are worst-case, however, and it is a topic of future investigations to study the practical usefulness of our algorithms—possibly adding heuristic elements—by implementations and experiments. In ongoing work, we try to develop efficient implementations of our algorithms. It has to be investigated how to cope with the high memory requirement while traversing the search tree in breadth-first manner. Also, we plan to experiment with real world data, e.g., those used in [19].

In the analysis of the search tree algorithm for LAPCS(nested,nested) (Section 4) it seemed necessary to assume a breadth-first processing of the search tree in order to make the mathematical analysis for the search tree size work. To the best of our knowledge, this is the first time that depth-first search seems not appropriate for the standard technique "bounded search trees" of parameterized complexity theory. It might be of general interest in parameterized complexity theory to further investigate this observation or to find out whether a less space-consuming depth-first search can replace breadth-first search in our scenario. A second challenge with respect to studying the parameterized complexity of LAPCS(nested,nested) is to find non-trivial reduction rules that lead to an efficient and practical reduction to problem kernel [1,6,7,11]. As a consequence, one might obtain an efficient data reduction by preprocessing, a feature of high importance in practical computing. Some additional considerations on LAPCS(nested,nested) and related problems can be found in [15].

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] M. Blanchette, B. Schwikowski, and M. Tompa. Algorithms for phylogenetic footprinting. *Journal of Computational Biology*, 9(2):211–224, 2002.

[3] 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.

[4] 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.

[5] P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110:13–24, 2001.

[6] R. G. Downey. Parameterized complexity for the skeptic (invited paper). In *Proc. of 18th IEEE Conference on Computational Complexity*, pages 147—168, 2003.

[7] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Springer. 1999.

[8] P. A. Evans. *Algorithms and Complexity for Annotated Sequence Analysis.* PhD thesis, University of Victoria, Canada. 1999.

[9] P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proc. of 10th CPM*, number 1645 in LNCS, pages 270–280, 1999. Springer.

[10] 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.

[11] 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.

[12] D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proc. of 40th IEEE FOCS*, pages 512–521, 1999.

[13] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesely Publishing Company. 1989.

[14] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. In *Proc. of 22nd FSTTCS*, number 2556 in LNCS, pages 182-193, 2002. Springer.

[15] J. Guo. *Exact Algorithms for the Longest Common Subsequence Problem for Arc-Annotated Sequences.* Diploma thesis, Universität Tübingen, Fed. Rep. of Germany. February 2002.

[16] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press. 1997.

[17] M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesely Publishing Company. 1978.

[18] 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.*

[19] T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. A general edit distance between two RNA structures. *Journal of Computational Biology*, 9(2): 371—388, 2002.

[20] S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science*, 289: 997–1008, 2002.

[21] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223: 1–72, 1999.

[22] 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.

[23] 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.

[24] G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *Journal of Computer and System Sciences*, 65(3): 465—480, 2002.

[25] B. Ma, L. Wang, and K. Zhang. Computing similarity between RNA structures. *Theoretical Computer Science*, 276: 111—132, 2002.

[26] M. Paterson and V. Dancik. Longest common subsequences. In *Proc. of 19th MFCS*, number 841 in LNCS, pages 127–142, 1994. Springer.

[27] K. Pietrzak. On the parameterized complexity of the fixed alphabet Shortest Common Supersequence and Longest Common Subsequence problems. *Journal of Computer and System Sciences*, 67(4): 757—771, 2003.

[28] D. Sankoff and J. Kruskal (eds.). *Time Warps, String Edits, and Macromolecules.* Addison-Wesley. 1983. Reprinted in 1999 by CSLI Publications.