Exact Algorithms for the LONGEST COMMON SUBSEQUENCE Problem for Arc-Annotated Sequences

Jiong Guo

May 13, 2002

Contents

1 Introduction **5**

2 Biological Motivation **9**

2.1 Some Molecular Biology 9

2.2 Biological Motivation 12

3 Some Basic Definitions **13**

3.1 **LCS** and Some Problems from Graph Theory 13

3.2 Parameterized Complexity 15

3.3 Arc Annotation 20

4 Previous Results **27**

4.1 Classical Complexity 27

4.2 Parameterized Complexity 29

4.3 Complexity of Arc-Preserving Subsequence Problem 30

4.4 Overview of This Work 31

5 c-FRAGMENT, c-DIAGONAL LAPCS **33**

5.1 c-FRAGMENT LAPCS(CROSSING,CROSSING) 33

5.2 c-DIAGONAL **LAPCS**(CROSSING, CROSSING) 38

5.3 **LAPCS**(UNLIMITED, UNLIMITED) 40

6 An Algorithm for LAPCS(NESTED, NESTED) **43**

7 Arc-Preserving Subsequence Problems **53**

7.1 NP-Hardness of **APS**(CROSSING, CHAIN) 53

7.2 **APS**(NESTED, NESTED) 57

8 Conclusions **69**

8.1 Summary of Results 69

8.2 Future Work 70

Chapter 1

Introduction

Algorithms on sequences of symbols have been studied for a long time and now form a fundamental part of computer science. One of the very important problems in analysis of sequences is the **LONGEST COMMON SUBSEQUENCE** (**LCS**) problem. The computational problem of finding the longest common subsequence of k sequences has been researched extensively over the last twenty years and it plays a special role in the field of sequence algorithms. This is partly for historical reasons (many sequence and alignment ideas were first worked out for the special cases of **LCS**), and partly because **LCS** often seems to capture the desired relationship between the strings of interest. This problem has many applications [6, 14, 25]. For $k = 2$, the longest common subsequence is a measure for the similarity of two sequences and is, thus, useful in pattern recognition [21], text compression [22] and, particularly, in molecular biology.

Sequence-level investigation has become essential in modern molecular biology. "The digital information that underlies biochemistry, cell biology, and cell development can be represented by a simple string over letters G, A, T and C. This string is the root data structure of an organism's biology [23]." But to consider genetic molecules only as long sequences consisting of the 4 basic constituents is too simple to determine the function and physical structure of the molecules. For this purpose, other information about the sequences and their parts should be added to the sequences. One prominent source of such

information in molecular biology is the secondary and tertiary structure of the molecules. For example, it is well known that the secondary and tertiary structural features of RNAs are important in molecular mechanism involving their functions. While the primary structure of a molecule is the sequence of bases, its secondary and tertiary structures reveal how the sequence folds into a three-dimensional structure. RNA secondary and tertiary structures are represented as a set of bonded pairs of bases. A bonded pair of bases (base pair) is usually represented as an edge between the two complementary bases involved in the bond. In tertiary structure, the bonds can cross each other, while secondary structure has no crossing bonds. A bond in secondary structure can either inside or outside other bonds. Hence, the ability to analyze molecules requires taking into account all the primary, secondary and tertiary information. More biological background is discussed in Chapter 2.

Early works with these additional information are primary structure based, the sequence comparison is basically done on the primary structure while trying to incorporate secondary structure data [3, 8]. This approach has the weakness that it does not treat a base pair as a whole entity. Recently, an improved model was proposed [10, 11]. In this model, the secondary and tertiary information is combined into the basic sequence, which represents the primary information, to affect subsequent analysis. The system of representing additional information is called annotation scheme. The objects used in this annotation are so-called arcs. An arc is a link or an edge that joins two symbols of the sequence, it corresponds to the chemical bonds between base pairs in the RNA sequence. The RNA structure can then be represented as a base sequence with arc annotations. We call these sequences **arc-annotated sequences**. Arc annotations are defined and discussed in Chapter 3. For related studies concerning algorithm aspects of (protein) structure comparison using "contact maps", refer to [13, 18].

In this work, we will follow this new model and exam the classical **LCS** problem for the sequences with different arc annotations. The focal points are two arc annotations, (CROSSING, CROSSING), where two sequences represent tertiary

structures of two RNA's, and (NESTED, NESTED), which corresponds to an instance of two RNA sequences with secondary structure. Since superimposing arc structures on the basic sequences creates many natural parameters, we explore both the classical and parameterized complexity [1, 9, 12] of the **LCS** problem for sequences with different arc annotation schemes. A summary of previous work will be given in Chapter 4. In Chapter 5, we will prove that the c-FRAGMENT (or c-DIAGONAL) **LAPCS**(CROSSING, CROSSING), parameterized by the length l of the desired subsequence, is fixed-parameter tractable, i.e., it belongs to the complexity class FPT. In Chapter 6, we will give an FPT-algorithm for the **LAPCS**(NESTED, NESTED) with parameters k_1 and k_2, where k_1 and k_2 are the number of the deletions from the two sequences, that we have to make to get an arc-preserving common subsequence. In Chapter 7, we answer some open questions for the **Arc-Preserving Subsequence** problem and give an algorithm which solves the **Arc-Preserving Subsequence** problem with arc structure (NESTED, NESTED) in polynomial time. The last chapter summarizes our results in this work and gives some aspects for future research.

Chapter 2

Biological Motivation

The purpose of this chapter is to provide a brief introduction to molecular biology, especially to DNA and RNA sequences. Here, we only give a few basics; more details can be found in [26].

2.1 Some Molecular Biology

A cell has two classes of molecules: large and small. The large molecules, known as macromolecules, are of three types: DNA, RNA, and protein, among which DNA and RNA are the molecules of most interest to us.

DNA is the basis of heredity and it is constituted of small molecules called nucleotides, which are referred to as bases: adenine (A), cytosine (C), guanine (G), and thymine (T). For our purpose, a DNA molecule can be viewed as a long sequence over the four letter alphabet $\Sigma = \{A, C, G, T\}$. The DNA contained in the cell is known as the *genome*. A *genome* of a human has about 3×10^9 letters, and each human cell contains the same DNA. For each base, there is a complementary base. A is paired with T, and C is paired with G. This pairing is formed by hydrogen bonds and it is essential for the structure of the DNA and for the replication and transcription of its code. The idea is that a single DNA sequence (or strand), e.g., $ACCTGAA$ is paired to a complementary strand $TGGACTT$, as shown in Figure 2.1.

DNA usually occurs double stranded and the bases on one strand fit together

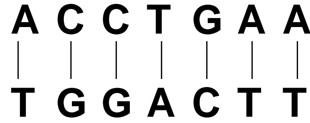

Figure 2.1: 2 DNA strands

with a complementary sequence of bases on the other strand. These two strands form a helical three-dimensional structure. Figure 2.2 presents such a structure.

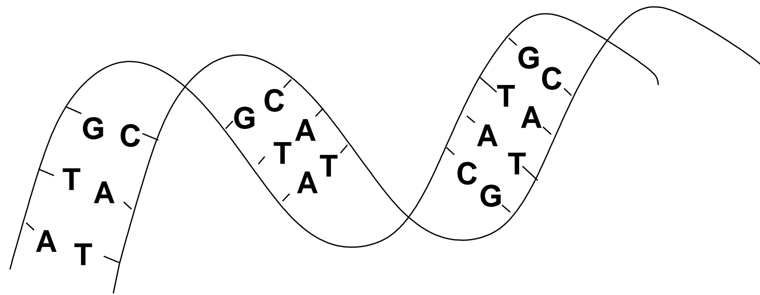

Figure 2.2: The double helix

DNA can be replicated from another DNA already existing. This replication starts with a double helix that has been separated into two single strands. Then, each single strand is used to template new double strands. In this way, two identical DNA molecules are produced, each has one strand of the original molecule. DNA strands can also be transcribed into RNA. RNA is a related ribonucleic acid and it can be modeled by a word over another four letter alphabet of ribonucleotides $\Sigma = \{A, C, G, U\}$, where thymine (T) is replaced by uracil (U). RNA is single-stranded. One strand of the DNA is used to template a single strand of RNA that is made by moving alone the DNA strand. Finally, the double stranded DNA remains as before and a single strand of RNA has been generated. A specific type of RNA, message RNA (mRNA), is read to produce a protein. The genetic code on the mRNA is a language in which triples of the 4 bases - these are 64 possible combinations - specify either a single amino acid or the termination of the protein sequence; such a triple of nucleotides is

called *codon*.

Proteins are built at the ribosomes of a cell, where the mRNA picks up complementary transfer RNA, tRNA. tRNA is another RNA molecule, which is also single stranded, without complementary strand that DNA has. This molecule tends to fold back on itself to form a three-dimensional cloverleaf structure built from approximately 80 bases. See Figure 2.3 for a tRNA.

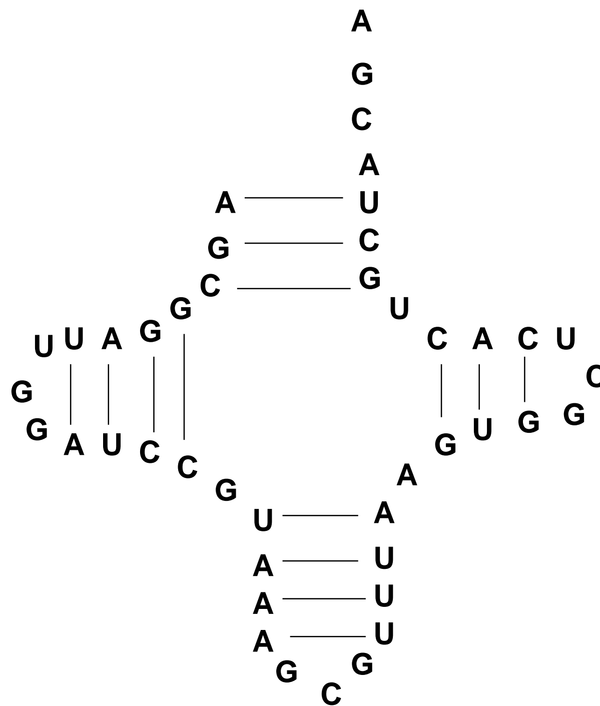

Figure 2.3: A tRNA

Amino acids are linked to these smaller tRNA molecules, and the tRNA interacts with the *codon* of mRNA. In this way, tRNA carries the appropriate amino acid to the mRNA. The ribosomes, which is a complex made of RNA and protein where the protein defined by a messenger RNA is synthesized, have also RNA whose three-dimensional structure enables them to interact with the other molecules physically. The three-dimensional structure of RNA can, thus, be extremely important for its function, and evolution is likely to preserve common structures. Determining the correct fold of a protein is a major open problem in protein analysis. The converse problem, to find an amino acid sequence

that will produce a particular folding or structure, is another great challenge in molecular biology.

2.2 Biological Motivation

Arc-annotated sequences can be applied to describe the secondary and tertiary structures of RNA and protein sequences. Therefore, the problem of comparing arc-annotated sequences has applications in the structural comparison of RNA and protein sequences and it has received much attention in the literature recently. One common way to measure the similarity of two sequences is pairwise sequence comparison, e.g., the longest common subsequence algorithm.

RNA performs a wide range of functions in biological systems. In particular, it is RNA that contains genetic information of viruses such as HIV and therefore regulates the functions of such viruses. Furthermore, it is also widely known that secondary and tertiary structural features of RNA are essential for the molecular mechanisms involved in their function. Thus, it is of massive interest to know how RNA folds to achieve its specific biological functions. A typical feature of RNA molecules is that the comparison of individual sequences can provide information concerning their common features in structure. During the course of evolution, a number of mutations have occurred in these molecules. Comparative analysis of those variations may clarify how such mutations can happen. The common features preserved in course of evolution are likely to be of importance for function. Hence, the ability to compare RNA structure builds the fundament for further study of RNA. When we represent the secondary and tertiary structure of RNA as a basic sequence with arc annotation, algorithms for the longest common subsequence problem for two arc-annotated sequences can play a key role in resolving a preserved secondary and tertiary structure, which corresponds to a preserved molecular conformation and to a preserved function.

Chapter 3

Some Basic Definitions

Since we will explore the classical and parameterized complexity of the **LCS** problem of arc-annotated sequences, this chapter gives some basic definitions and terminologies which we will use in the following chapters. In the first section, we will give a formal definition of the **LCS** problem and introduce some problems that originally arise in graph theory und are useful for our analysis of **LCS** of arc-annotated sequences. Section 3.2 is concerned with parameterized complexity. Since we cannot cover all aspects of parameterized complexity, interested readers are referred to [9]. Arc annotation and the **LCS** problem of arc-annotated sequences are the main objects of the last section. The definitions of various levels of arc annotation and of the **Longest Arc-Preserving Common Subsequence** problem are taken from [11].

3.1 LCS and Some Problems from Graph Theory

As mentioned in Chapter 1, our main method to analyze the similarity of sequences is the pairwise comparison. Thus, the central problem in this work comes from the **LCS** problem, which is very important in classical and parameterized complexity. Here, we give a definition for **subsequence** and the **LCS** problem.

Definition 3.1 ***Subsequence***

Given two sequences S_1, S_2 over some given alphabet Σ, S_2 is a subsequence

of S_1*, if* S_2 *can be obtained from* S_1 *by deleting some letters from* S_1*.*

The *length* of a sequence is denoted by $|S|$. For simplicity, we use $S[i]$ to refer to the ith letter in S, and $S[i_1, i_2]$ to denote the subsequence of S from the i_1th letter to the i_2th letter ($1 \leq i_1 \leq i_2 \leq |S|$).

Definition 3.2 ***Longest Common Subsequence Problem (LCS)***
Given a set of k *sequences* $S_1, S_2, \ldots, S_k$ *over some alphabet* Σ*, the* longest common subsequence problem *asks for a longest sequence* P *that is a subsequence of* $S_1, S_2, \ldots$ *and* S_k*.*

To date, most research has focused on deriving efficient algorithms for the **LCS** problem when $k = 2$. This problem can be solved by dynamic programming in time $O(|S_1| \cdot |S_2|)$ [15]. If the number of sequences k is unrestricted, the **LCS** problem is NP-complete [22]. However, certain algorithms for the case $k = 2$ have been extended to yield algorithms that require $O(n^{k-1})$ time and space, where n is the length of the longest of the k sequences [2, 16].
In order to prove some complexity results of the **LCS** problem of arc-annotated sequences, we will use reductions to or from some problems in graph theory with known complexity. These problems are VERTEX COVER, INDEPENDENT SET and CLIQUE.

Definition 3.3 VERTEX COVER, VC
An edge e *of an undirected graph* $G = (V, E)$ *is incident to a vertex* v *if* v *is one of the endpoints of* e*. A set of vertices* $V' \subseteq V$ *is called a* vertex cover *in* G *if for each* $e \in E$*, there exists a* $v \in V'$ *such that* e *is incident to* v *(*$\exists u \in V$ *such that* $(u, v) = e$*). Given an undirected graph* $G = (V, E)$ *and a positive integer* k*, the* VERTEX COVER *problem asks whether* G *has a vertex cover of size at most* k*.*

Definition 3.4 INDEPENDENT SET, IS
Let $G = (V, E)$ *be an undirected graph, and let* $I \subseteq V$*. We say that* I *is an* independent set *if for each pair* $i, j \in I$*,* $i \neq j$*, there is no edge between* i

and j. The INDEPENDENT SET *problem asks, given a parameter k, if there is an independent set I with $|I| \geq k$.*

Definition 3.5 CLIQUE
Given an undirected graph $G = (V, E)$, and a parameter k, the CLIQUE *problem asks whether there is a vertex set $C \subseteq V$ with $|C| \geq k$ such that for all vertices $u, v \in C$ with $u \neq v$ there is an edge between v and u.*

These three problems are all known to be NP-complete [24]. It is easy to see that a set VC is a vertex cover in G if and only if $V \setminus VC$ is an independent set in G. Also, there is a vertex cover in G of size k if and only if there is an independent set in G of size $|V| - k$. The VERTEX COVER problem is, conveniently, a minimization problem, while the INDEPENDENT SET problem is a maximization problem.

3.2 Parameterized Complexity

In this section, we give an overview of the aspects of parameterized complexity, which are relevant to this work. Classical polynomial complexity, its reductions and NP-hardness are discussed in depth by Paradimitriou [24]. Parameterized complexity was introduced by Downey and Fellows [9].
Many natural problems have now been shown to be NP-complete or worse, which means, it is highly unlikely that there exits efficient algorithm for these problems. However, we can find for some of the problems algorithms such that a main part of the problem instance contributes to the overall running time “in a good way” (e.g., polynomially), and identify aspects of the input, which determine the combinatorial explosion of the running time. Then, these aspects can be used as parameters in the hope that these parameters are small in applications.
While the notation of *polynomial time* is central to the classical formulation of computional complexity, central to parameterized complexity is the notion of ***fixed-parameter tractability***.

Definition 3.6 ***Fixed-Parameter Tractability*** *A parameterized problem L is fixed-parameter tractable, if and only if there is an algorithm which can in time $f(k)n^c$ decide whether $(x, k) \in L$, where x is the input and k is the parameter. Further $n := |x|$, c is a constant that is independent of both n and k, and $f : \mathbb{N} \mapsto \mathbb{R}$ is an arbitrary function.*

We denote the family of all fixed-parameter tractable parameterized problems by FPT.

Here, we give an example for fixed-parameter tractability, which will be used in Chapter 5.

Definition 3.7 ***Maximum Independent Set-B, MAXIS-B***

Given a simple graph $G = (V, E)$ in which each vertex has degree at most B, the MAXIS-B problem asks for a maximum independent set of G.

Lemma 3.8 *MAXIS-B, parameterized by the size k of the independent set, can be solved by an FPT-algorithm in time $O((B + 1)^k B^2)$.*

Proof. We use the notation $G - \{u\}$ to denote the deletion of the vertex u and all edges incident to u from the graph G. We can construct a search tree of height k as follows.

The root of the tree is labeled with an empty independent set I and the graph G. First, we find the vertex u with minimum degree which can have at most B neighbors $\{v_1, v_2, \ldots\}$. Any independent set of G can contains at least either u or one of its neighbors, so we create the children of the root corresponding to these possibilities. The first child is labeled with $\{u\}$ and $G - \{u\} - \{$all neighbors of $u\}$, and the second is labeled with $\{v_1\}$ and $G - \{v_1\} - \{$all neighbors of $v_1\}$, and the other children are labeled in the same way for the remaining neighbors of u. There are at most $B + 1$ children of the root node. The set of vertices labeling a node represents a "possible" independent set, and the graph labeling the node represents what remains to be checked in G. In general, for a node labeled with the set of vertices S and the subgraph H of G, we choose the vertex v with minimum degree in H and create at most $B + 1$ child nodes. These child nodes are labeled in a similar way as the children of the root node.

If we can create a node at height k in the tree, then an independent set of cardinality at least k has been found. There is no need to explore the tree beyond height k. As we can easily see, each node has at most $B+1$ children. Thus, the tree can have a maximum size of $(B+1)^k$. At each node, the deletion of vertex u and its neighbors and all edges incident to u and its neighbors can be done in time $O(B^2)$. Therefore, this algorithm takes $O((B+1)^k B^2)$ many steps. □

As we have seen in classical complexity, the basic idea behind virtually all completeness results is the notion of a reduction. Therefore, we will need a new kind of reduction which is "parameter preserving" and can be used to show that two problems have the same parameterized complexity.

Definition 3.9 ***Fixed-Parameter Reducibility***
Let L and $L^{'}$ be two parameterized problems, $L \subseteq \Sigma^* \times \mathbb{N}$ and $L^{'} \subseteq \Gamma^* \times \mathbb{N}$. We say that L is fixed-parameter reducible to $L^{'}$, if there are functions $k \mapsto k^{'}$ and $k \mapsto k^{''}$ on $\mathbb{N}$ and a function $(x,k) \mapsto x^{'}$ from $\Sigma^* \times N$ to Γ^* such that

(a) $(x,k) \mapsto x^{'}$ is computable in time $k^{''} |x|^{O(1)}$,

(b) $(x,k) \in L \Leftrightarrow (x^{'}, k^{'}) \in L^{'}$.

Before we establish a hierarchy of parameterized complexity, we need some definitions, which help to define the classes in the hierarchy.

Definition 3.10 ***Boolean Circuit***
A Boolean circuit is a directed graph $G = (V,E)$, where the nodes in $V = \{1,\ldots,n\}$ are called the **gates** *of G. There are no cycles in the graph. All nodes in the graph have* **fan-in** *(the number of incoming edges). Each gate $i \in V$ in the graph has a* **sort** *$s(i)$ associated with it, where $s(i) \in \{$ TRUE, FALSE, AND, OR, NEGATION $\} \cup \{x_1, x_2, \ldots\}$. If $s(i) \in \{$ TRUE, FALSE $\} \cup \{x_1, x_2, \ldots\}$, then the* **fan-in** *of i is 0, that is, i has no incoming edges. Gates with no incoming edges are called the* **input gate***. Finally, there is a gate, which has no outgoing edges. It is called* **output gate** *of the circuit.*

Circuits can have gates of two types: a **small gate** *has bounded* **fan-in***, while a* **large gate** *has unbounded* **fan-in***.*

A circuit, which has no inputs of sort TRUE or FALSE, can be thought of as representing a Boolean expression. Conversely, given a Boolean expression δ, there is a simple way to construct a circuit C_δ such that, for any truth assignment T appropriate to both (all variables in δ and C_δ are defined in T), $T(C_\delta) =$ TRUE if and only if δ is satisfied by the assignment T. A truth assignment T satisfies a Boolean expression δ, if all variables in δ are defined in T and δ becomes true with variables replaced by their truth values in T. A circuit C has a weight k satisfying assignment, if the Boolean expression δ, which corresponds to C, has a satisfying assignment, where δ hat exactly k variables set to be TRUE. The construction of C_δ follows the inductive definition of δ, and builds a new gate i for each subexpression encountered.

Definition 3.11 ***Circuit Depth***

The depth of a circuit is the maximum number of gates on any path from an input gate to the output gate.

Definition 3.12 ***Circuit Weft***

The weft of a circuit is the maximum number of large gates on any path from an input gate to the output gate.

Let $\Gamma = \{C_1, C_2, C_3, \ldots\}$ be a family of circuits. Associated with Γ is a basic parameterized language $L_\Gamma = \{\langle C_i, k\rangle \mid C_i$ has a weight k satisfying assignment$\}$. By $L_{\Gamma(t,h)}$, we denote the subset of L_Γ of circuits with weft t and depth h.

Definition 3.13 ***Basic Hardness Class***

A parameterized problem L is in the complexity class $W[t]$ if it is fixed-parameter reducible to $L_{\Gamma(t,h)}$, where the depth h is constant.

Definition 3.14 ***W-hierarchy***

The W-hierarchy is the set of the classes $W[t]$ together with two other classes, $W[SAT]$ and $W[P]$. $W[P]$ denotes the class obtained by having no restriction

on the depth, i.e, P-size circuits, and $W[SAT]$ denotes the restriction to boolean formulas of P-size. Hence, the W-hierarchy is

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq W[SAT] \subseteq W[P].$$

We conjecture that each of the containments is proper.

$W[SAT]$ denotes the class of problems reducible to WEIGHTED SATISFIABILITY, while $W[P]$ denotes the class of problems reducible to WEIGHTED CIRCUIT SATISFIABILITY. Given a Boolean formula X and a positive integer k, WEIGHTED SATISFIABILITY asks whether X has a weight k satisfying assignment. WEIGHTED CIRCUIT SATISFIABILITY asks whether a given decision circuit C has a weight k satisfying assignment.

Some common problems known to be NP-complete in classical complexity fall into different classes in the W-hierarchy when their natural parameters are used. For example, if the parameter is the desired size of the vertex subset, INDEPENDENT SET and CLIQUE are both $W[1]$-complete, while VERTEX COVER$\in FPT$. VERTEX COVER can be solved by an algorithm with running time $O(kn + 1.2852^k)$ [7].

Definition 3.15 ***Parameterized Variations of*** **LCS**
Given a set of k sequences $S_1, \ldots, S_k$ and a positive integer m, parameterized variations of **LCS** *ask for a sequence P of length at least m that is a subsequence of all of $S_1, \ldots, S_k$. We refer to the variation with parameter k as* **LCS-1***, with parameter m as* **LCS-2***, with parameters k and m as* **LCS-3***.*

The parameterized complexity of the variations of the **LCS** problem is summarized in the following table (Table 3.1). The results are all due to Bodlaender *et al.* [5, 4].

The fixed-parameter tractability of a problem can illustrate some of the possibilities for problem parameterization. It is frequently complained by computer scientists with a practical orientation that the classical complexity framework is not sufficiently realistic. As shown by the example of VERTEX COVER, parameterization provides a way how to cope with NP-hardness.

Problem	Parameter	$\|\Sigma\|$ Unbounded	$\|\Sigma\|$ Fixed
LCS-1	k	$W[t]$-hard, $t \geq 1$	unknown
LCS-2	m	$W[2]$-hard	FPT
LCS-3	k, m	$W[1]$-complete	FPT

Table 3.1: Parameterized complexity of **LCS**

3.3 Arc Annotation

While the previous two sections have provided some basic knowledge of classical and parameterized complexity, we will from now focus on the main problem of this work, the **Longest Arc-Preserving Common Subsequence** problem. The purpose of having arc annotation is to express additional information about a sequence in a way that the sequence and the additional information can be analyzed and manipulated simultaneously. Arcs represent binary relations between sequence symbols. Hence, they can be used to join base pairs that are chemically bonded in the represented biological sequence. This application is particularly relevant to RNA sequences, whose chemical bonds can be described by annotating the sequence with these arcs. Figure 3.1 shows a part of tRNA and its corresponding arc-annotated sequence.

Definition 3.16 ***Arc Annotation***
The arc annotation set A of a sequence S is a set of pairs of positions in S:

$$A = \{\, (i_1, i_2) \mid 1 \leq i_1 < i_2 \leq |S| \} \subseteq \{1, \ldots, |S|\}^2$$

Then, the sequence S with such an arc annotation is called an arc-annotated sequence*, denoted by (S,A).*

Since we incorporate both arcs and sequences into an overall measure of similarity, the definition of **LCS** must be adjusted to incorporate the arc structure. A common subsequence should also have the common arcs of the input sequences. To preserve arcs, a subsequence that selects both endpoints of an arc from one sequence must map those endpoints to the endpoints of some arc from the other sequence.

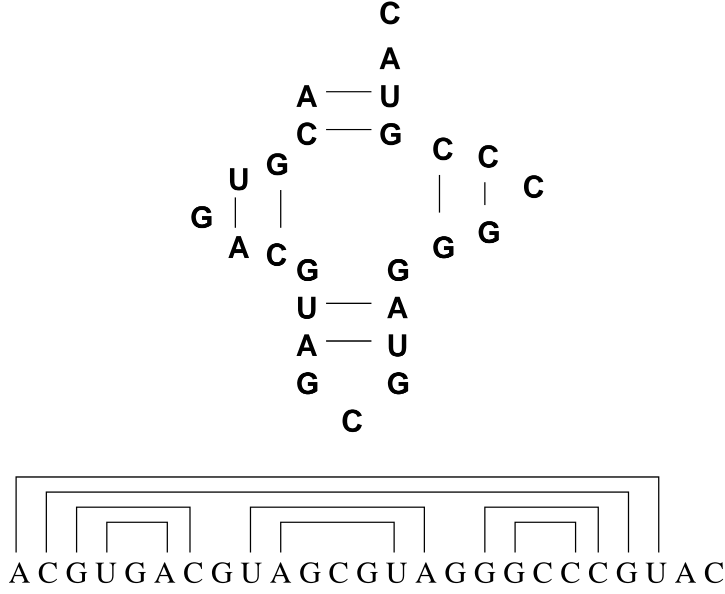

Figure 3.1: A tRNA and its corresponding arc-annotated sequence

Definition 3.17 ***Longest Arc-Preserving Common Subsequence Problem*** *(***LAPCS***)*
Given two arc-annotated sequences (S_1,A_1) and (S_2,A_2), the **LAPCS** *problem asks to find the longest common subsequence of S_1 and S_2 which preserves the arcs, i.e., to find a* **mapping** *$MS \subseteq \{1, \ldots, |S_1|\} \times \{1, \ldots, |S_2|\}$ such that*

1. *the* **mapping** *is one-to-one and preserves the order of the subsequence:*

$$\forall (i_1, j_1),\ (i_2, j_2) \in MS$$

$$i_1 = i_2 \Longleftrightarrow j_1 = j_2 \quad \text{and}$$

$$i_1 < i_2 \Longleftrightarrow j_1 < j_2$$

2. *the arcs induced by the* **mapping** *are preserved:*

$$\forall (i_1, j_1),\ (i_2, j_2) \in MS$$

$$(i_1, i_2) \in A_1 \Longleftrightarrow (j_1, j_2) \in A_2$$

3. *the* **mapping** *produces a common subsequence:*

$$\forall (i, j) \in MS,\ S_1[i] = S_2[j]$$

We name the pair $\langle i, j \rangle$ a *base match*, if $S_1[i] = S_2[j]$ for some pair of positive integers i and j. 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 $(j_1, j_2) \in A_2$ for some integers $i_1 < i_2$ and $j_1 < j_2$, then the pair $\langle (i_1, i_2), (j_1, j_2) \rangle$ is an *arc match*.

A restricted version of the above problem is defined as follows.

Definition 3.18 ***Arc-Preserving Subsequence Problem (APS)***
Given two arc-annotated sequences (S_1,A_1) and (S_2,A_2), $|S_1| \leq |S_2|$, the **APS** *problem asks whether there is an arc-preserving mapping from S_1 to S_2, i.e., S_1 can be obtained from S_2 by deleting some bases and arcs incident on these bases from S_2.*

When arcs are used to link sequence symbols to represent nonsequential information, comparing the resulting annotated sequences is much more complex than classical **LCS**. Since in practice of RNA and protein sequence comparison arc sets are likely to satisfy some constraints (e.g. bond arcs do not cross in the case of tRNA sequences), it is of interest to consider various restrictions on arc structure. As we will see, the different restrictions on arc annotation can alter the computational complexity of the **LCS** problem.

Definition 3.19 ***Restricted Variations of LAPCS***
There are four natural restrictions on the arc set A of a sequence S:

1. *no two arcs share an endpoint:*

$$\forall (i_1, i_2), (i_3, i_4) \in A$$

$$(i_1 \neq i_4) \wedge (i_2 \neq i_3) \wedge (i_1 = i_3 \Longleftrightarrow i_2 = i_4)$$

2. *no two arcs cross each other:*

$$\forall (i_1, i_2), (i_3, i_4) \in A$$

$$i_1 \in [i_3, i_4] \Longleftrightarrow i_2 \in [i_3, i_4]$$

3. *no two arcs nest:*

$$\forall (i_1, i_2), (i_3, i_4) \in A$$

$$i_1 \leq i_3 \Longleftrightarrow i_2 \leq i_3$$

4. no arcs:

$$A = \emptyset$$

These four restrictions produce five levels of permitted arc structures:

- UNLIMITED*: no restrictions,*
- CROSSING*: restriction (1),*
- NESTED*: restrictions (1) and (2),*
- CHAIN*: restrictions (1), (2), and (3),*
- PLAIN*: restriction (4).*

In the following, **LAPCS**(x, y) *represents an* **LAPCS** *problem where the arc structure of* S_1 *is of level* x *and the arc structure of* S_2 *is of level* y. *Assume that* x *is at the same level of or higher than* y.

Note that the problem **LAPCS**(NESTED, NESTED) effectively models the similarity between two tRNA sequences, particularly the secondary structures.
The following table (Table 3.2) shows the inclusion relation between the levels of restrictions on **LAPCS**(x, y). Moreover, we give the definitions of two special cases of the **LAPCS** problem, which were first studied in [20]. The special cases are motivated from biological applications [14, 19].

Definition 3.20 c-FRAGMENT **LAPCS** ***Problem*** *(*$c \geq 1$*)*
Given two arc-annotated sequences which are divided into fragments of lengths exactly c (the last fragment can have a length less than c), the allowed matches are those between fragments at the same location.

For example, all matches induced by a 2-FRAGMENT **LAPCS** are required to have the form

$$\langle 2i - \frac{1}{2} \pm \frac{1}{2}, 2i - \frac{1}{2} \pm \frac{1}{2} \rangle, \quad i \geq 1.$$

UNLIM, UNLIM								
$\cup$								
UNLIM, CROSS	$\supset$	CROSS, CROSS						
$\cup$		$\cup$						
UNLIM, NEST	$\supset$	CROSS, NEST	$\supset$	NEST, NEST				
$\cup$		$\cup$		$\cup$				
UNLIM, CHAIN	$\supset$	CROSS, CHAIN	$\supset$	NEST, CHAIN	$\supset$	CHAIN, CHAIN		
$\cup$		$\cup$		$\cup$		$\cup$		
UNLIM, PLAIN	$\supset$	CROSS, PLAIN	$\supset$	NEST, PLAIN	$\supset$	CHAIN, PLAIN	$\supset$	PLAIN, PLAIN

Table 3.2: Problem inclusions for different levels of restriction.

We use x, y to denote the arc structures of two arc-annotated sequences. The symbol $\supset$ indicates the inclusion relation between different levels resulting by the restriction hierarchy. UNLIM, UNLIM is the most general **Longest arc-preserving Common Subsequence** problem, and PLAIN, PLAIN is the unannotated **Longest Common Subsequence** problem.

Definition 3.21 c-DIAGONAL **LAPCS** ***problem*** *($c \geq 0$)*
c-DIAGONAL **LAPCS** *is an extension of* c-FRAGMENT **LAPCS**, *where base* $S_1[i]$ *is allowed only to match bases in the range* $S_2[i-c, i+c]$.

The c-DIAGONAL and c-FRAGMENT **LAPCS** problems are relevant in the comparison of conserved RNA sequences where we already have a rough idea about the correspondence between bases in the two sequences.
The arc structure can provide many natural parameters for the **Longest Arc-Preserving Common Subsequence** problem. In the following, we give two examples of such parameters concerning arc structure.

Definition 3.22 ***Cutwidth***
Given an arc-annotated sequence (S, A), *the* **cutwidth** *of the arc structure is the maximum number of arcs that pass by or end at any position of the sequence.*

Definition 3.23 ***Bandwidth***
Given an arc-annotated sequence (S, A), *the* **bandwidth** *of the arc structure is the maximum distance between the two endpoints of an arc, i.e., if we denote*

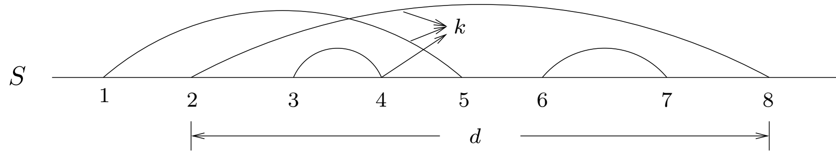

Figure 3.2: Cutwidth and Bandwidth.

The arc-annotated sequence S has 8 bases. There are 3 arcs passing by or ending at the 4th base and no other base has more arcs passing by or ending at it. Thus, S has a cutwidth of 3, denoted by c. It is obvious that the arc between the 2nd and 8th bases is the longest arc of S. The bandwidth of S is then 6, denoted by d.

bandwidth *as* d*, then for any* $(i_1, i_2) \in A, i_2 - i_1 \leq d$*.*

Figure 3.2 illustrates an arc-annotated sequence with cutwidth of 3 and bandwidth of 6.

Chapter 4

Previous Results

As already mentioned in Chapter 3, for referring to the problems, we follow the convention that at the arc structure of sequence S_1 is at least as complex as that of sequence S_2. Using five levels of arc structures, we distinguish 15 distinct variations of **LAPCS** where S_1 and S_2 may have different level of arc structure (see Table 3.2). This chapter summarizes previous results of these 15 **LAPCS** variations, not only in classical but also in parameterized complexity framework. Various parameters have been used to exam **LAPCS**, such as the length l of the desired subsequence, the cutwidth k and the bandwidth d. The third section is concerned with the ARC-PRESERVING SUBSEQUENCE problem. The 5 levels of arc structure can also be used to APS. At last, we will address the problems that we will explicitly discuss in this work.

4.1 Classical Complexity

When the arc structure x of sequence S_1 is at least CROSSING, **LAPCS**(x,y) is NP-hard [11]. INDEPENDENT SET, which is known to be NP-complete, can be reduced to **LAPCS**(UNLIMITED, PLAIN) or **LAPCS**(CROSSING, PLAIN). If the arc structures of both sequences are lower than NESTED, then the **LAPCS** problem is solvable in polynomial time [11, 15]. The NP-hardness of the problem **LAPCS**(NESTED, NESTED) was shown in [20]. Jiang et al. [17] presented a dynamic programming algorithm to compute the **LAPCS**(NESTED, CHAIN)

and **LAPCS**(NESTED, PLAIN) in running time $O(nm^3)$. **LAPCS**(CROSSING, CROSSING) admits a 2-approximation algorithm running in $O(nm)$, and **LAPCS** (UNLIMITED, PLAIN) cannot be approximated within ratio n^ϵ for any $\epsilon \in (0, \frac{1}{4})$, where n denotes the length of the longer input sequence [17]. Table 4.1 gives a summary of results concerning classical complexity of **LAPCS**.

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	$NP - c^*$ [11]				
CROSSING	—	$NP - c^+$ [11]			
NESTED	—		$NP - c^\#$ [20]	$O(nm^3)$ [17]	
CHAIN	—			$O(nm)$ [11]	
PLAIN	—				$O(nm)$ [15]

*: *not approximable within* $n^\epsilon, \epsilon < 1/4$ [17]

+: *2-approximable, MAXSNP-Hard* [17]

#: *2-approximable*

Table 4.1: Classical Complexity

Lin *et al.* [20] showed also the NP-hardness results for the c-FRAGMENT (with $c > 2$) and c-DIAGONAL (with $c > 1$) **LAPCS**. The 1-FRAGMENT **LAPCS** (CROSSING, CROSSING) and 0-DIAGONAL **LAPCS** (CROSSING, CROSSING) are solvable in time $O(n)$. See Table 4.2.

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	NP-hard [20]			?	
CROSSING	—	NP-hard [20]		?	
NESTED	—		$NP - hard^\#$ [20]	?	

#: *admits a PTAS*

Table 4.2: Complexity result for c-FRAGMENT $(c > 1)$ and c-DIAGONAL $(c > 0)$ **LAPCS**

4.2 Parameterized Complexity

Since many of these 15 variations of **LAPCS** are NP-hard or have no currently known polynomial time algorithm, the parameterized complexity of these problems has also been investigated in some of the above works. The parameters being used include: the length l of the desired subsequence , the cutwidth k of the arc structure and the bandwidth d of the arc structure. The length of desired subsequence l is independent of the other parameters, while the cutwidth of an arc structure lower than UNLIMITED is upper-bounded by the bandwidth of the arc structure.

If parameterized by the length of the desired subsequence, the **LAPCS** problem with at least one sequence having an UNLIMITED arc structure was shown to be $W[1]$-complete [11]. If the arc structures of both sequences are CROSSING, the problem also turns out to be $W[1]$-complete [11]. The same reductions as for the classical hardness result can be used to show the corresponding parameterized complexity. For other variations, in which the arc structure of sequence S_1 is CROSSING or NESTED and the arc structure of S_2 is at most NESTED, the parameterized complexity of **LAPCS** is still unknown. Table 4.3 summarizes the parameterized complexity of **LAPCS**, when parameterized by the length of the desired subsequence.

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	$W[1] - complete$ [11]				
CROSSING	—	$W[1] - complete$ [11]	?		
NESTED	—		?		

Table 4.3: Parameterized by the length l of desired subsequence

Evans [11] presented an algorithm running in time $O(9^k nm)$, where k is the cutwidth or bandwidth of the arc structure, to solve the **LAPCS** problem for variations with arc structure of both sequences being at most CROSSING. It uses multiple tables to compute the length of longest arc-preserving common subsequence in a manner similar to the algorithm without arcs. To enable

matched final endpoints to be aligned with matched starting endpoints of arcs, the algorithm uses a tree data structure to keep track of all combinations of initial endpoints matches that lie on a path that produces this maximum value. Since the bandwidth of a CROSSING arc structure is an upper bound of cutwidth, the algorithm developed for the parameter cutwidth can also be used for the parameter bandwidth. Therefore, Table 4.4 and Table 4.5 are identical.

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	?				
CROSSING	—	$O(9^k nm)$ [11]			
NESTED	—		$O(k^2 4^k nm)$ [11]		

Table 4.4: Parameterized by the cutwidth k of arc structure of both sequences

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	?				
CROSSING	—	$O(9^d nm)$ [11]			
NESTED	—		$O(d^2 4^d nm)$ [11]		

Table 4.5: Parameterized by the bandwidth d of arc structure of both sequences

4.3 Complexity of Arc-Preserving Subsequence Problem

The exact matching version of **LAPCS**, arc-preserving subsequence problem, arises in widely varying applications. For example, searching a specific pattern in DNA/RNA database. The existing works that analyze the **LAPCS** problem give no hardness result for this problem. But we can extend the reductions for classical complexity of **LAPCS** to show that even the **APS** problem of some arc structures is NP-hard. Assuming the shorter sequence always has the same

or lower level of arc structure, we also summarize the classical complexity of **APS** problem in Table 4.6.

	UNLIMITED	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	$NP-hard$ [11]				
CROSSING	—	$NP-hard$ [11]	?		
NESTED	—		?	$O(nm^3)$ [20]	

Table 4.6: Classical complexity for **APS** problem

4.4 Overview of This Work

Comparing the summaries in the previous three sections with the fact that classical **LCS** for two sequences can be solved in polynomial time, we can come to the conclusion that adding the arc annotation to the basic sequences makes the **LCS** problem much more complex. However, the arc annotation model provides the most natural and most intuitive way to describe the structure of the large molecules. Thus, to find exact and effective algorithms for the **LAPCS** problem with various arc annotations, is the main goal of this work. At first, we will prove the fixed-parameter tractability for the restricted versions of **LAPCS**, c-FRAGMENT and c-DIAGONAL **LAPCS**, when taking the length of the common subsequence as the problem parameter. Lin *et al.* [20] gave polynomial time approximation schemes (**PTAS**) for c-FRAGMENT and c-DIAGONAL **LAPCS**(NESTED, NESTED). Our fixed-parameter tractability result is also tenable for more general arc structures, (CROSSING, CROSSING) and even (UNLIMITED, UNLIMITED) with the **degree** of sequences as the second parameter (see Chapter 5). For the most important variant of **LAPCS** problem, general **LAPCS**(NESTED, NESTED), there are only an FPT-algorithm from Evans [11] with cutwidth as parameter and a quadratic time factor-2-approximation algorithm from Jiang *et al.* [17]. However, the fixed-parameter tractability of this problem is still an open question, when parameterized by the length l of the desired subsequence. We will give an exact, fixed-parameter algorithm

that solves the **LAPCS**(NESTED, NESTED) problem in time $O(3.31^{k_1+k_2} \cdot n)$, where n is the maximum input sequence length and k_1 and k_2 are the number of deletions allowed for S_1 and S_2 respectively. It should be clear that $l = |S_1| - k_1$ and $l = |S_2| - k_2$. This algorithm provides an effective solution for the case of reasonably small values of k_1 and k_2 (see Chapter 6). Furthermore, we will answer some open questions in Table 4.6, namely the complexity for **APS**(CROSSING, CHAIN), **APS**(CROSSING, PLAIN) and **APS**(NESTED, NESTED) (see Chapter 7).

Chapter 5

c-FRAGMENT, c-DIAGONAL LAPCS

In this chapter, we investigate the c-FRAGMENT LAPCS(CROSSING, CROSSING) and the c-DIAGONAL LAPCS(CROSSING, CROSSING) problems. We give algorithms for these problems when parameterized by the length l of the desired subsequence. The restricted versions c-DIAGONAL and c-FRAGMENT of LAPCS(CROSSING, CROSSING) were already treated by Lin *et al.* [20]. They gave PTAS's for these problems. We want to remark that the running times for the following algorithms are based on worst case analysis. The algorithms are expected to perform much better in practice.

5.1 c-FRAGMENT LAPCS(CROSSING, CROSSING)

Before entering into details of the algorithm for c-FRAGMENT LAPCS(CROSSING, CROSSING), we review briefly an algorithm which solves 1-FRAGMENT LAPCS (CROSSING, CROSSING) in linear time [20].

Let (S_1, A_1) and (S_2, A_2) be an instance of 1-FRAGMENT LAPCS (CROSSING, CROSSING). We assume that $n = |S_1| = |S_2|$. If the sequences do not have the same length, we can extend the shorter one by adding a sequence of a letter not in the alphabet at its end. We construct a graph G as follows. If the two sequences induce a base match $\langle i, i \rangle$, then we create a vertex v_i. If the sequences

induce a pair of base matches $\langle i, i\rangle$ and $\langle j, j\rangle$ and (i, j) is an arc in either A_1 or A_2 but not both, then we impose an edge connecting v_i and v_j in G. It is clear that G has maximum degree 2 and every independent set of G one-to-one corresponds to an arc-preserving common subsequence of (S_1, A_1) and (S_2, A_2). Since G is composed only of a collection of disjoint cycles and paths, we can compute a maximum independent set of G in linear time. Therefore, the 1-FRAGMENT LAPCS(CROSSING, CROSSING) is solvable in $O(n)$ time.

Since 1-FRAGMENT limits the base matches to bases at the same position in the two sequences, the resulting graph G is simple. c-FRAGMENT relaxes this limitation and allows base matches of the form $\langle i, j\rangle$, where $S_1[i]$ and $S_2[j]$ are not at the same position but in the same fragment. Using the above reduction, the resulting graph will be much more complicated. However, we will show in the following that this graph has a bounded degree, such that the fixed-parameter tractable algorithm in Lemma 3.8 can be used to find out the maximum independent set of such a graph.

Lemma 5.1 *The c-FRAGMENT LAPCS (CROSSING, CROSSING) is polynomially reducible and fixed-parameter reducible to MAXIS-B, in time $O(c^3 n)$.*

Proof.

Reduction: 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. We assume that both sequences have the same length as in the algorithm for the 1-FRAGMENT variant, $n = |S_1| = |S_2| = p \cdot c$, where $p \in \mathbb{N}$. We construct a graph $G = (V, E)$ as follows.

Each base of S_1 in the ith fragment $(1 \leq i \leq p)$ can only be matched to the bases in the ith fragment of S_2. If there is a such base match, then we create a vertex in G, i.e., we define

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

As explained in Definition 3.17, the **LAPCS** problem asks for a matching, which is one-to-one, order-preserving and arc-preserving. Since we want to translate

the **LAPCS** instance into an instance of the INDEPENDENT SET problem on G, the edges of G will represent all conflicting matches. Therefore, for each two vertices v_{i_1,j_1} and v_{i_2,j_2}, i.e, for two base matches $S_1[i_1] = S_2[j_1]$ and $S_1[i_2] = S_2[j_2]$ ($i_1 \neq i_2$ or $j_1 \neq j_2$), such a conflict may arise from three different situations:

1. Both matches are in the same fragment and both matches involve the same position in S_2 or in S_1, i.e., $(i_1 \neq i_2 \wedge j_1 = j_2) \vee (i_1 = i_2 \wedge j_1 \neq j_2)$.
2. Both matches are in the same fragment and they cross each other, they do not preserve the order of the subsequence, i.e., $((i_1 < i_2) \wedge (j_1 > j_2)) \vee ((i_1 > i_2) \wedge (j_1 < j_2))$.
3. The two matches represented by v_{i_1,j_1} and v_{i_2,j_2} are not arc-preserving, i.e., $((i_1, i_2) \in A_1 \wedge (j_1, j_2) \notin A_2) \;\vee\; ((i_1, i_2) \notin A_1 \wedge (j_1, j_2) \in A_2)$

Figure 5.1 illustrates an example of this reduction.

For the running time analysis of this construction, note that there can be up to c^2 vertices in G for each fragment of the sequence. Hence, we have a total of cn vertices. Each vertex in G can have at most $c^2 + 2c - 1$ adjacent edges, which come from following three groups:

- 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 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 due to the first situation.
- 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 will be imposed on vertex $v_{i,j}$ due to the second situation.
- 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

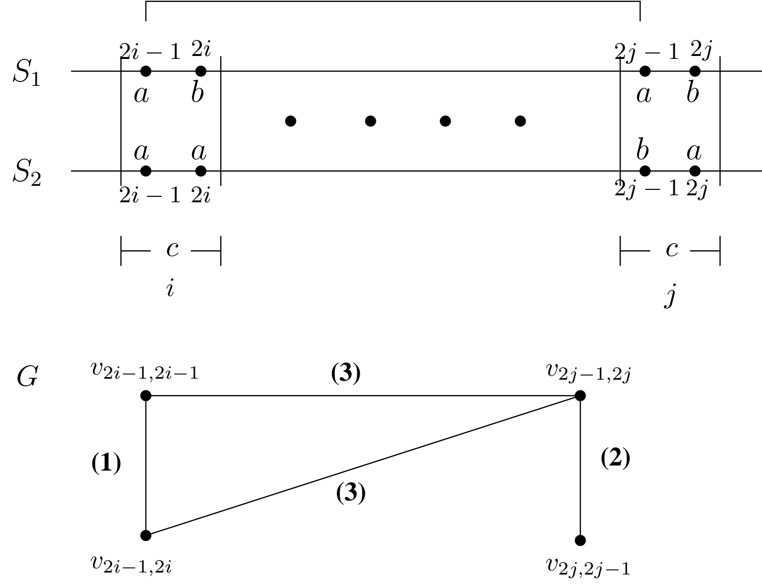

Figure 5.1: 2-fragment **LAPCS**

There are four base matches in the two segments shown in this figure. They correspond to the four vertices in G. The edge (1) in G is imposed due to the first situation in our construction, the base matches $\langle 2i-1, 2i-1\rangle$ and $\langle 2i-1, 2i\rangle$ share the base $S_1[2i-1]$. Since base matches $\langle 2j-1, 2j\rangle$ and $\langle 2j, 2j-1\rangle$ fit the second situation, their corresponding vertices are joined by an edge denoted by (2). While an arc joins bases $S_1[2i-1]$ and $S_1[2j-1]$, there is no arc in S_2 between ith and jth fragments. According to the third situation, edges are imposed between the vertices which correspond the base matches involving the endpoints of the arc $(2i-1, 2j-1)$. These edges are marked with (3).

$\langle i, j\rangle$ 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 imposed on vertex $v_{i,j}$ due to the third situation can amount to $2c$.

Thus, the resulting graph G has a vertex degree bounded by $B = c^2 + 2c - 1$. 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 $O(c^3 n)$.

To show that the above construction is a correct reduction from c-FRAGMENT **LAPCS**(CROSSING,CROSSING) to MAXIS-B, we need to verify that there is a

mapping MS of size l,

$$MS \subseteq \{1, \ldots, |S_1|\} \times \{1, \ldots, |S_2|\}$$

i.e., there is an APCS with length l if and only if the graph G has an independent set of size l.

"$\Longrightarrow$": Assume that there is an APCS with length l, then there is a mapping MS of size l for (S_1, A_1) and (S_2, A_2),

$$MS = \{\, (j_1, j_2) \mid S_1[j_1] = S_2[j_2], \lceil j_1/c \rceil = \lceil j_2/c \rceil \,\}.$$

For each element (j_1, j_2) in MS, there is a vertex v_{j_1,j_2} in the graph G. We claim that these vertices form an independent set. To prove this, we show that its opposite is not correct. If the set of these vertices is not an independent set, there are at least two vertices joined by an edge. Assume that there is an edge between vertices v_{j_1,j_2} and v_{j_3,j_4}. From the construction above, one of following cases must hold for the two matches $(j_1, j_2) \in MS$ and $(j_3, j_4) \in MS$:

- $j_1 = j_3$ or $j_2 = j_4$ but not both. In this case, these two matches cannot be both in MS, because they violate the property that the matching is one-to-one.

- $(j_1 < j_3) \wedge (j_2 > j_4)$ or $(j_1 > j_3) \wedge (j_2 < j_4)$. Then they violate the order of the subsequence, and thus, they cannot be both in MS.

- There is an arc between (j_1, j_3) or between (j_2, j_4), but not both. If this holds true, the mapping is not arc-preserving, because there is only one arc between two base matches.

Consequently, the two vertices cannot be connected by an edge. Hence, the vertex set $\{v_{j_1,j_2} | (j_1, j_2) \in MS\}$ is an independent set of G and its size is l.

"$\Longleftarrow$": Assume that there is an independent set V' of size l of G, we have a mapping T of size l, i.e., $T = \{\, (j_1, j_2) \mid v_{j_1,j_2} \in V' \,\}$. Because a vertex of G is created only if the two bases of the positions match, we have $S_1[j_1] = S_2[j_2]$, and both bases are in the same fragment, so each element $(j_1, j_2) \in T$ represents

a base match of S_1 and S_2. Then, T induces a common subsequence.
Since a pair of vertices v_{j_1,j_2} and v_{j_3,j_4} in V' is not linked by an edge, the matches $(j_1, j_2) \in T$ and $(j_3, j_4) \in T$ cannot fit one of the three situations. This means that $j_1 \neq j_3$ and $j_2 \neq j_4$ and T is an one-to-one matching. Furthermore, they preserve the order of subsequence, i.e., $j_1 < j_3 \Longleftrightarrow j_2 < j_4$. They preserve the arcs too, i.e., there can be arcs between both (j_1, j_2) and (j_3, j_4) or there is no arc between both. Thus, the sequence induced by T is an APCS of (S_1, A_1) and (S_2, A_2) and its length is l. □

Theorem 5.2 *The* c-FRAGMENT **LAPCS**(CROSSING, CROSSING) *problem, parameterized by the length l of the desired subsequence, is fixed-parameter tractable and can be solved in time $O((B+1)^l B^2 + c^3 n)$, where $B = c^2 + 2c - 1$.*

Proof. The problem MAXIS-B has a straight bounded search tree FPT-algorithm (see Lemma 3.8) and c-FRAGMENT **LAPCS**(CROSSING, CROSSING), parameterized by the length l of the desired subsequence, is fixed-parameter reducible to MAXIS-B in time $O(c^3 n)$. The resulting graph has cn vertices and a bounded degree $B = c^2 + 2c - 1$. Thus, c-FRAGMENT **LAPCS**(CROSSING, CROSSING) is also fixed-parameter tractable and solvable in time $O((B+1)^l B^2 + c^3 n)$. □

5.2 c-DIAGONAL LAPCS(CROSSING, CROSSING)

The reduction in the last section can be easily extended to a reduction from c-DIAGONAL **LAPCS**(CROSSING, CROSSING) to **MAXIS-B**.
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. We put an edge $\{v_{i_1,j_2}, v_{i_2,j_2}\}$ iff the corresponding matches $\langle i_1, j_1\rangle$ and $\langle i_2, j_2\rangle$ (1) share a common base, (2) are not order-preserving, or (3) are not arc-preserving. Figure 5.2 illustrates an example of the extended reduction.

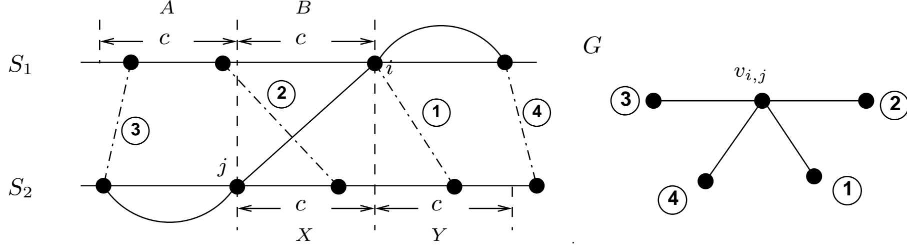

Figure 5.2: c-diagonal **LAPCS**

The vertex $v_{i,j}$ is created for the base match $\langle i, j\rangle$. The dashed lines 1, 2, 3 and 4 represent other four base matches, each of them corresponds a vertex in G. Since the base match 1 shares with the base $S_1[i]$ with $\langle i, j\rangle$, an edge is imposed between vertex $v_{i,j}$ and vertex 1. The 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 $\langle i, j\rangle$ can preserve the arc of the sequences. Two edges are added to the graph G. Note that A, B, X and Y are all substrings of length c.

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. In Figure 5.2, base match $\langle i, j\rangle$ share with base matches, which are between $S_1[i]$ and bases in substrings X and Y or between $S_2[j]$ and bases in substrings A and B. Since all these substrings have the length c, there can be at most $4c$ such bases matches. One example is the base match denoted by 1.

- We can observe in Figure 5.2 that a vertex in G has a maximum number of edges imposed due to the second situation, if the distance between the bases involved in its corresponding base match is equal to c. Consider, e.g., the base match $\langle i, j\rangle$ in Figure 5.2. There, a base matches crossing $\langle i, j\rangle$ must be from one of the following sets: $M_1 = \{\, \langle i_1, j_1\rangle \mid S_1[i_1]$ is in substring B, $S_2[j_1]$ is in substring $X \,\}$, $M_2 = \{\, \langle i_2, j_2\rangle \mid S_1[i_2]$ is in substring B, $S_2[j_2]$ is in substring Y, and $j_2 - i_2 \leq c \,\}$ and $M_3 = \{\, \langle i_3, j_3\rangle \mid$

$S_1[i_3]$ is in substring A, $S_2[j_3]$ is in substring X, and $j_3 - i_3 \leq c\}$. The set M_1 can have at most c^2 elements. The 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 Figure 5.2, then this base match cannot be in an arc-preserving match with base matches, which involve only one of the other endpoints of the arcs. Two such base matches are marked in Figure 5.2 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 $O(c^3 n)$ edges. The construction of G can be done in time $O(c^3 n)$.

Proof of correctness of the reduction works analogously to the one shown in Section 3.1.

Theorem 5.3 *The* c-DIAGONAL **LAPCS**(CROSSING, CROSSING) *problem, parameterized by the length* l *of the desired subsequence, is fixed-parameter tractable and can be solved in time* $O((B + 1)^l B^2 + c^3 n)$*, where* $B = 2c^2 + 7c + 2$.

Proof. Analogous to the proof of Theorem 5.1. □

5.3 c-FRAGMENT(c-DIAGONAL) LAPCS(UNLIMITED, UNLIMITED)

Note that the fact that the graph $G = (V, E)$ constructed in the previous sections 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 we use the so-called "degree of a sequence" as an additional parameter, we can upper-bound 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 start or end in a base in S. Clearly, the *cutwidth* (see Definition 4.4) of an arc-annotated sequence is an upper bound on the *degree*. The amount of vertices in the resulting graph G is not changed, but the bounded degree is changed. In the construction for the arc structure (CROSSING, CROSSING), we have added three groups of edges to G. Since the first two groups have nothing to do with arcs, these edges remain in the graph for UNLIMITED arc structure. Due to the third situation, $2c$ edges for c-FRAGMENT and $4c + 2$ edges for c-DIAGONAL are added 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 group of edges must be extended to include the edges between $v_{i,j}$ and all vertices, which correspond to base matches involving one of $S_1[2], \ldots, S_1[b], S_2[2], \ldots, S_2[b]$. 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 $B = 2c^2 + (4b + 3)c + 2b$. The construction can be carried out in time $O((c^3 + bc^2)n)$. Thus, c-FRAGMENT and c-DIAGONAL **LAPCS**(UNLIMITED, UNLIMITED) is also fixed-parameter tractable, when the parameters are the length l of the desired subsequence and the maximum *degree* b of the two sequences; they can be solved in time $O((B + 1)^l B^2 + (c^3 + 2bc^2)n)$, where $B = c^2 + 2bc - 1$, and in time $O((B' + 1)^l B'^2 + (c^3 + 2bc^2)n)$, where $B' = 2c^2 + (4b + 3)c + 2b$, respectively.

Chapter 6

An Algorithm for LAPCS(NESTED, NESTED)

In this chapter, 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 $S - S[i]$. 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 deter-

mine 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 a *longest arc-preserving common subsequence (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.

If $|S_1| > 0$ and $|S_2| = 0$, then */* ...but not the other. */*

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:

- LAPCS$(S_1[2, n_1], S_2, k_1 - 1, k_2)$ */* delete $S_1[1]$ */*
- 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 $1 \dotplus$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:

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

/ Since there is an arc in one sequence only, $S_1[1]$ and $S_2[1]$ can be matched only if $S_1[i]$ and arc $(1, i)$ 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:

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

/ Since the arcs cannot be matched, $S_1[1]$ and $S_2[1]$ can be matched only if $S_1[i]$, $S_2[j]$, and the arcs 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
$i \dotplus \text{LAPCS}(S_1[i+1, n_1], S_2[j+1, n_2], k_1, k_2)$.

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

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 $S_2[1]$. */*,
- $1 \dotplus \text{LAPCS}(S_1 - S_1[i], S_2 - S_2[j], k_1 - 1, k_2 - 1)$
  / 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 analysis

of running time, 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:

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

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):

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

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

of length l', 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 6.1 *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.*[1] *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.

[1] 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 corresponding branching vectors: Suppose that in one search tree node with current sequences S_1, S_2 and parameters k'_1, k'_2, we have q branches. Moreover, suppose that in branch t, $1 \leq t \leq q$, we call LAPCS with new parameter values $k'_{1,t}$ and $k'_{2,t}$. Then, the branching vector for this branch is given by $p = (p_1, \dots, p_q)$, where $p_t := (k'_1 + k'_2) - (k'_{1,t} + k'_{2,t})$. Assuming that all branchings in the search tree had branching vector p, we can compute a basis c_p which yields an upper bound on the search tree size of the form $c_p^{k'_1+k'_2}$. 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 basis for these branching vectors is given for $p = (1, 1, 1)$ with $c_p = 3 \leq 3.31$.

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 $p = (1, 1, 2, 1)$ which corresponds to a basis $c_p = 3.31$. 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 $3.31^{k'_{1,2}+k'_{2,2}}$.

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

$$3.31^{k'_{1,1}+k'_{2,1}} + 3.31^{k'_{1,2}+k'_{2,2}}. \tag{6.1}$$

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,

$$3.31^{k'_{1,1}+k'_{2,1}} + 3.31^{k'_{1,2}+k'_{2,2}} \leq 3.31^{k'_1+k'_2-1}. \tag{6.2}$$

Inequality (6.2) holds true since, by assumption, $k'_{1,1} + k'_{2,1}, k'_{1,2} + k'_{2,2} \geq 2$. Plugging Inequality (6.2) in Expression (6.1) 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 as branching vector $p = (1, 1, 2, 1)$. The corresponding basis c_p of this branching vector is 3.31, which again is the worst case basis 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 6.2 *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*.*

Chapter 7

Arc-Preserving Subsequence Problems

In this chapter, we deal with the **Arc-Preserving Subsequence** problem (**APS**). Such a problem can be encountered when we search a certain RNA pattern in an RNA database, or when one of the two parameters of the algorithm in the previous chapter is equal to zero. This problem is NP-complete, whenever one of the input sequences has an UNLIMITED arc structure or the arc structures of both sequences are CROSSING [11].Up to our knowledge, the complexity of this problem for arc structures (CROSSING, NESTED), (CROSSING, CHAIN), (CROSSING, PLAIN) and (NESTED, NESTED) seemly has not been investigated prior to this work. In Section 7.1, we show that **APS**(CROSSING, CHAIN) is NP-hard. This implies that **APS**(CROSSING, NESTED) is NP-hard, too. Section 7.2 is concerned with an algorithm which can solve **APS**(NESTED, NESTED) in polynomial time.

7.1 NP-Hardness of APS(CROSSING, CHAIN)

The NP-hardness of **LAPCS**(CROSSING, CROSSING) can be shown by a reduction form CLIQUE [11]. We use a similar construction to find a reduction from INDEPENDENT SET to **APS**(CROSSING, CHAIN). From a given INDEPENDENT SET instance, we construct an instance for the **APS**(CROSSING, CHAIN) prob-

lem consisting of two arc-annotated sequences, S_1 and S_2, where S_2 represents the graph and S_1 represents an independent set of size k. Since the information about the edges in the graph must also be incorporated in S_2, we use a fragment of S_2[1] with length equal to the number of the vertices of the graph in order to encode a vertex and use arcs between fragments in order to encode the edges. A similar concept can be used to build S_1, but in S_1, there is no arc between fragments, because it represents an independent set. Then, the question, whether the graph has an independent set of size k, can be transformed to the question, whether S_1 is an arc-preserving subsequence of S_2.

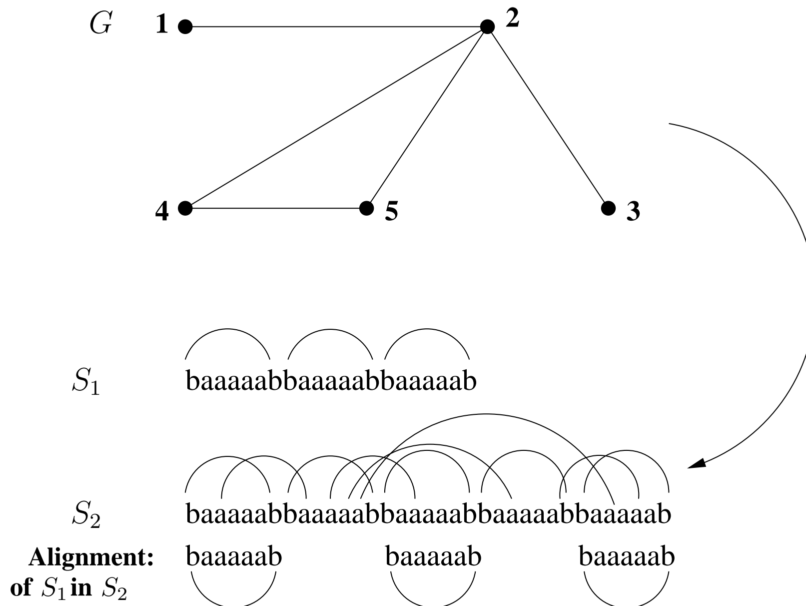

Figure 7.1: **APS**(CROSSING,CHAIN)

A small example of this construction is illustrated in Figure 7.1. The graph G in Figure 7.1 has 5 vertices and we want to know if G has an independent set of size 3, i.e., $n = 5$ and $k = 3$. We construct the sequence S_2 with five fragments, each fragment has five a symbols and two b symbols. Two b symbols of the same fragment are joined by an arc. They are used to separate fragments from each other. Each edge in G is represented by an arc between two a symbols from

[1]Note that we use *fragment of* S_2 here to denote a substring of S_2 with the arcs of S_2 between two bases in this substring.

two different fragments in S_2. For example, edge $\{1, 2\}$ has a corresponding arc between the second a in the **first** fragment and the first a in the **second** fragment. Sequence S_1, which should represent a graph with k vertices and no edge, has only three such fragments and none of the symbols a in S_1 is an endpoint of an arc. The alignment of S_1 in S_2 indicates that the three fragments of S_1 can be matched to the first, the third and the fifth fragment of S_2. Therefore, the graph G has an independent set of size three, namely $\{1, 3, 5\}$. Note that S_1 has a CHAIN arc structure and S_2 has a CROSSING arc structure.

In the following, we give a more detailed description of the construction. Given a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, we first construct the shorter sequence S_1 with CHAIN arc structure representing an independent set of size k in the same way as illustrated in Figure 7.1. For each vertex in the independent set, we create a fragment in S_1, which consists of n symbols a and two symbols b. S_1 has length $k(n + 2)$. The two b symbols build the beginning and the end of the fragment. We join the pair of b's with an arc. Because we want to encode an independent set by S_1 and no two vertices in an independent set are joined by an edge, there is no arc between the a's in S_1. Therefore, S_1 has a CHAIN arc structure. Next, we construct the second sequence S_2 to denote G in the similar way. For each vertex in V, we create again a fragment of n symbols a and two symbols b. The length of S_2 is $n(n + 2)$. The two b's are again joined by an arc. If there is an edge between two vertices v_i and v_j in G, we impose an arc between the ith and the jth fragment in S_2. This arc, which represents the edge between v_i and v_j, starts at the jth a in the ith fragment and ends at the ith a in the jth fragment. Since the arcs between two a's are between two different fragments, they definitively cross the arcs between b's of these fragments. Thus, the arc structure of S_2 is CROSSING.

The construction above can be formally described by:

$$S_2 = (ba^n b)^n,$$

$$A_2 = \big\{((i-1)(n+2)+1, i(n+2)) \mid v_i \in V\big\}$$

$$\cup \quad \big\{((i-1)(n+2)+j+1, (j-1)(n+2)+i+1) \mid \{v_i, v_j\} \in E\big\};$$

$$S_1 = (ba^n b)^k,$$

$$A_1 = \{((i-1)(n+2)+1, i(n+2)) \mid v_i \in V\}.$$

Based on the construction, there are only two types of base matches, one is the matching between symbols a, the other one is the matching between symbols b. In order to verify if S_1 is an arc-preserving subsequence of S_2, we observe that a fragment in S_1 can only be matched to an entire fragment in S_2 because of the arc between the two b symbols, which indicates the beginning and the end of the fragment. If there are no edges between vertices in G, which induces that S_2 has no arc between symbols a, then S_1 definitively is an arc-preserving subsequence of S_2. If G has some edges and an independent set I of size k, where $k < n$, S_1 has length $k(n+2)$. According to the construction above, S_2 has then arcs between two symbols a, one of which is in the fragments, which represent the vertices in I, and the other of which is in the fragments, which represent the vertices in $V \setminus I$. Nevertheless, no arcs in S_2 are imposed between two a's in the same fragment or in two different fragments, which correspond two vertices in I. Thus, by deleting the fragments, which correspond to vertices in $V \setminus I$, S_2 also has no arcs between two symbols a. Now, we have two identical sequences and can affirm that S_1 is an arc-preserving subsequence of S_2. Thus, this construction can reduce INDEPENDENT SET to **APS**(CROSSING, CHAIN). Since the length of the sequence is bounded by a polynomial in n, this construction can be done in polynomial time.

Lemma 7.1 INDEPENDENT SET *is polynomially reducible to* **APS**(CROSSING, CHAIN).

Proof. We have shown above that INDEPENDENT SET can be transformed to **APS** (CROSSING, CHAIN) and the construction works in polynomial time. The only thing that we must prove is that the graph $G = (V, E)$ has an independent set of size k if and only if S_1 is an arc-preserving subsequence of S_2.

$\Longrightarrow$: Let $V' \subseteq V$ be an independent set in G of size k. Each vertex $u \in V'$ corresponds to a fragment of S_2. We match each of the fragments, which correspond to the vertices in V', to an entire fragment of S_1 and denote this matching

as MS. Because there is no arc between two symbols a in two such fragments of S_2, we only have arcs between two symbols b from the same fragment and no arcs between symbols a in MS. Thus, MS is an arc-preserving matching. And there are exactly k such fragments in S_2. Hence, S_1 is an arc-preserving subsequence of S_2.

$\Longleftarrow$: Assume S_1 an arc-preserving subsequence of S_2. The linked pairs of symbols b in both sequences enforce the matching of symbols which come from only k fragments of S_2. Since all fragments in S_1 are not linked with each other and S_1 is an arc-preserving subsequence of S_2, there is also no arc between the k selected fragments of S_2. Since, according to the construction, every edge in G results in an arc linking two corresponding fragments in S_2, the vertices in G which correspond to these k selected segments in S_2 cannot be joined by edges. Hence, these k vertices form an independent set of G. □

Theorem 7.2 ***APS**(CROSSING, CHAIN) is NP-complete.*

Proof. NP-completeness of **APS**(CROSSING, CHAIN) can be directly followed from Lemma 7.1 and the fact that INDEPENDENT SET is NP-complete. □

The NP-completeness result for **APS**(CROSSING, CHAIN) implies that the **APS** problem for arc structure (CROSSING, NESTED) is also NP-complete, which answers two open questions in Table 4.6.

7.2 APS(NESTED, NESTED)

In this section, we investigate the **APS** problem for the arc structure (NESTED, NESTED) and describe an algorithm which solves this problem in time $O(n^4m)$, where n is the length of the shorter sequence and m is the length of the longer sequence. Note that, in the fixed-parameter algorithm for the NP-complete **LAPCS**(NESTED, NESTED) in Chapter 6, we have an **APS**(NESTED, NESTED) problem, if one of the parameters becomes 0. Using this polynomial time algorithm, the size of the search tree can be significantly reduced, because there is no need to make a branching at search tree nodes with one parameter equal to 0.

Assume that we have an instance of **APS**(NESTED, NESTED), (S_1, A_1) and (S_2, A_2). The sequences are over some alphabet Σ. We denote the length of S_1 and S_2 by n and m, respectively, and we assume that S_1 is the shorter sequence, i.e., $n \leq m$. As shown in [17], the problems **LAPCS**(NESTED, CHAIN) and **LAPCS**(NESTED, PLAIN) can be solved by a dynamic programming algorithm in time $O(n^3 m)$. Since the **APS** problem is easier than the **LAPCS** problem, we can also use this algorithm to solve the problems **APS**(NESTED, CHAIN) and **APS**(NESTED, PLAIN) in polynomial time. With NESTED arc structure of both sequences, we use the algorithm from [17] recursively from the inner arcs to the outer arcs in S_1. We want to find, for all arcs in S_1, all possible matching arcs in S_2 and mark these possible candidates for the arcs in S_1 with some new letters, which are not in Σ. These markings enable us to treat the arcs in S_1 and the substrings inside these arcs as black boxes, thereby the NESTED arc structure of S_1 can be resolved into a CHAIN structure. At the end, we have an instance of **APS**(NESTED, CHAIN), which the dynamic programming algorithm from [17] for **LAPCS**(NESTED, CHAIN) can solve in polynomial time.

Before giving a formal description of the algorithm in detail, we define some notation employed. As mentioned above, we will use the dynamic programming algorithm from [17] to determine whether the sequence inside an arc of S_1 is an arc-preserving subsequence of the sequence inside an arc of S_2. We call this algorithm **DPA**. The cutwidth of S_1 is denoted by d (for cutwidth see Definition 3.22). We can divide the arc set A_1 of S_1 into d subsets $A_1^1, \ldots, A_1^d$. A_1^1 includes the arcs which are not inside any other arcs. The arcs in A_1^t are directly inside the arcs from A_1^{t-1}, which means that there are no arcs, that are between an arc from A_1^{t-1} and an arc from A_1^t in the NESTED arc structure. It can be easily seen that the arcs from the same subset can form at most a CHAIN arc structure and that each arc in A_1 can belong to only one of the d subsets. If we use k to denote the size of A_1 and k_i to denote the size of a subset A_1^i, then we have $k = \sum_{i=1}^{d} k_i$. In order to identify the arcs of S_1, we use a set of symbols to denote the arcs in A_1. This set has k symbols, a_1, a_2, ..., a_k. Because the subsets of A_1 have no common element, we can assume that a

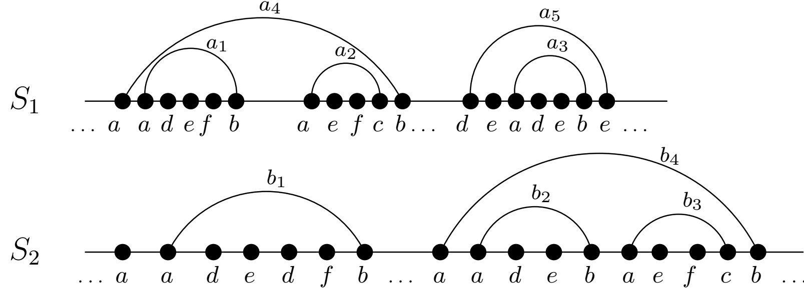

Figure 7.2: An instance of **APS**(NESTED, NESTED).

S_1 has 5 arcs, denoted by symbols a_1, a_2, a_3, a_4, and a_5. The cutwidth of S_1 is 2. Hence, the arcs can be divided into 2 groups. The first group has 2 arcs, a_4 and a_5, which are not inside any arcs. We call these arcs external arcs. The other three arcs, a_1, a_2, and a_3, which are inside a_4 or a_5, form the second group. They are called internal arcs. The arcs a_1 and a_3 can be matched to arcs b_1, b_2, and b_4, where b_2 is inside b_4. However, the matches between a_1 or a_3 and b_2 are more advantageous for other bases of S_1 than the matches between a_1 or a_3 and b_4, because more bases of S_2 are left to be matched to other bases of S_1. For example, the match between a_1 and b_4 excludes the possible match between arcs a_2 and b_3, while the match between a_1 and b_2 does not. Therefore, we ignore the arc matches between a_1 or a_3 and b_4. The arc a_2 can only be matched to arc b_3.

subset A_1^i is given by $A_i = \{a_{i+\delta} \mid \delta = 0, \ldots, k_i - 1\}$. In the following, we will use an example, illustrated in Figure 7.2, to explain the steps of the algorithm.

Algorithm for APS(NESTED, NESTED)

This algorithm has three phases. The first phase checks and replaces the innermost arcs, i.e., the arcs in A_1^d. The second phase uses **DPA** to process and resolve the arcs in the subsets A_1^i, $i = d-1, \ldots, 1$, until S_1 has a CHAIN arc structure. In the last phase, we then use **DPA** to verify whether the remaining S_1 is an arc-preserving subsequence of S_2.

Phase 1:

For each arc $a_{d+\delta}$ with left endpoint $S_1[i_1]$ and right endpoint $S_1[i_2]$ in A_1^d, $0 \le \delta \le k_d - 1$, we search all arcs in A_2 whose corresponding endpoints are the

same as $S_1[i_1]$ and $S_1[i_2]$ and denote the resulting arc set by $A_2^{d+\delta}$. The set $A_2^{d+\delta}$ has at least one element, otherwise S_1 cannot be an arc-preserving subsequence of S_2. Assume that arc (j_1, j_2) is in $A_2^{d+\delta}$. Because the arc structure of the sequence $S_1[i_1+1, i_2-1]$ must be PLAIN and the sequence $S_2[j_1+1, j_2-1]$ has at most NESTED arc structure, we use **DPA** to check whether $S_1[i_1+1, i_2-1]$ is an arc-preserving subsequence of $S_2[j_1+1, j_2-1]$. If the answer is negative, we delete the arc (j_1, j_2) from the set $A_2^{d+\delta}$. If the set $A_2^{d+\delta}$ is empty, after all arcs in $A_2^{d+\delta}$ have been checked, S_1 cannot be an arc-preserving subsequence of S_2, because for the arc $a_{d+\delta}$ we cannot find a matching arc in S_2. If the set is not empty, we replace $S_1[i_1, i_2]$ by an arc connecting symbols $x_{d+\delta}$ and $y_{d+\delta}$; this arc is also inserted directly outside all arcs in $A_2^{d+\delta}$. The bases $x_{d+\delta}$ and $y_{d+\delta}$ are new symbols, which do not come from Σ. For example, for an arc (j_1, j_2) in $A_2^{d+\delta}$, we insert $x_{d+\delta}$ directly before $S_2[j_1]$ and $y_{d+\delta}$ directly behind $S_2[j_2]$ and join the two new symbols $x_{d+\delta}$ and $y_{d+\delta}$ by an arc. Figure 7.3 illustrates the example after this step. Note that an arc in A_1^d can have more than one possible matching arc in S_2. If some of these possible matching arcs form a NESTED structure, i.e., one arc is inside another arc, we consider only the innermost arcs, because a match between an arc of S_1 and the innermost arcs of S_2 leaves more bases of S_2 for other bases of S_1. The arc a_1 in Figure 7.2 is an example for such arcs of S_1. It has three possible matching arcs in S_2. Two among them, b_2 and b_4, form a NESTED structure. Hence, we consider only the match between a_1 and b_2. Note that the original two sequences are changed after this phase. The innermost arcs of S_1 are replaced, together with the bases inside them, by some new arcs with endpoints not in Σ; the same new arcs are also insert into S_2 directly outside the appropriate arcs of S_2. We use S_1^d and S_2^d to denote the new sequences. After all arcs in A_1^d have been processed, we go to the second phase.

Phase 2:

In this phase, we deal with the remaining arcs of the original sequence S_1 in a recursive manner, starting with the arcs in A_1^{d-1} and continuing up to the arcs in A_1^1. Each iteration processes one subset. In the $(d-i)$th iteration,

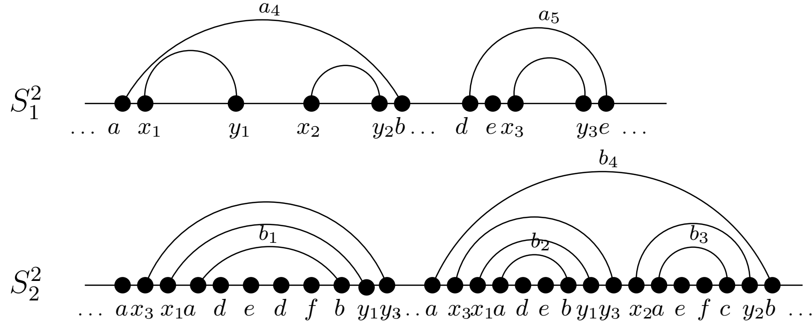

Figure 7.3: The instance after processing of the internal arcs a_1, a_2, and a_3. The arcs a_1, a_2, and a_3 and the substrings inside them are now replaced by three arcs (x_1, y_1), (x_2, y_2), and (x_3, y_3). Note that there are no bases between the endpoints of the three new arcs. The sequence S_2 is extended by the arcs (x_1, y_1), (x_2, y_2), and (x_3, y_3), which record all possible arc matches involving the internal arcs of S_1.

$1 \leq i \leq d-1$, the arcs in the subset A_1^i are processed in a similar way as the arcs in A_1^d, i.e., we search and mark all possible matching arcs in S_2 for them and delete them afterwards. However, there are two differences between the process for the arcs in these subsets and the process for the arcs in A_1^d. One is that the sequences inside the arcs in A_1^i, $i < d$, is not of PLAIN arc structure. They can have arcs with two endpoints not in Σ, which are the replacements of the arcs in A_1^{i+1} inside the arcs in A_1^i. Note that, while processing the subset A_1^i, all arcs inserted before processing A_1^{i+1} have been deleted. As we have mentioned, the arcs in A_1^{i+1} form at most a CHAIN structure. Hence, their replacements can also be of at most CHAIN structure. However, the **DPA** algorithm can be applied to arc structure (NESTED, CHAIN), too. The second difference relates to the arcs inserted into the longer sequence as markings for the arcs in A_1^{i+1}. Since all arcs in A_1^{i+1} are inside the arcs in A_1^i, their matching arcs must also be in the matching arcs of the arcs in A_1^i. While searching possible matching arcs for the arcs in A_1^i, we take into account the markings for the arcs in A_1^{i+1}, which record the possible matching arcs for the arcs in A_1^{i+1}. If we have found and marked, for each arc in A_1^i, all possibly matching arcs in A_2, then these newly inserted marking arcs represent not only that the arcs in A_1^i have a possibly

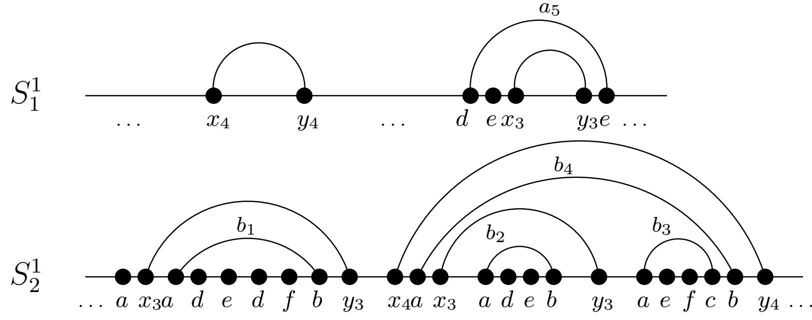

Figure 7.4: The instance after processing of the external arc a_4

In Figure 7.3, we can see that the arc a_4 can be matched to the arc b_4 in S_2 and the substring inside a_4 is of CHAIN arc structure. This is the first difference between the first phase and the second phase. We replace a_4 and the substring inside it by an arc (x_4, y_4) and add the same arc outside b_4. It is also clear that the arcs (x_1, y_1) and (x_2, y_2) will be no more needed, so we can delete them, this is the second difference.

matching arc but also that all arcs in A_1^{i+1} have possibly matching arcs inside the marking arcs. This means that, for the following iterations, we do not need the markings for the arcs in A_1^{i+1}. Hence, we can delete all arcs from S_2, which were inserted while processing A_1^{i+1}. These two differences are also illustrated in Figure 7.4.

Phase 3:

After all arcs in A_1 are processed, we have only a CHAIN arc structure in S_1 and the arc structure of S_2 throughout this algorithm is NESTED. Therefore, DPA can find out if the remaining sequence of S_1 is an arc-preserving subsequence of S_2. If it is, then the original S_1 must be also an arc-preserving subsequence of S_2.

(Note that, if we make a copy of S_2 before deleting arcs from S_2 in each iteration of the second phase, we can also back trace the positions in S_2, where a base of S_1 has a matching.)

Correctness of the algorithm:

The **DPA** algorithm can verify whether a sequence with PLAIN or CHAIN arc

structure is an arc-preserving subsequence of a sequence with NESTED sequence. Since the first phase and all iterations of the second phase have only instances of **APS**(NESTED, CHAIN) or **APS**(NESTED, PLAIN), by using **DPA**, we can find out recursively, from inner arcs to outer arcs, whether the arcs of S_1 together with the subsequences inside them are arc-preserving subsequences of some subsequences of S_2. After **DPA** gives positive answers for all arcs in A_1^1, we treat then S_1 entirely. By using **DPA** in each phase and in each iteration, the one-to-one and order-preserving properties are checked for each subsequences of S_1 and, at the end, the entire S_1. The recursive way of processing arcs in A_1 corresponds exactly to the NESTED arc structure and guarantees that two matchings arcs in A_2 for two NESTED arcs in A_2 have also a corresponding NESTED structure. This implies the arc structure of S_1 is also preserved.

Running time analysis:
After the first phase and each iteration of the second phase, the two sequences in our instance are changed. We use then S_1^d to denote the first sequence and S_2^d to denote the second sequence after the arcs in A_1^d have been processed, namely after the first phase. Similarly, S_1^i denotes the first sequence and S_2^i denotes the second sequence after the arcs in A_1^i having been processed. Further new notations employed in the running time analysis are in the following list:

- $n_{i+\delta}$: the length of the δth arc of A_1^i in sequence S_1^{i+1}. (Note that, for our purpose, the length of an arc does not include the endpoints of this arc. For example, an arc $a_{i+\delta}$ with endpoints $S_1^{i+1}[i_1]$ and $S_1^{i+1}[i_2]$ has length of $i_2 - i_1 - 1$.)

- $A_2^{i+\delta}$: the set of arcs in A_2, which are possible matching arcs for the arc $a_{i+\delta}$.

- $l_{i+\delta}$: the size of the set $A_2^{i+\delta}$.

- $b_{i+\delta}^{(j)}$: the jth arc in the set $A_2^{i+\delta}$.

- $m_{i+\delta}^{(j)}$: the length of the arc $b_{i+\delta}^{(j)}$ in sequence S_2^{i+1}.

Phase 1:

In this phase, we use **DPA** algorithm to check whether the sequences inside the innermost arcs in A_1^d are arc-preserving subsequences of the sequences inside arcs of S_2. Because of its dynamic programming property, DPA needs at most $O((n_{d+\delta})^3 m_{d+\delta}^{(j)})$ time to find a matching arc $b_{d+\delta}^{(j)}$ in S_2 for an arc $a_{d+\delta}$ in A_1^d. Since $0 \leq \delta \leq k_d - 1$ and $1 \leq j \leq l_{d+\delta}$, the total time for the first phase sums to

$$\sum_{\delta=0}^{k_d-1} \sum_{j=1}^{l_{d+\delta}} O((n_{d+\delta})^3 m_{d+\delta}^{(j)}).$$

Phase 2:

Actually, this phase consists of $(d-1)$ iterations of the first phase with the two mentioned differences. The first difference, that the subsequence to be checked in this phase can have CHAIN structure, does not affect the running time of the **DPA** algorithm, because **DPA** works both for (NESTED, CHAIN) and (NESTED, PLAIN) in time $O(n^3 m)$. However, by inserting new arcs in the second sequence in each iteration, the length of S_2 is increased by the amount of the endpoints of the new arcs. We must upper-bound this amount, otherwise we cannot have a polynomial time algorithm at the end. The second difference, which is deleting the marking arcs from previous iterations, can help us to have such an upper bound. In the first iteration, i.e., for the subset A_1^{d-1}, the two sequences have the following lengths:

$$\left|S_1^d\right| = n - \sum_{\delta=0}^{k_d-1} n_{d+\delta},$$

$$\left|S_2^d\right| = m + 2\sum_{\delta=0}^{k_d-1} l_{d+\delta}.$$

Since $k_d \leq n$ and $l_{d+\delta} \leq m$, we obtain $\left|S_2^d\right| \leq m + 2nm$. At the end of the ith iteration, which processes the subset A_1^{d-i}, the arcs inserted into the second sequence in the $(i-1)$th iteration are deleted. There are no new arcs in the second sequence, which are not in S_2, and thus, the second sequence is now the same as the original sequence S_2. After new arcs, that represent the matching possibilities for arcs in A_1^{d-i}, are inserted into the second sequence, we have now S_2^{d-i}, whose length is the same as the sum of $|S_2|$ and the amount of the

endpoints of the new arcs inserted in this iteration:

$$\left|S_2^{d-i}\right| = m + 2\sum_{\delta=0}^{k_{d-i}-1} l_{d-i+\delta} \leq m + 2nm.$$

The upper bound for the length of S_2^{d-i} is the same as the one for S_2^d. And the first sequence becomes shorter after each iteration:

$$\left|S_1^{d-i}\right| = \left|S_1^{d-i+1}\right| - \sum_{\delta=0}^{k_{d-i}-1} n_{d-i+\delta}.$$

For an arc $a_{d-i+\delta}$ in A_1^{d-i}, $0 \leq \delta \leq k_{d-i}$, the **DPA** algorithm needs time

$$\sum_{j=1}^{l_{d-i+\delta}} O((n_{d-i+\delta})^3 m_{d-i+\delta}^{(j)})$$

to find all possible matching arcs. Note that $n_{d-i+\delta}$ and $m_{d-i+\delta}^{(j)}$ are the length of arc $a_{d-i+\delta}$ and arc $b_{d-i+\delta}^{(j)}$ in the sequences S_1^{d-i+1} and S_2^{d-i+1}. Hence, the ith iteration takes a total time:

$$\sum_{\delta=0}^{k_{d-i}-1} \sum_{j=1}^{l_{d-i+\delta}} O(n_{d-i+\delta})^3 m_{d-i+\delta}^{(j)}).$$

The deletion of arcs inserted in the $(i-1)$th iteration and the insertion of new arcs can be done in time $O(nm)$. Putting the time for all $d-1$ iterations together, we have the total time for the second phase:

$$\sum_{i=1}^{d-1} \sum_{\delta=0}^{k_i-1} \sum_{j=1}^{l_{i+\delta}} O((n_{i+\delta})^3 m_{i+\delta}^{(j)}).$$

After all arcs in A_1^1 having been processed, we have two sequences with the following lengths:

$$\left|S_1^1\right| = \left|S_1^2\right| - \sum_{\delta=0}^{k_1-1} n_{1+\delta},$$

$$\left|S_2^1\right| = m + 2\sum_{\delta=0}^{k_1-1} l_{1+\delta} \leq m + 2nm$$

for the third phase.

Phase 3:

In this phase, we do not have any arcs of the original S_1. Thus, we use the **DPA** algorithm only once for S_1^1 and S_2^1 and the running time is:

$$O(\left|S_1^1\right|^3 \left|S_2^1\right|).$$

The overall running time of the whole algorithm is then equal to the sum of the running time of the three phase:

$$\underbrace{\sum_{\delta=0}^{k_d-1}\sum_{j=1}^{l_{d+\delta}} O((n_{d+\delta})^3 m_{d+\delta}^{(j)})}_{1.Phase}$$

$$+\underbrace{\sum_{i=1}^{d-1}\sum_{\delta=0}^{k_i-1}\sum_{j=1}^{l_{i+\delta}} O((n_{i+\delta})^3 m_{i+\delta}^{(j)})}_{2.Phase}$$

$$+\underbrace{O(\left|S_1^1\right|^3\left|S_2^1\right|)}_{3.Phase}.$$

With following three conditions:

- regarding Phase 1:
  $\sum_{j=1}^{l_{d+\delta}} m_{d+\delta}^{(j)} \leq |S_2| = m$,
- regarding Phase 2:
  $\sum_{j=1}^{l_{i+\delta}} m_{i+\delta}^{(j)} \leq \left|S_2^{d-1-i}\right| = m + 2\sum_{\delta=0}^{k_{d-1-i}-1} l_{d-1-i+\delta} \leq m + 2nm$, for all ith iteration, $1 \leq i \leq d-1$, and
- regarding Phase 3:
  $\left|S_2^1\right| \leq m + 2nm$,

we can give an upper bound for the length of the second sequence in all phases, $m + 2nm$. The overall running time has also an upper bound as the following:

$$\underbrace{\sum_{\delta=0}^{k_d-1} O((n_{d+\delta})^3(m+2nm))}_{1.Phase}$$

$$+\underbrace{\sum_{i=1}^{d-1}\sum_{\delta=0}^{k_i-1} O((n_{i+\delta})^3(m+2nm))}_{2.Phase}$$

$$+\underbrace{O(\left|S_1^1\right|^3(m+2nm))}_{3.Phase}.$$

Because the subset $A_1^1, \ldots, A_1^d$ are disjoint and we define $n_{i+\delta}$ as its length in sequence S_1^{i+1}, we have a fourth condition:

$$\sum_{i=1}^{d} \sum_{\delta=0}^{k_i-1} n_{i+\delta} + \left|S_1^1\right| = n.$$

Therefore, we can conclude that the total running time of the algorithm cannot exceed $O(n^4 m)$. The next theorem summarizes the results in this section.

Theorem 7.3 **APS**(NESTED, NESTED) *can be solved in time* $O(n^4 m)$*, where* n *is the length of the shorter sequence and* m *is the length of the longer sequence.*

Chapter 8

Conclusions

8.1 Summary of Results

In this study, we considered the **LAPCS** problems for arc-annotated sequences with various types of arc structures, a problem motivated by biological structure comparison. The results of classical and parameterized complexity with various parameters previous to this work have been summarized in Chapter 4. The present work examined a new aspect for c-FRAGMENT and c-DIAGONAL LAPCS problem, namely the parameterized complexity with the length l of the desired subsequence as parameter. In particular, we provide an algorithm to solve c-FRAGMENT **LAPCS**(CROSSING, CROSSING) in time $O((B+1)^l B^2 + c^3 n)$, where $B = c^2 + 2c - 1$, and c-DIAGONAL **LAPCS**(CROSSING, CROSSING) in time $O((B+1)^l B^2 + c^3 n)$, where $B = 2c^2 + 7c + 2$. This indicates that these problems are fixed-parameter tractable, when none of the two sequences has an UNLIMITED arc structure. This algorithm can also be extended to solve a restricted version of c-FRAGMENT and c-DIAGONAL **LAPCS**(UNLIMITED, UNLIMITED) where the sequences both have a bounded *degree* b, it works in time $O((B+1)^l B^2 + (c^3 + 2bc^2)n)$, where $B = c^2 + 2bc - 1$, and in time $O((B'+1)^l B'^2 + (c^3 + 2bc^2)n)$, where $B' = 2c^2 + (4b+3)c + 2b$, respectively. Lin *et al.* [20] have shown that c-FRAGMENT and c-DIAGONAL **LAPCS**(NESTED, NESTED) admit a PTAS. However, our algorithm is the first fixed-parameter solution for the c-FRAGMENT and c-DIAGONAL **LAPCS**, even for the more gen-

eral arc structure, (CROSSING, CROSSING).

The parameterized complexity of **LAPCS**(NESTED, NESTED) problem with the length l of the desired subsequence as parameter is still unknown. In positive aspect, we have shown in this work that this problem is fixed-parameter tractable, when parameterized by k_1 and k_2, the number of deletions allowed in the sequences, respectively. We designed an algorithm, which finds a longest arc-preserving common subsequence in time $O(3.31^{k_1+k_2} \cdot n)$. This algorithm works efficiently for small values for k_1 and k_2 and initiates future work on the **LAPCS** problem with the length l of the desired subsequence as parameter (Note that $l = |S_1| - k_1 = |S_2| - k_2$).

The **APS** problem for arc-annotated sequences has not been discussed explicitly before this work. Because the **LAPCS** problem for many arc structures is known to be NP-complete and $W[1]$-complete, examining the exact-matching version of this problem is significant for practice. Algorithms verifying that an arc-annotated sequence is an arc-preserving subsequence of another arc-annotated sequence, can be used for pattern searching in DNA/RNA databases. In this work, we have shown that **APS** is also NP-complete, if one of the sequences has an UNLIMITED arc structure or one sequence has a CROSSING arc structure and the other sequence is of an arc structure higher than PLAIN. An algorithm was given to solve the **APS**(NESTED, NESTED) problem in time $O(n^4 m)$, where n is the length of the shorter sequence and m is the length of the longer sequence.

8.2 Future Work

The obvious next step is to optimize the algorithms in this work. For example, the algorithm for **LAPCS**(NESTED, NESTED) with k_1 and k_2 as parameters is not very efficient for large values of k_1 and k_2. The algorithm for **APS**(NESTED, NESTED) is based on the algorithm from [17], which is designed for **LAPCS**(NESTED, PLAIN) and **LAPCS**(NESTED, CHAIN) and has a running time $O(n^3 m)$. We are confident that the problems **APS**(NESTED, PLAIN) and **APS**(NESTED, CHAIN) can be solved more efficiently than the **LAPCS**

problems. If so, to get the degree of the polynomial as small as possible remains a research issue. It is also a topic of future investigations to study the practical usefulness of our algorithms by implementations and experiments. As we observed in Chapter 4, there are still many unsolved **LAPCS** problems for arc-annotated sequences. One of the most interesting points is to determine the parameterized complexity of **LAPCS**(NESTED, NESTED) when parameterized by the length of the desired subsequence. Another interesting research topic would be to examine some restricted versions of the **LAPCS** problem, for example, we can allow the resulting longest common subsequence to have some arc mismatches whose amount is fixed. Algorithms for this restricted **LAPCS** may provide a compromise between NP-hardness of **LAPCS** and the need for efficient exact solutions.

Bibliography

[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] R.A. Baeza-Yates. Searching subsequences. *Theoret. Computer Sci.*, 78:363–376, 1991.

[3] V. Bafna, S. Muthukrishnan, and R. Ravi. Comparing similarity between rna strings. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, LNCS 937:1–16, 1995.

[4] H.L. Bodlaender, R.G. Downey, M.R. Fellows, M.T. Hallet, and H.T. Wareham. Parameterized complexity analysis in computational biology. *CABIO*, 11(1):49–57, 1994.

[5] H.L. Bodlaender, R.G. Downey, M.R. Fellows, and H.T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoret. Computer Science*, 147:31–54, 1995.

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

[7] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[8] F. Corpet and B. Michot. Rnalign program: alignment of rna sequences using both primary and secondary structures. *Comput. Appl. Biosci.*, 10:389–399, 1994.

[9] R.G. Downey and M.R. Fellows. *Parameterized Complexity.* Springer-Verlag New York Inc., 1999.

[10] P. A. Evans. Finding common subsequences with arcs and pseudoknots. *Proceedings of 10th Annual Symposium on Combinatorial Pattern Matching*, LNCS 1645:270–280, 1999.

[11] P.A. Evans. *Algorithm and Complexity for Arc-Annotated Sequence Analysis.* PhD Thesis, University of Victoria, 1999.

[12] M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. *Proc. of 12th ISAAC*, LNCS 2223:291–307, 2001.

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

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

[15] D.S. Hirschberg. *The Longest Common Subsequence Problem.* PhD Thesis, Princeton University, Canada, 1975.

[16] R.W. Irving and C.B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS 644:214–229, 1992.

[17] T. Jiang, G.H. Lin, B. Ma, and K.Z. Zhang. The longest common subsequence problem for arc-annotated sequences. *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, LNCS 1848:154–165, 2000.

[18] 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. *Proc. of 32nd ACM STOC*, pages 425–434, 2000.

[19] M. Li, B. Ma, and L. Wang. *Near optimal multiple alignment within a band in polynomial time.* Cambridge University Press, 1997.

[20] G.H. Lin, Z.Z. Chen, T. Jiang, and J.J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, LNCS 2076:444–455, 2001.

[21] S.Y. Lu and K.S. Fu. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Tran. Syst.*, 8:381–389, 1978.

[22] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[23] M.V. Olson. A time to sequence. *Science*, 270:394–396, 1995.

[24] C.H. Papadimitrion. *Computational Complexity.* Addison-Wesley Publishing Company, 1994.

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

[26] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.