Fixed-Parameter Algorithms for the Consensus Analysis of Genomic Data

Dissertation

der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Jens Gramm

aus Stockach am Bodensee

Tübingen
2003

Tag der mündlichen Qualifikation: 23. Juli 2003

Dekan: Prof. Dr. Ulrich Güntzer,

1. Berichterstatter: PD Dr. Rolf Niedermeier (Universität Tübingen),

2. Berichterstatter: Prof. Dr. Daniel Huson (Universität Tübingen),

3. Berichterstatter: Prof. Dr. Benny Chor (Tel Aviv University, Israel)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Zusammenfassung

Festparameter-Algorithmen bieten einen konstruktiven Ansatz zur Lösung von kombinatorisch schwierigen, in der Regel NP-harten Problemen, der zwei Ziele berücksichtigt: innerhalb von beweisbaren Laufzeitschranken werden optimale Ergebnisse berechnet. Die entscheidende Idee ist dabei, einen oder mehrere Aspekte der Problemeingabe als Parameter der Problems aufzufassen und die kombinatorische Explosion der algorithmischen Schwierigkeit auf diese Parameter zu beschränken, so dass die Laufzeitkosten polynomiell in Bezug auf den nicht-parametrisierten Teil der Eingabe sind. Gibt es einen Festparameter-Algorithmus für ein kombinatorisches Problem, nennt man das Problem festparameter-handhabbar. Die Entwicklung von Festparameter-Algorithmen macht vor allem dann Sinn, wenn die betrachteten Parameter im Anwendungsfall nur kleine Werte annehmen. Festparameter-Algorithmen sind zu einem algorithmischen Standardwerkzeug in vielen Anwendungsbereichen geworden, unter anderem in der algorithmischen Biologie, wo in vielen Anwendungen kleine Parameterwerte beobachtet werden können. Zu den bekannten Techniken für den Entwurf von Festparameter-Algorithmen gehören unter anderem größenbeschränkte Suchbäume. In der algorithmischen Biologie gibt es bislang nur wenige Beispiele für die Anwendung von größenbeschränkten Suchbäumen.

Diese Arbeit untersucht den Einsatz größenbeschränkter Suchbäume für NP-harte Konsens-Probleme in der Analyse von DNS- und RNS-Daten. Wir betrachten Konsens-Probleme in der Analyse von DNS-Sequenzdaten, in der Analyse von sogenannten Quartettdaten zur Erstellung von phylogenetischen Hypothesen, in der Analyse von Daten über die Anordnung von Genen und beim Vergleich von RNS-Strukturdaten. In allen Fällen stellen wir neue effiziente Algorithmen vor, in denen das Paradigma der größenbeschränkten Suchbäume auf neuartige Weise realisiert wird. Auf diesem Weg zeigen wir auch Ergebnisse parametrisierter Härte, die zeigen, dass für die dabei betrachteten Probleme ein Festparameter-Algorithmus unwahrscheinlich ist. Außerdem führen wir ganzzahliges lineares Programmieren als eine neue Technik ein, um die Festparameter-Handhabbarkeit eines Problems zu zeigen. Die Mehrzahl der hier vorgestellten Algorithmen wurde implementiert und auf Anwendungsdaten getestet. Im folgenden geben wir einen Überblick über die einzelnen Kapitel dieser Arbeit.

In Kapitel 1 geben wir eine Einführung und eine Motivation für die Fragestellungen der Arbeit, indem wir überblicksartig den Schnittbereich von parametrisierten Algorithmen und algorithmischer Biologie vorstellen, in dem die Arbeit angesiedelt ist. Weiter geben wir eine Übersicht über die Ziele der Arbeit, führen Terminologie ein, die in den weiteren Teilen der Arbeit relevant ist, und geben einen zusammmenfassenden Überblick über den Inhalt der Arbeit.

In Kapitel 2 geben wir einen Einblick in biologisches Hintergrundwissen, das für diese Arbeit von Interesse ist. Das Kapitel soll ein sehr grundlegendes

Verständnis einiger biologischer Prozesse vermitteln und biologische Terminologie einführen, die in dieser Arbeit benutzt wird. Dabei weisen wir insbesondere auf die biologischen Fragestellungen hin, durch die die in der Arbeit angesprochenen kombinatorischen Probleme motiviert sind. Wir behandeln DNS- und RNS-Sequenzen, DNS-Signale und Genexpression, RNS-Struktur, die Polymerase-Kettenreaktion, Genomevolution und die Erstellung von Hypothesen über evolutionäre Verwandtschaften.

In Kapitel 3 geben wir eine Einführung in die Teile der parametrisierten Komplexitätstheorie, die für diese Arbeit relevant sind. Wir führen formal Festparameter-Handhabbarkeit und Festparameter-Algorithmen ein und, in begrenztem Umfang, auch das Konzept der parametrisierten Härte. Wir stellen einige Techniken vor, um Festparameter-Algorithmen zu entwerfen: Problemkerne, Enumeration und dynamisches Programmieren. Jede dieser Techniken sowie das Konzept parametrisierter Härte illustrieren wir durch Beispielprobleme aus der algorithmischen Biologie, für die entsprechende Resultate angeführt werden.

In Kapitel 4 beschreiben wir den Entwurf von Algorithmen nach dem Prinzip größenbeschränkter Suchbäume. Wir beleuchten auf systematische Weise verschiedene Aspekte größenbeschränkter Suchbäume und illustrieren unsere Erklärungen durch einen einfachen Suchbaum-Algorithmus für das Vertex Cover Problem.

In Kapitel 5 ermitteln wir die parametrisierte Komplexität von NP-vollständigen Problemen bei der Suche nach Konsens-Zeichenketten, einer zentralen Frage im Kontext der Analyse von DNS-Sequenzen. Die hier untersuchten Probleme sind Closest String, Closest Substring und Varianten von diesen. Closest String ist, gegeben k Zeichenketten der Länge L und eine nicht-negative ganze Zahl d, die Frage, ob es eine "Konsens-Zeichenkette" der Länge L gibt, deren Hamming-Abstand zu jeder der Eingabezeichenketten höchstens d ist. In Abschnitt 5.1 zeigen wir, dass das Problem sowohl festparameter-handhabbar ist, wenn wir den Distanzparameter d betrachten als auch wenn wir die Zahl k von Eingabezeichenketten als Parameter betrachten. Um die Festparameter-Handhabbarkeit bezüglich des Distanzparameters zu zeigen, präsentieren wir einen Suchbaum-Algorithmus mit $O(d^d \cdot dk + Lk)$ Laufzeit, den wir auf einige in praktischen Anwendungen relevante Varianten des Problems verallgemeinern. Die Festparameter-Handhabbarkeit bezüglich der Zahl k von Eingabezeichenketten zeigen wir durch Formulierung der Fragestellung als ganzzahlig lineares Programm mit einer nur von k abhängigen Zahl von Variablen. Dies ist das erste Beispiel, bei dem die Festparameter-Handhabbarkeit eines Problems auf diese Weise gezeigt wurde. Wir diskutieren die Anwendung der vorgestellten Algorithmen für Anwendungsfragen in der Motivsuche und beim Entwurf von Primern und stellen experimentelle Ergebnisse sowohl für den Suchbaum-Algorithmus als auch für die Lösung über ganzzahlig lineare Programme vor.

Das Closest Substring Problem verallgemeinert Closest String dadurch, dass es nach Teilzeichenketten einer vorgegebenen Länge L in jeder Eingabezeichenkette frägt, die eine Konsens-Zeichenkette, wie bei Closest String beschrieben, haben. Insbesondere im Kontext der Motivsuche in DNS-Sequenzen, d.h. bei der Suche nach gemeinsamen Signalen in den Sequenzen, erfuhr diese Fragestellung große Aufmerksamkeit. Wir zeigen, dass Closest Substring und Varianten davon W[1]-hart sind in Bezug auf die Anzahl von Eingabezeichenketten als Parameter. Dieses Ergebnis gibt ein starkes theoretisch fundamentiertes Indiz für die intuitive Vermutung, dass Closest Substring kombinatorisch sehr viel schwieriger ist als Closest String. Bemerkenswerterweise wurde diese Vermutung bisher nicht durch "klassische" Methoden unterstützt, da beide Probleme NP-hart sind und beide ein Polynomzeit-Approximationsschema besitzen. Unser Resultat zeigt, dass es wenig Hoffnung auf Algorithmen für Closest Substring gibt, deren Laufzeit nur exponentiell in der Zahl der Eingabezeichenketten ist.

In Kapitel 6 betrachten wir das Problem der Berechnung eines "Konsens-Baums" für eine gegebene Menge von kleinen Bäumen, die jeweils nur vier Blätter haben. Dies ist die zentrale Idee von Quartettmethoden, einem populären Ansatz in der Erstellung von Hypothesen über Evolutionsbäume. Das von uns untersuchte Problem ist Minimum Quartet Inconsistency (MQI): Gegeben n Taxa und genau eine binäre Baumtopologie, genannt Quartett-Topologie, für jede Menge von vier der Taxa, sowie ein nicht-negativer Parameter k, ist die Frage, einen binären Baum zu finden, dessen Blättern die n Taxa in einer 1-zu-1 Relation zugeordnet sind und der höchstens k der gegebenen Quartett-Topologien "widerspricht". Während das allgemeinere Problem, bei dem nicht notwendigerweise eine Quartett-Topologie für jede Vierermenge von Taxa verlangt wird, nicht festparameter-handhabbar zu sein scheint, zeigen wir für MQI einen Festparameter-Algorithmus in Bezug auf den Parameter k: Wir präsentieren einen Suchbaum-Algorithmus mit $O(4^k \cdot n + n^4)$ Laufzeit. Dies bedeutet, dass bei einer kleinen Zahl von "Fehlern" in den Quartett-Topologien eine effiziente Berechnung des Baums möglich ist. Insbesondere stellen wir heuristische Methoden vor, um den Algorithmus zu beschleunigen. Anhand von experimentellen Ergebnissen, unter anderem bei der Analyse einiger Datensätze von Pilzspezies, demonstrieren wir die Einsetzbarkeit unseres Algorithmus.

In Kapitel 7 behandeln wir ein Konsens-Problem in der Analyse von Genanordnungen, d.h. den Anordnungen von Genen in den Genomen verschiedener Spezies. Die sogenannte Breakpoint-Distanz ist dabei ein häufig verwendetes Maß zum Vergleich von Genanordnungen. Zentral in diesem Kapitel das NP-vollständige Breakpoint Median Problem, welches danach frägt, für eine gegebene Menge von Genanordnungen ihren "optimalen Konsens" bezüglich der Breakpoint-Distanz zu berechnen. Der optimale Konsens ist hier eine Genanordnung, mit der die Summe d der Breakpoint-Distanzen zu jeder der Eingabeanordnungen minimal wird. Wir präsentieren für das Problem einen Suchbaum-Algorithmus mit $O(2.15^d + kn)$ Laufzeit, wobei k Genanordnungen jeweils der Länge n gegeben sind. Damit geben wir den ersten exakten Algorithmus für dieses Problem mit nicht-trivialer Laufzeitgarantie. Wir zeigen die praktische Einsetzbarkeit des Algorithmus durch experimentelle Ergebnisse. Wir demonstrieren, wie sich durch Kombination unseres Algorithmus mit einer Heuristik zur Erstellung phylogenetischer Bäume sogenannte Breakpoint-Phylogenien berechnen lassen, d.h., aufgrund von Genanordnungen berechnete evolutionäre Bäume, die möglichst geringe Kosten in Bezug auf die Breakpoint-Distanz haben.

In Kapitel 8 untersuchen wir ein Konsens-Problem in der Analyse von RNS-Sekundärstrukturdaten. Der Vergleich von RNS-Strukturdaten ist ein Gebiet, das in algorithmischer Hinsicht herausfordernde Fragen aufwirft. In dieser Arbeit beschäftigen wir uns mit dem Modell des NP-vollständigen Longest Common Subsequence Problems für den Vergleich von zwei Sequenzen, die zusätzlich mit sogenannten, die Struktur repräsentierenden "Arcs" versehen sind. Gegeben zwei mit Arcs versehene Sequenzen der Länge höchstens n und nicht-negative ganze Zahlen k_1 und k_2, ermittelt unser Algorithmus in $O(3.31^{k_1+k_2})$ Laufzeit, falls existent, eine längste gemeinsame Teilsequenz, die "Arc-erhaltend" ist und die durch Löschung (zusammen mit den entsprechenden Arcs) von k_1 Zeichen in der einen Sequenz und von k_2 Zeichen in der zweiten Sequenz erhalten werden kann. Diese gemeinsame Teilsequenz repräsentiert die Übereinstimmung der gegebenen Sequenzen und ist ein Maß für ihre Ähnlichkeit. Unser Algorithmus zeigt, dass das Problem festparameter-handhabbar ist in Bezug auf die Anzahl der Löschungen. Die Laufzeitanalyse für diesen Algorithmus beinhaltet neue Analysemethoden für größenbeschränkte Suchbäume. Für den speziellen Fall, dass entweder $k_1 = 0$ oder $k_2 = 0$ zeigen wir, dass sich das Problem durch dynamisches Programmieren in quadratischer Laufzeit lösen lässt. Dies bedeutet zum einen eine Beschleunigung für unseren Suchbaum-Algorithmus und erlaubt es zum anderen, eine offene Frage aus der Literatur nach einem Polynomzeit-Algorithmus für ein eng verwandtes Problem zu beantworten.

In Kapitel 9 fassen wir wichtige Ergebnisse aus den einzelnen Kapiteln zusammen, stellen sie in den Kontext aktueller Forschung bezüglich parametrisierter Algorithmen und stellen Bezüge zu anderen algorithmischen Gebieten her. So zeigen wir Zusammenhänge zum ganzzahlig linearen Programmieren, zu Approximationsalgorithmen und zu heuristischen Methoden. Da die Wahl der Problemparameter ein zentraler Punkt bei der Entwicklung parametrisierter Algorithmen ist, fassen wir diesbezügliche Beobachtungen zusammen, die wir bei den in dieser Arbeit studierten Problemen gemacht haben.

In Kapitel 10 zeigen wir mögliche Richtungen auf für auf dieser Arbeit aufbauende, zukünftige Forschung. Zum einen beschreiben wir drei Gebiete von kombinatorischen Fragestellungen im Kontext der algorithmischen Biologie, in denen die hier vorgestellten algorithmischen Techniken möglicherweise anwendbar sind. Daneben zeigen wir zwei möglicherweise interessante Richtungen für die Weiterentwicklung der algorithmischen Technik größenbeschränkter Suchbäume.

Preface

This thesis covers parts of my research on the tractability of hard combinatorial problems in molecular biology, particularly focusing on exact algorithms with optimal solutions and guaranteed bounds on the running time. Since January 2000, I have been supported by the Deutsche Forschungsgemeinschaft in the project OPAL (optimal solutions for hard problems in computational biology), NI 369/2-1/2, which was initiated and led by Rolf Niedermeier and Klaus-Jörn Lange. I thank Rolf and Klaus-Jörn for the opportunity to work with them and I am also grateful to many other people who supported me in my research: The members of the Tübingen theoretical computer science group—Jochen Alber, Henning Fernau, Jiong Guo, Rolf Niedermeier, Klaus-Jörn Lange and Klaus Reinhardt—for the perfect research enviroment they provided; Mike Fellows (University of Newcastle, Australia), Mike Hallett (McGill University, Montréal, Canada), and Peter Rossmanith (TU München, now RWTH Aachen) for common research and inviting me to research visits; Falk Hüffner for his support in the implementations; Dominik Begerow and Michael Weiß (Group for Systematic Biology and Mycology, Tübingen) for their collaboration, their help and their advice concerning biological questions; and Gerhard J. Woeginger for pointing me to the result of H. W. Lenstra (used in Section 5.1.3).

The most important partner in my research has been Rolf Niedermeier. This thesis emerged from the continuous work with him and from collaborations with Jochen Alber, Mike Fellows, Jiong Guo, and Peter Rossmanith. The focus of this work are fixed-parameter algorithms for combinatorial consensus problems in the analysis of DNA and RNA data. New results are achieved for problems in the context of consensus sequence analysis (Chapter 5), of quartet problems (Chapter 6), of the analysis of gene order information (Chapter 7), and of the analysis of RNA secondary structure (Chapter 8). For these chapters, I provide, in the following, a detailed chapterwise overview pointing out those parts of this research which have been, in particular, my own contribution.

Chapter 5: "Consensus of Sequences." Section 5.1 is based on [92] and shows, as central results, fixed-parameter tractability of the Closest String problem, both when considering an maximally allowed Hamming distance as parameter as well as when considering the number of input sequences as parameter. Regarding the distance parameter, we give an elegant and nevertheless very efficient search tree algorithm. I developed the algorithm and conducted its analysis. Regarding the number of input strings as parameter, the fixed-parameter tractability was unclear for some time. The key for finally coming up with an ILP formulation using only a bounded number of variables, was the "normalization" of the input instance that considers the input instance as a character matrix and that merges columns in the matrix which exhibit an isomorphic structure. Recognizing that it is the column structure that matters, to develop an appropriate notion of isomorphism, and to describe the normalization was, in particular, my contribution.

Section 5.2, based on [70], contains involved W-hardness proofs for the Closest Substring and Consensus Patterns problems. Especially for extending the results to a bounded size—in our case even binary—alphabet, which is the biologically relevant case, required special care. The employed reduction reduces from a graph problem and constructs gadgets that encode edges of the graph. Especially difficult parts in the binary case are a so-called "synchronization" of gadgets which prevents that a solution matches between two adjacent gadgets, and a balancing of "1"s within a gadget. In the construction of gadgets, I formulated both the "synchronization" and the balancing, and conducted the correctness proofs for our results. Moreover, I generalized the results to the—in many situations more relevant—Consensus Patterns problem.

Chapter 6: "Consensus of Quartets." Our main result in this chapter, which is based on [88, 89], is a bounded search tree algorithm for the Minimum Quartet Inconsistency problem. The key for this algorithm was to exploit the results of Bandelt and Dress [11] in order to trace inconsistencies back to local structures. Only these local inconsistencies allow the development of a bounded search tree algorithm. One of my contributions was to draw the connection between the results by Bandelt and Dress [11] and Minimum Quartet Inconsistency, and to prove the claimed upper bound on the running time for our algorithm. Moreover, I carried out the implementation of the algorithm as well as the performance tests.

Chapter 7: "Consensus of Gene Orders." This chapter is based on [90] and, among others, contains a new fixed-parameter algorithm for the Breakpoint Median problem. Here, I developed the concept of avoiding worst cases which appear in prior algorithms and conducted the proofs of running time. Further, I developed the test application in which the new algorithm was used to compute phylogenies based on gene order information.

Chapter 8: "Consensus of RNA Structures." This chapter contains parts of research presented in [2, 82], of which parts are also contained in the "Diplomarbeit" of Jiong Guo [93]. One central result presented here in an overview (for the details we refer to [93]) is a search tree algorithm for the Longest Common Subsequence Problem for nested arc structures. A bottleneck case in this algorithm (to which we focus our discussion here) is the case of an arc match by which the sequence is split into two parts, one inside the arcs and one outside the arcs. It was my contribution to develop a way for a new "amortized" analysis of both parts which, finally, led to our time bounds.

Following the standard in scientific texts, I use in the remainder of this thesis the personal pronoun "we" (except from sentences where I express my personal opinion).

Contents

Chapter 1

Introduction

Fixed-parameter algorithms offer a constructive and powerful approach to efficiently obtain solutions for NP-hard problems combining two important goals: Fixed-parameter algorithms compute *optimal* solutions within *provable time bounds* despite the (almost inevitable) computational intractability of NP-hard problems. The essential idea is to identify one or more aspects of the input to a problem as the parameters, and to confine the combinatorial explosion of computational difficulty to a function of the parameters such that the costs are polynomial in the non-parameterized part of the input. This makes especially sense for parameters which have small values in applications. Fixed-parameter algorithms have become an established algorithmic tool in a variety of application areas, among them computational biology where small values for problem parameters are often observed. A number of design techniques for fixed-parameter algorithms have been proposed and bounded search trees are one of them. In computational biology, however, examples of bounded search tree algorithms have been, so far, rare.

This thesis investigates the use of bounded search tree algorithms for consensus problems in the analysis of DNA and RNA data. More precisely, we investigate consensus problems in the contexts of sequence analysis, of quartet methods for phylogenetic reconstruction, of gene order analysis, and of RNA secondary structure comparison. In all cases, we present new efficient algorithms that incorporate the bounded search tree paradigm in novel ways. On our way, we also obtain results of parameterized hardness, showing that the respective problems are unlikely to allow for a fixed-parameter algorithm, and we introduce integer linear programs (ILP's) as a tool for classifying problems as fixed-parameter tractable, i.e., as having fixed-parameter algorithms. Most of our algorithms were implemented and tested on practical data. In the following, we give a more detailed introduction to this thesis.

1.1 Computational Biology

The exponential growth of publicly available biological data during the last ten years, e.g., generated from national and international genome projects, offers remarkable opportunities for application of modern computer science. Due to the discrete nature of genomic data, the analysis methods are mostly discrete, i.e., relying on models based on finite objects such as graphs or strings. Computational biology can be seen as the process of two interacting steps: A first step is to identify a biological question and to construct a mathematical model of the biological reality, in order to formulate the biological question as a combinatorial problem. The second step, then, is to construct algorithms in order to address the questions with the aid of computers. Computational biology became a independent field of research in the intersection of biology and computer science. Textbooks on computational biology include [94, 164, 179, 198].

Given a combinatorial model of a biological question, a main contribution of computer science lies in the design of algorithms and in their analysis with respect to the quality of the solution as well as with respect to the algorithm's time and space requirements. This is the part of computational biology this thesis focuses on: we explore the application of new algorithmic techniques to address combinatorial questions arising in the analysis of biological data. Of particular interest to us are, here, problems classified as "computationally intractable" by classical complexity measures, i.e., which have been shown NP-hard (for background on NP-completeness theory refer to [74]). NP-hard problems are encountered in many areas of computational biology. The molecular biologist Joseph Felsenstein is cited in 1998 as follows [60]: "About ten years ago, some computer scientists came by and said they heard we have some really cool problems. They showed that the problems are NP-complete and went away!" Addressing the issue raised in this quote, the goal of this thesis is the design of practical algorithms for solving NP-hard problems occurring in the analysis of DNA and RNA data to which we refer as genomic data here.

1.2 Parameterized Complexity

To cope with the intractability of computational problems, several methods have been developed, e.g., approximation algorithms [7, 101], heuristic methods [136], and randomized algorithms [143]. However, these methods have their drawbacks. Particularly in the context of computational biology, *optimal* solutions are often preferable to approximated results. For example, in the context of phylogenetic reconstruction according to the maximum parsimony criterion (see Section 2.5) it was pointed out that "because suboptimal solutions can yield very different evolutionary reconstructions, exact solutions are strongly preferred over approximate solutions" [140]. Besides this request for optimal solutions, it is still desirable to have a mathematical analysis of the algorithms leading to *performance guarantees* concerning the running time.

Parameterized complexity theory is another proposal on how to cope with computational intractability in some cases, leading to optimal solutions as well as performance guarantees (see [58] for a monograph and [3, 68, 69] for recent surveys on parameterized complexity). In a sense, so-called “fixed-parameter algorithms” form a very interesting variant of exact, exponential-time solutions mainly for NP-hard problems. Many computational problems are formulated as an *optimization* problem: given an object x, find a solution such that an optimization criterion k is maximized or minimized. E.g., the NP-complete Vertex Cover problem can be formulated in this way: Given an undirected graph $G = (V, E)$, find a vertex cover for G such that the size k of the vertex cover is minimum. Herein, a *vertex cover* for G is a subset of vertices $V' \subseteq V$ such that each edge in E has at least one of its endpoints in V'. Usually, these problems can alternatively be formulated as an *parameterized* problem: Given an object x and a non-negative integer k, does x have a solution where some specified property of the given object or of the computed solution has value k? In parameterized complexity theory, the non-negative integer k is called the *parameter*. E.g., Vertex Cover can be formulated in this way: Given a graph $G = (V, E)$ and a non-negative integer k, does G have a vertex cover of size k? As a solution, we would expect either a “no” answer or a “yes” answer supplemented with a vertex cover of at most k vertices. In this case, the parameter of the problem is the size of the vertex cover. In this work, we consider, if not indicated otherwise, the parameterized version of problems, although we also discuss the extension of our algorithms to solve the corresponding optimization version.

In many applications, the parameter k can be considered to be “small” in comparison with the size $|x|$ of the given object x. Hence, it may be of high interest to ask whether these problems have deterministic algorithms, so-called *fixed-parameter algorithms*, whose running time is exponential *only* with respect to k and polynomial with respect to $|x|$. More formally, we call the problem *fixed-parameter tractable* with respect to parameter k if it allows an algorithm with a running time of $f(k) \cdot |x|^{O(1)}$ for an arbitrary function f. An algorithm with a running time of this kind is, then, a fixed-parameter algorithm. Equally, we also say that the algorithm is fixed-parameter.

The basic observation of parameterized complexity, as mainly developed by Downey and Fellows [58], is that for many hard problems, the seemingly inherent “combinatorial explosion” can be restricted to a “small part” of the input, the parameter. So, for instance, the NP-complete Vertex Cover problem allows for an algorithm with running time $O(kn + 1.29^k)$ [43, 154], where the parameter k is the maximum size of the vertex cover set we are looking for and n is the number of vertices of the given graph. These results show that Vertex Cover is fixed-parameter tractable.

A specific type of fixed-parameter algorithms are *search tree algorithms*. Here, one explores the search space of a particular problem using a recursive algorithm which, for a given instance, generates a set of simplified instances and calls itself recursively on each of these instances. The recursion stops if either a solution is found or the algorithm can determine that the instance has no solution. The dependencies of the recursive calls can, then, be modeled by a tree which is termed *search tree.* An example of a search tree approach is the Davis-Putnam algorithm for determining the satisfiability of a boolean formula in conjunctive normal form [54]. Notably, the running time of search tree algorithms is mainly determined by the (usually exponential) search tree size, i.e., the number of their nodes, while the time spent in one node of the search tree is polynomial or even linear. Thus, we have *fixed-parameter* search tree algorithms if the worst-case size of the search tree depends only on the parameter. In this case, we refer to the algorithm as *bounded search tree.* Bounded search trees are a commonly known design technique for fixed-parameter algorithms, e.g., the mentioned algorithms for Vertex Cover [43, 154] are bounded search trees.

Many computational problems in computational biology can be formulated as parameterized problems. Moreover, computational biology provides many examples of problem parameters which are likely to be small in the applications: a bounded number of errors in measurements, the alphabet size, a bounded evolutionary distance etc. Therefore, computational biology seems a prosperous ground for the application of fixed-parameter algorithms, since they provide optimal solutions and guaranteed upper bounds on the running time. In this sense, fixed-parameter algorithms seem, as was formulated by an anonymous referee of [90], to be “laudable approaches to NP-hard problems in biology, better than approximation methods in most cases.”

Considering recent research in computational biology, we find several examples where fixed-parameter algorithms are presented, for example, in motif search [27], in the context of SNP haplotyping [170], in the analysis of genome rearrangements [98], in the analysis of gene duplications [97], or in the analysis [192] and visualization [13] of microarray data (more details concerning some of these examples can be found in Chapter 3). Notably, only few examples of bounded search tree algorithms can be found in the computational biology literature.

The parameterized complexity of problems in computational biology was also subject of dissertation projects before, and we mention two respective theses here: Hallett [96] shows parameterized hardness results for Intervalizing Colored Graphs, having applications in physical mapping, and Shortest Common Superstring, having applications in sequence assembly. Evans [64] investigates the parameterized complexity of problems in the comparison of RNA secondary structures, modeled as the Longest Arc-Preserving Common Subsequence problem for arc-annotated sequences, providing hardness results as well as fixed-parameter algorithms. In contrast to these works, which focus on single problems and on their classification in the parameterized world, this thesis addresses the design of efficient fixed-parameter algorithms, in particular *search tree algorithms*, for a *class* of problems, namely consensus problems. Besides giving worst-case bounds for the algorithms, we also discuss heuristic improvements, relevant in practice, and underline the applicability of our algorithms, in most cases, by supplementing implementations and initial tests on real data.

1.3 Consensus Problems

Consensus analysis problems occur in many computational biology settings. Given a set of input objects, e.g., strings, and a distance measure, e.g., Hamming distance, we ask for a object which is a good "consensus" of the input objects following a given criterion. In the example in which the given objects are strings, e.g., we can ask for a "consensus string" s that minimizes the sum of Hamming distances between s and each of the given strings. Consensus objects are computed, e.g., as a representative for the input objects, to find a common pattern, or as a common agreement of several measurements or predictions. Various results and algorithms were published in this context and we mention some examples as follows. Regarding the consensus of strings, an own section in the monograph [164] (Section 8.6) is dedicated to this question. As one of the most recent results, Li *et al.* [128, 129] present polynomial-time approximation schemes for several variants of the consensus string problem with respect to Hamming distance. In the context of genome rearrangements, consensus objects are computed as hypothetical ancestors for a set of genomes, each represented by its gene order. Here, several distance measures have been examined, e.g., breakpoint distance [142, 174] and reversal distance [39, 183]. In the context of phylogenetic reconstruction, the computation of a consensus for a given set of trees over the same or a partly differing species set is an important question [158]; particularly the computation of supertrees, i.e., the combination of trees with only partly overlapping species set, currently receives considerable attention [26, 157]. In RNA structure analysis, families of sequences, e.g., specific types of introns [133], can be characterized by their common structure, i.e., their "structural consensus."

Summarizing, consensus problems seem to represent a core question arising in many formulations and in many contexts within computational biology.

1.4 Notation, Conventions, Preliminaries

In this work, we assume basic familiarity with combinatorial objects such as strings, graphs in general, and trees in particular [50]. The names of combinatorial problems are written in Small Caps style (e.g., Vertex Cover) and the problems are defined, as it is common in computer science literature, in an "Input"/"Question" format.

Strings and Sequences. Since strings play a central role in this work, we point out some conventions here: As it is common in the biological context, we use the terms *string* and *sequence* interchangeably, denoting a sequence of elements from a given alphabet Σ. However, we clearly distinguish *substrings* (which denote a continuous segment of a string) from *subsequences* (which do not need to be continuous). Given a string s, we refer to the element at its ith position by $s[i]$. The most commonly used distance measure for strings is *Hamming* distance, where the Hamming distance between two strings s_i and s_j, both of length l, is given by $d_H(s_i, s_j) = \{ 1 \leq p \leq l \mid s_i[p] \neq s_j[p] \}$.

Graphs. The graphs used in this work are, in general, undirected.

"Big O"-notation. The running times given in this work are worst-case running times and we use the "O" notation: For functions $f(x)$ and $g(x)$, $f(x)$ is in $O(g(x))$ iff there are a constant c and an integer n_0 such that $f(x) \leq c \cdot g(x)$ for all $x \geq n_0$.

Decision problems. Both for the optimization version as well as for the parameterized version of a problem, one can distinguish the *decision* version, only asking for an integer value (in the optimization case) or for a yes/no answer (in the parameterized case), from the *constructive* version of the problem, which also supplies a solution object. As it is common in complexity theory, all problems in this work are formulated as decision problems. However, we implicitly address also the constructive problems and our algorithms do in all cases, if the question of the problem is answered positively, also provide solution objects.

Approximation. To give basic familiarity with terms from approximation theory which are used throughout this work, we explain some of them, thereby restricting to minimization problems. More details on approximation theory can be found, e.g., in [7, 101]. Given a minimization problem, a solution of the problem is r-*approximate*, $r \geq 1$, if the cost of the solution is k, the cost of an optimal solution if k_{opt}, and $k/k_{\text{opt}} \leq r$. An algorithm is a *factor-r approximation* if it computes r-approximate solutions. A *polynomial-time approximation scheme (PTAS)* is an algorithm that computes, for any given real $\epsilon > 0$, a $(1+\epsilon)$-approximate solution in polynomial time where ϵ is considered to be constant. Typically, PTAS's have a running time $n^{O(1/\epsilon)}$, often with large constant factors hidden in the exponent which make them infeasible already for moderate approximation ratio. Therefore, Cesati and Trevisan [41] propose the concept of an *efficient* polynomial-time approximation scheme (EPTAS) where the PTAS is required to have an $f(\epsilon) \cdot n^{O(1)}$ running time where f is an arbitrary function depending only on ϵ and not on n. Notably, most known PTAS's are *not* EPTAS's [69].

Integer Linear Programming. Throughout this work, we will refer several times to integer linear programming, which constitutes a central technique in combinatorial optimization. The underlying combinatorial problem is described as follows. A linear inequality $a_1 x_1 + a_2 x_2 + \ldots + a_m x_m \leq b$ over m variables

$x_1, x_2, \ldots, x_m$ where $\bar{a} = (a_1, a_2, \ldots, a_m)$ is an m-tuple of integers and b is an integer can be abbreviated as the pair $(\bar{a}, b)$. Given a positive integer m, an integer linear program (ILP) is given by a set X of pairs $(\bar{a}, b)$ which denote linear inequalities over m variables, an m-tuple $\bar{c}$ of integers, and an integer B. Then, Integer Linear Programming is, given an ILP over m variables, the question whether there is an m-tuple $\bar{x}$ of integers, i.e., an assignment of integer values to the variables, such that $\bar{a} \cdot \bar{x} \leq b$ for every $(\bar{a}, b) \in X$ and $\bar{c} \cdot \bar{x} \geq B$ ($\bar{a} \cdot \bar{x}$ denoting the cross product of $\bar{a}$ and $\bar{x}$). Integer Linear Programming is NP-complete [74]. For details on integer linear programming, refer to [119, 145, 176].

1.5 Goals and Achievements

The goal of this thesis is to investigate the use of bounded search tree algorithms for consensus problems in the analysis of DNA and RNA data. In its outcome, this work seems to be one of the first works

- exploring systematically the use of search tree algorithms in computational biology,
- discussing the use of fixed-parameter algorithms for consensus problems,
- putting emphasis not only on the classification of problems but onto the design of practically efficient algorithms, including heuristic enhancements, while
- also providing experimental evaluations.

Our algorithms introduce novel design and analysis techniques regarding bounded search trees. Moreover, this work provides new example problems for illustrating concepts of parameterized complexity, e.g., as class material [149]: We give several examples of how to apply the search tree paradigm in various novel ways; we give examples for several issues concerning the choice of parameters, e.g., the analysis of one problem with respect to several parameters or the use of dual parameterizations; we give examples of combining heuristic speed-up techniques with fixed-parameter algorithms; we introduce ILP's with a bounded number of variables as a technique to classify a problem as fixed-parameter tractable; further, we give novel hardness results regarding recently intensively studied motif search problems.

1.6 Structure and Overview

This section provides an overview on the remaining thesis. We describe the contents of each of the following chapters.

Chapter 2: Biological Background. We give a coarse outline of some biological background relevant in this work. This chapter is intended to give only a very basic understanding of biological processes and to introduce biological terminology that is used later on. Further, we point out the motivation for the combinatorial problems discussed in this thesis. We address DNA and RNA sequences, DNA signals and gene expression, RNA structure, polymerase chain reaction, genome evolution, and evolutionary relationships.

Chapter 3: Parameterized Complexity. We give an introduction into the theory of parameterized complexity, as far as it is relevant in this work. We formally introduce fixed-parameter tractability and, to a limited extent, the theory of parameterized intractability. We present several techniques for designing fixed-parameter tractable algorithms, namely kernelization, enumeration, and dynamic programming. We illustrate the presented concepts giving examples from recent research in computational biology.

Chapter 4: Search Tree Algorithms. We describe the concept of bounded search trees, the algorithm design technique which is most relevant in this work. We present aspects of bounded search trees in a systematic way and illustrate our explanations by easy-to-understand examples, using Vertex Cover as the running example.

Chapters 5 to 8 contain the main results of this thesis. Since we address problems from different contexts requiring different notation, the employed notation is presented in the respective chapters. In this way, each of these chapters starts with a general introduction which states the central problem(s) of the respective chapter, which gives an outline of the context in which these problems occur, and which gives an overview on previous work as well as on the new results presented here. This general introduction is, in each case, followed by a section on formal aspects of the models, notation, and conventions used in the respective chapter. The core part of each chapter contains our results. Each of the chapters concludes with a list of open questions and directions for future research. In detail, Chapters 5 to 8 contain the following results:

Chapter 5: Consensus of Sequences. This chapter explores the parameterized complexity of NP-complete problems in the analysis of DNA sequences, namely in the search for consensus strings. The addressed problems are Closest String and Closest Substring and variants thereof. Closest String is, given k length-L input strings and a non-negative integer d, the question whether we can find a length-L solution string that has Hamming distance at most d to each of the input strings. In Section 5.1, we show that Closest String can be solved in linear time both when the distance parameter d is fixed and also when the number k of input strings is fixed. More precisely, we give a bounded search tree algorithm with $O(d^d \cdot dk + Lk)$ running time that is generalized to several practical variants of the problem. We obtain the linear time algorithm for a fixed number of input strings by giving an ILP with a bounded number of variables. Notably, this is the first time that ILP's are used for classifying a problem as fixed-parameter tractable. We investigate the application of the algorithms in primer design and motif search. Further, we provide experimental evaluation both regarding the search tree algorithm as well as regarding the ILP approach.

The Closest Substring problem which generalizes Closest String by asking for substrings of the input strings that have a "closest string" has recently received a lot of attention in the area of motif search, i.e., the search for common signals in DNA sequences. We show that Closest Substring and variants thereof are W[1]-hard with respect to the number of input strings. This is the first strong theory-based support for the common intuition that Closest Substring (W[1]-hard) is more difficult than Closest String (fixed-parameter tractable); notably this could not be expressed by "classical" measures since both are NP-complete and both do have a polynomial-time approximation scheme. Our result shows that there is little hope to obtain exact algorithms for Closest Substring that are exponential in the number of input sequences only.

Chapter 6: Consensus of Quartets. This chapter explores the computation of a consensus tree for a given set of small trees, each having only four leaves. This is the central idea of quartet methods, a popular approach in the reconstruction of phylogenetic trees. Given n taxa, exactly one topology for every subset of four taxa, and a non-negative integer k (the parameter), the Minimum Quartet Inconsistency (MQI) problem is the question whether we can find an evolutionary tree inducing a set of quartet topologies that differs from the given set in only k quartet topologies. The more general problem where we are not necessarily given a topology for every subset of four taxa appears to be fixed-parameter intractable. For MQI, however, which is also NP-complete, we can compute the required tree in $O(4^k \cdot n + n^4)$ time. This means that the problem is fixed-parameter tractable and that in the case of a small number k of "errors" the tree reconstruction can be done efficiently. Moreover, we also discuss its combination with heuristic strategies. Notably, combining heuristics and (exact) fixed-parameter algorithms is a currently prominent line of research in parameterized complexity. We exhibit the use of the algorithm for reconstructing the evolutionary relationship of several sets of fungi species.

Chapter 7: Consensus of Gene Orderings. With breakpoint distance, the genome rearrangement field delivered one of the currently most popular measures in phylogenetic studies for related species. In this chapter, we focus onto the problem of, given gene orderings for a set of species, computing a "consensus" of the gene orderings with respect to breakpoint distance; gene orderings refer to the sequence of genes on a genome. Here, Breakpoint Median, which is NP-complete already for three given species (whose genomes are represented as gene orderings), is the core basic problem. For the important special case of three species, approximation (ratio 7/6) and exact heuristic algorithms are known. Here, we provide an exact, fixed-parameter algorithm with provable performance bounds. For instance, a breakpoint median for three signed orderings over n elements that causes at most d breakpoints can be computed in $O(2.15^d \cdot n)$ time. We show the algorithm's practical usefulness through experimental studies. In particular, we demonstrate that a simple implementation of our algorithm combined with a new tree construction heuristic allows for a new approach to breakpoint phylogeny, yielding evolutionary trees that are competitive in comparison with known results developed in a recent series of papers that use clever algorithm engineering methods.

Chapter 8: Consensus of RNA Secondary Structures. Structure comparison of RNA has become a central computational problem bearing many challenging computer science questions. In this thesis, we present an exact algorithm for the NP-complete LONGEST COMMON SUBSEQUENCE problem for sequences with nested arc annotations. Given two sequences of length at most n and nested arc structure, our algorithm determines (if existent) in $O(3.31^{k_1+k_2} \cdot n)$ time an arc-preserving subsequence of both sequences, which can be obtained by deleting (together with corresponding arcs) k_1 letters from the first and k_2 letters from the second sequence. Thus, the problem is fixed-parameter tractable when parameterized by the number of deletions. Notably, our algorithm introduces new analysis techniques for bounded search trees. This algorithm complements known approximation results which give a quadratic-time factor-2 approximation for the general and polynomial-time approximation schemes for restricted versions of the problem. In addition, we show that in the special case when $k_1 = 0$ or $k_2 = 0$ the problem is solvable in quadratic time by dynamic programming, which results in a considerable speed-up for our search tree algorithm.

Chapter 9 summarizes some results of this work by putting them into the context of current research in parameterized complexity. Using examples taken from this work, we discuss connections to integer linear programming, to approximation algorithms, and to heuristics.

Chapter 10 points out starting points for future research based on this thesis. We describe areas in computational biology where the presented techniques might be applicable. Further, we show promising research directions concerning the development of bounded search tree algorithms.

Chapter 2

Biological Background

Computational biology can be seen as an approach to questions in biology which consists of two interacting steps [160]: In the first step, we pose a biological question and construct a mathematical model of the biological reality such that we can formulate the question as a computational problem. In the second step, we construct an algorithm that solves the problem, where the quality of the algorithm is measured with respect to its time and space requirements and with respect to the optimality of the solution. While, in general, this work is mainly devoted to the second of these two steps, this chapter addresses the first step. We intend to give a coarse overview on those fundamentals of molecular biology which will be relevant in this work. On our way, we will also give some first ideas for models of the biological reality and provide forward pointers to the forthcoming chapters where respective questions are addressed. By no means is this chapter a complete coverage, but rather an introduction into a basic biological terminology that will be used later on. For a more complete discussion of molecular biology, we refer the reader to one of the extensive list of books on this topic, e.g., [32, 126, 191].

2.1 DNA, RNA, and Proteins

The biological information needed to construct and maintain living cells is stored in molecules called *deoxyribonucleotides (DNA)*. These molecules are chains of single units, the *nucleotides*, each of them consisting of the sugar 2'-deoxyribose linked to a phosphate group and one of four bases. These bases are called *adenine*, *cytosine*, *guanine*, and *thymine*, and they determine four different types of nucleotides which are abbreviated by A, C, G, and T, respectively. In living cells, DNA molecules consist of two chains of nucleotides which are wound one around the other, in this way forming a helix structure; this double-stranded structure is based on hydrogen bonds that nucleotides from two different chains can build between each other following certain rules: bonds can exist between nucleotides of type A and T and between nucleotides of type C and G. Therefore, the two chains of a double-stranded DNA molecule have a *complementary* sequence of nucleotides, when we consider A as the complement of T (and vice versa) and C as the complement of G (and vice versa). We can uniquely identify a direction of the DNA molecule, from the so-called $5'$ end to the $3'$ end, which are specified by chemical properties. A DNA molecule can be described by its sequence of nucleotides, i.e., it is modeled as a string over alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. By the *genome* of an organism, we refer to the entire DNA content of one of its cells, consisting of one or several DNA molecules.

While DNA can be seen as a storage of biological information, this information is translated into other types of molecules which can be called the building blocks of life in a cell. Firstly, *ribonucleotides (RNA)* are very similar to DNA with differences in the chemical structure of their nucleotides, more precisely, they contain the sugar ribose rather than 2'-deoxyribose and *thymine* is replaced by the related base *uracil (*U*)*. Secondly, *proteins* are also chains of units, here called *amino acids*. We distinguish 20 types of amino acids. Proteins exhibit a large chemical and structural diversity and perform a vast range of different functions in a cell.

We now outline how the connections between DNA, RNA, and proteins can be described in a simplified manner. Discrete subunits of the DNA molecule, the *genes*, are read in a biochemical process, the *gene expression*, and translated into RNA and proteins. This process can be divided into two stages, *transcription* and *translation*. Transcription produces a one-to-one RNA copy of the gene. We distinguish between genes for whose RNA copy is translated into a protein and genes which do not encode for proteins. In the former case, the RNA copy is called *messenger RNA (mRNA)* and translation produces a protein from an mRNA transcript according to the rules of the *genetic code*: Three nucleotides from the RNA chain encode one amino acid following a fixed correspondence between nucleotide triples and amino acids, which is, however, not one-to-one since, in many cases, more than one of the 64 possible triples correspond to one of the twenty amino acids.

In a broad distinction, DNA and RNA can be seen as the “construction plan” of a cell and its “blueprint” and we refer to them as *genomic data*; the proteins can be seen as the products of this plan. The focus of this work are *genomic* data; here, proteins are only of marginal interest and, therefore, they are not covered in more detail.

Notably, not all parts of a DNA molecule are translated to RNA, but only discrete subunits, the genes, such that one gene encodes for one RNA molecule and, if the gene is protein-coding, for one protein. Genes constitute the *coding* part of a DNA sequence. The DNA sequence preceding the start of a gene is referred to as *upstream region*, the DNA sequence succeeding the end of a gene as *downstream region*. In most cases, genes are preceded and succeeded by a noncoding sequence, and, especially in higher organisms, the sequence of one

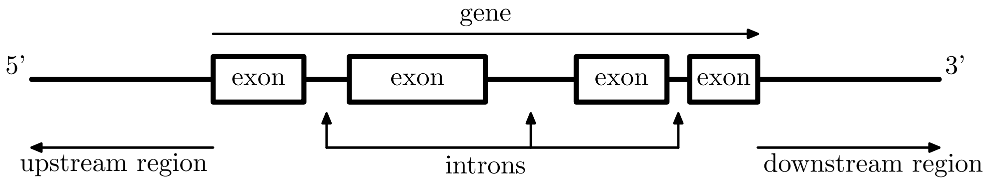


Figure 2.1: Organization of a gene with interspersed introns (noncoding) such that the coding part of the gene is divided into several exons.

gene is not necessarily continuous, but can be intermitted by noncoding parts of sequence. The coding parts of a gene are called *exons* and the noncoding parts between the exons are called *introns*. An outline of the organization of a gene with its introns and exons is given in Fig. 2.1. The introns are removed on the RNA level by a process called *splicing* to reproduce the RNA molecule which, then, is translated into a protein. Noncoding sequence does not encode for RNA or proteins, but may nevertheless contain necessary information, e.g., for controlling and regulating gene expression as we outline in the following.

2.2 DNA Signals and Regulation of Gene Expression

An important role in the translation of a gene into mRNA is attributed to special proteins which interact with the DNA molecule by attaching to the DNA at specific *binding sites*, which are subunits of the DNA sequence with appropriate chemical structure. Binding sites are located in the noncoding region preceding or succeeding the gene. It is generally accepted that the property of a binding site that allows a particular molecule to attach is reflected in its nucleotide sequence. However, a binding site does not correspond to a fixed nucleotide sequence. We observe that, usually, the same protein is involved in the regulation of several genes, but the occurrences of the binding site exhibit variations in its nucleotide sequence. We make the same observation when comparing corresponding binding sites near corresponding genes in different species. Therefore, it is common practice to characterize a binding site by a consensus sequence of its occurrences. This shows a way how we can search for unknown binding sites: We collect upstream or downstream regions of genes that are believed to have related mechanisms of gene expression. In these strings, we search for substrings that have a common consensus. Analogously as shown here, we can search not only for binding sites but, more generally, for common signal sequences, called *motifs*, in DNA sequences. This problem is addressed in Chapter 5.2.

Note some significant differences in the organization of DNA between eukaryotes and prokaryotes (eukaryotes are organisms in which cells contain membrane-bound compartments including, in particular, a nucleus which is the case, e.g., in animals, plants, and fungi; prokaryotes have no such compartments, e.g., bacteria): In eukaryotes, the genome is divided into one or more linear molecules, the *chromosomes*. The genes of eukaryotes are in many cases discontinuous, i.e., interspersed by introns. In prokaryotes, most of the DNA is contained in one circular DNA molecule, genes have no introns, and the genes are more compact, i.e., the portion of noncoding sequence is smaller in comparison to eukaryotes.

2.3 RNA Structure

We have already mentioned that RNA has a similar composition as DNA molecules since both consist of a nucleotide chain, where the single nucleotides consist of a phosphate group and a sugar. In RNA, however, the sugar is ribose compared to 2'-deoxyribose in DNA. Here, we outline further properties that distinguish RNA molecules from DNA. In comparison with DNA, RNA molecules are of smaller length, i.e., at most a few 1000 nucleotides long. RNA molecules occur only single-stranded since they do not form a helix structure as DNA molecules do. Within one RNA molecule, however, bonds between nucleotides are possible following similar rules as we have seen for DNA nucleotides: Bonds can arise between G and C and between A and U, but bonds are also possible between G and U. However, one nucleotide, usually, is involved in at most one bond with another nucleotide. We refer to the nucleotides also as *bases* and when there is a bond between two bases, we call the two involved bases a *base pair*. Bonds turn out to be especially stable if several bonds follow each other, e.g., forming the typical structural feature of a *hairpin loop* as shown in Fig. 2.2(a). An example of *nested* hairpin loops is shown in Fig. 2.2(b). By the term *primary structure* we refer to the sequence of nucleotides. We call the structure of the RNA based on its base pairs *secondary structure* if, intuitively speaking, the structure of an RNA molecule consists only of nested hairpin loops; more formally, in a *secondary structure* or, equivalently, *nested* arc structure, base pairs between positions i_l and i_r and between positions j_l and j_r with $i_l < j_l$ imply that either $i_l < i_r < j_l < j_r$ or $i_l < j_l < j_r < i_r$. Many RNA structures observed in practice satisfy this property. A structure that does not satisfy this requirement is called *tertiary structure* (see Fig. 2.2(c) for an example, see [197] for a more precise definition of secondary and tertiary RNA structure). The structure of RNA can be of central importance for the function of the RNA molecule; in the following, we provide several examples. Two types of RNA molecules which are not translated into proteins and whose function is highly dependent on their structure are *transfer RNA (tRNA)* and *ribosomal RNA (rRNA)*. Transfer RNA is involved in the translation of mRNA to proteins. Ribosomal RNA forms part of *ribosomes*, complexes of RNA and proteins involved in protein synthesis. Structure can also be of importance for mRNA, e.g., for the removal of introns in the translation of the gene into RNA: The introns are removed on the RNA level in the splicing process, which, in many cases, is caused by the RNA structure of the intron. Therefore, there is a close relationship between structure and function and, in many cases, structure is the more appropriate way to characterize RNA sequences. Chapter 8

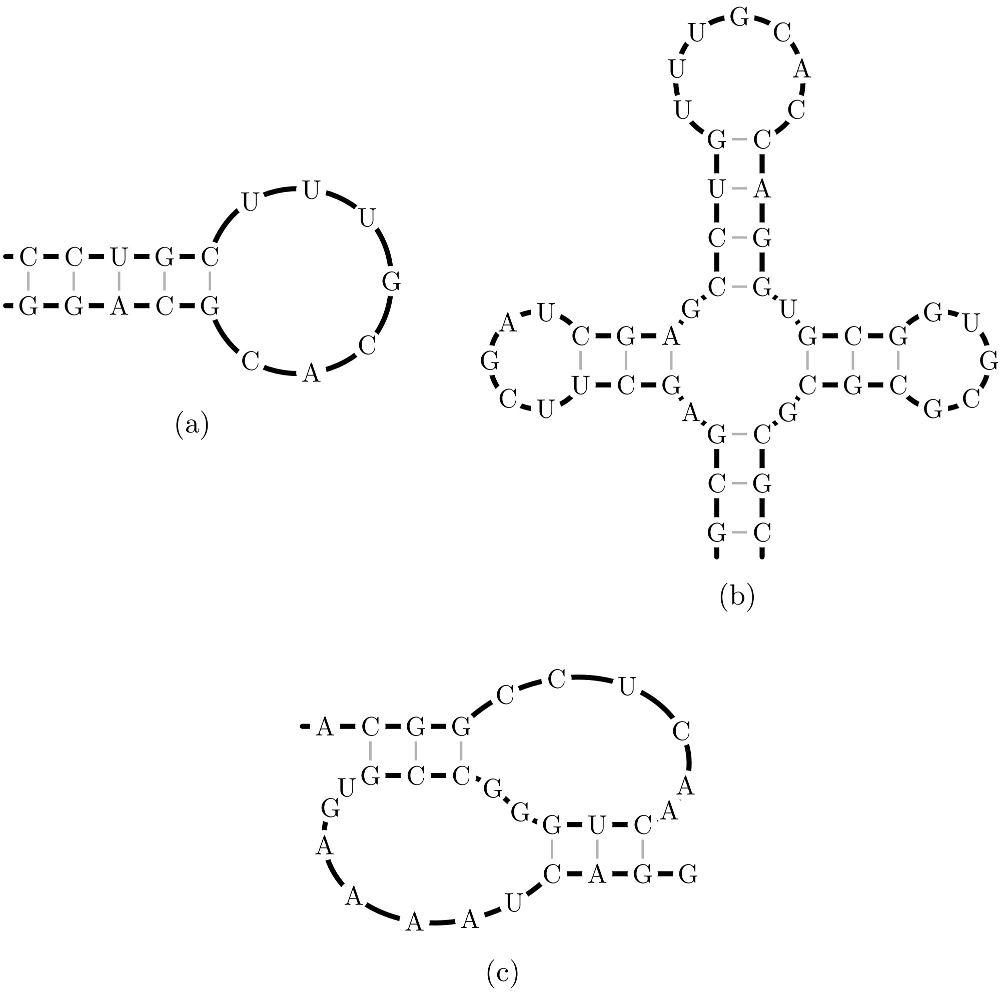


Figure 2.2: Different types of RNA structures. (a) and (b) Examples of RNA secondary structure, (a) an hairpin loop, and (b) an example of nested hairpin loops. (c) An example of a so-called "pseudo-knot" which does not satisfy the requirements of RNA secondary structure.

discusses the comparison of RNA sequences based on *both* their sequence and their structure.

2.4 Polymerase Chain Reaction and Primers

Whenever a cell divides, its DNA duplicates in a process called *DNA replication.* Replication begins with a double-stranded DNA sequence that is separated into two single strands; these single strands are used as a template and complemented, by the addition of single nucleotides, to form double-strands again. Here, we want to outline how a technique called *polymerase chain reaction*

(PCR) uses this process to synthetically amplify small continuous substrings of a given target DNA sequence in order to produce a large number of copies of this substring. These copies are necessary, e.g., in *sequencing*, i.e., when determining the sequence of nucleotides of a given DNA molecule: current sequencing techniques can only process a small piece of a DNA molecule which is available in a large number of copies. The first step of PCR is to design a pair of two oligonucleotides, i.e., small DNA molecules composed of a few nucleotide bases. These oligonucleotides are chosen such that they are complementary to a substring of the DNA sequence at either end of the region to be amplified. They are produced synthetically and called *primers* as they are used to initiate the the transcription on the DNA molecule with an polymerase enzyme, in analogy to a natural chemical reaction occurring in cells. We refer to the complement of a primer in the DNA sequence as the location of the primer. The process of PCR as outlined in the following is also illustrated in Fig. 2.3. In the first phase of PCR, the primers, the double-stranded target DNA (as shown in Fig. 2.3(a)), single nucleotides, and a polymerase enzyme are heated such that the strands of the target DNA detach. When cooling down this mixture again, primers are likely to attach to some of the DNA strands at their location, as shown in Fig. 2.3(b); in these cases, a DNA replication process is started at the location of the primer, yielding a shortened DNA copy. Repeating these steps, heating and cooling down, for several times yields large amounts of small DNA strands that extend from the location of one primer up to the position of the second primer, as shown in Fig. 2.3(c). In Section 5.1, we address the problem of designing PCR primer candidates which are suitable for a set of sequences.

2.5 Genome Evolution and Evolutionary Relationships

Genomes are subject to changes caused by irregularities in the duplication of DNA or by chemical "damage" of DNA. Although cells have several kinds of repair and checking mechanisms that try to prevent such changes, it can happen that changes persist. The effect of such a change can be lethal to the cell causing its death. In other cases, such a change may have no effect to the life of the cell. Inbetween these two extremes, changes have the potential to positively influence the fitness of a cell, i.e., its ability to survive. If such changes are occurring in a one-cell organism or if they are occurring in a germ cell of a multi-cell organism, then these changes can influence the fitness of the whole organism, and, if they are transferred to the next generation, influence the structure of a population. In this way, the genome *evolves* and changes accumulate such that, at some point, individuals with a such evolved genome are considered to belong to a new species. Thus, genome evolution is the reason for the diversity of different species and can be used to gain insight into the evolutionary relationship between organisms and species (note that it is not clearly defined when two organisms with differences in their genomes are

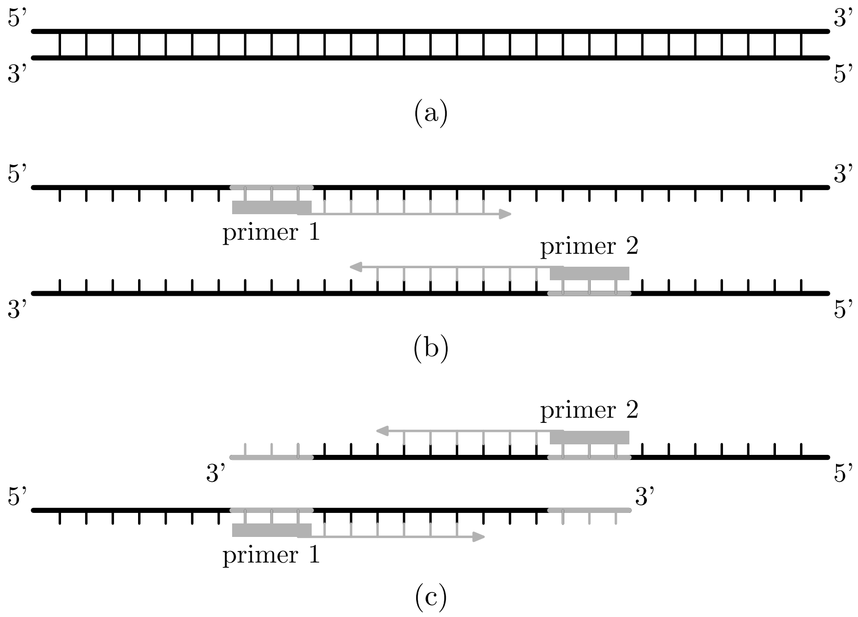


Figure 2.3: The process of PCR to amplify a substring of the target DNA. (a) The double-stranded target DNA. (b) After heating, the strands detach and, when cooling again, primers can attach at their locations at one of the single strands. A DNA replication process is started at the primer. (c) Repeating this process yields pieces of DNA that extend from the location of one primer up to the location of the second primer.

still considered to belong to the same species or when they are considered to already form two different species). We will first outline some forms of genomic changes and, then, outline how they can be used to reconstruct evolutionary relationships.

We distinguish two main types of changes DNA can be subject to, *point mutations* and *gene mutations*. Point mutations are changes that affect single or a small number of nucleotides, by substitution of a nucleotide, by deletion or by insertion of one or a few consecutive nucleotides. Point mutations in coding regions are likely to cause differences in the function of genes, point mutations in non-coding regions are, in comparison, more likely to have no effect to the function of the cell. Since only those changes can persist which do not cause the death or a malfunction of the cell, point mutations are observed with much higher frequency in non-coding regions of DNA. In contrast to point mutations, gene mutations restructure the DNA molecule. Gene mutation events that are considered to be relatively frequent are *transpositions* and *reversals*. Transpositions move a segment of the molecule to another location within the molecule. Reversals result in a segment of the molecule being turned round in its order. Such events are particularly evident if the affected segment contains a sequence of genes such that the order of genes on the molecule is changed by the gene mutation event. Gene mutation events that affect several chromosomes include, e.g., cross-over events which result in the exchange of end segments of two chromosomes.

Duplications constitute a further gene mutation event which replaces a segment of a molecule by two adjacent copies of the segment. This gives rise to the following relations between genes. Two genes can have a common ancestor, i.e., they originated from the same ancestral gene by a series of evolutionary events, and are, then, called *homologous*. Genes originating from a common ancestor can be homologous on the basis of two types of evolutionary events. Firstly, genes may originate from a common ancestor that was duplicated such that each of the two genes evolved *from a different duplication copy* by subsequent point and gene mutation events. In this case, the two genes are called *paralogous*. Secondly, genes may have evolved, by mutations, *from the same copy* of of a common ancestor, finally leading to *speciation*, i.e., the case that the two genomes are considered to belong to different species. If the two genes evolved from the same gene copy of their common ancestor, they are called *orthologous*. Homologous and, in particular, orthologous genes can be a good basis for the inference of hypotheses of the evolutionary relationship of a set of species.

Phylogeny or *phylogenetics* is the classification of species and organisms according to their evolutionary relationships. In *molecular* phylogenetics, this classification is based on genomic data. The single units being compared, usually species, are referred to as *taxa*. Given a set of taxa, a commonly used model for their evolutionary relationship is a tree called *phylogenetic tree* in which the leaves are in one-to-one correspondence to the taxa and in which inner nodes correspond to (unknown) ancestors of these taxa. Often this tree is required to be binary, since we assume that speciation events result in splitting up one into two species. Often this tree is unrooted since choosing the root of an evolutionary tree is difficult based on molecular data only. Unrooted trees are, usually, rooted by including an *outgroup taxon* into the analysis, i.e., a taxon which is known to be more distantly related to the set of analyzed taxa than the taxa among each other; the root of the tree is then placed on the edge that leads to the outgroup taxon. Here, we only address some aspects of phylogeny which will be a useful background for the following chapters. Extensive coverage can be found in monographs such as [158, 178].

A common first step towards using DNA sequences, e.g., of homologous genes, for a phylogenetic analysis is to *align* the sequences. This means that we arrange the sequences in such a way in a character matrix such that we have one row for every gene and such that nucleotides in different genes that obtained their state from one common ancestor nucleotide are grouped in one column. Gap symbols "–" are inserted where one gene does not have a corresponding nucleotide due to deletion or insertion events. In most commonly used models, it is NP-hard to compute an "optimal" alignment for a given set of sequences, e.g., see [30]. However, there is an abundance of algorithms and programs to compute alignments in a heuristic way (for an overview, refer, e.g., to [15, 94, 111]).

Among the methods to reconstruct evolutionary trees, we distinguish *distance-based* from *character-based* methods. Distance-based methods compute a distance value for each pair of taxa and compute a phylogenetic tree from the thereby obtained distance matrix. Distance values may be obtained, e.g., by computing a minimum number of substitution, deletion, and insertion events that are necessary to obtain one sequence from the other. An example of a distance-based method is the popular *neighbor joining method* [158, 173].

In contrast to distance-based methods, character-based methods construct a phylogenetic tree right from the sequence data, in most cases on the basis of the aligned sequences. As an example, we outline the *maximum parsimony* criterion. Maximum parsimony means in phylogeny that we decide between different tree topologies by identifying the one that has the smallest "evolutionary cost," where the cost may depend on the input data and the chosen model. We explain the maximum parsimony criterion for the example that the input is an alignment of nucleotide sequences and we use a model allowing as mutation events the substitution, deletion, and insertion of single nucleotides. Given a tree T in which every node is assigned a sequence of equal length, then, let the *cost of a branch* denote the Hamming distance of the two sequences at its ends. Let the *cost* of T, then, be the sum of scores of all branches in T. Using these conventions, we can explain what is a parsimony tree in this setting: It denotes a phylogenetic tree T in which every leaf is assigned a sequence of the alignment, and for which we can assign sequences to the inner nodes of T such that the cost of T is minimal among all such trees. For the described model having an alignment as input and using the Hamming distance as a cost function, finding the best tree under the maximum parsimony criterion is NP-hard but, e.g., heuristics and approximation algorithms have been given (see [94] for further details).

Naturally, these methods are not limited to sequence data that are evaluated under the assumption of point mutation events. Only as one example, we mention that distance values and the parsimony criterion can also be based on gene order data that are evaluated under the assumption of certain gene mutation events. Searching for optimal trees under the maximum parsimony criterion with respect to gene mutation events is addressed in Chapter 7.

There are also phylogenetic methods that can be used both in a distance-based way as well as in a character-based way. One such method, the quartet method, is discussed in Chapter 6.

Chapter 3

Parameterized Complexity

NP-completeness theory distinguishes computationally “tractable” problems, i.e., those solvable in deterministic polynomial time, from those problems that are classified as “intractable” by showing that they are NP-hard. As a drawback of this classification, NP-hard problems arise in many contexts and, often, real-world instances of these problems, despite their intractability, can be efficiently solved in practice. The observation that “not all forms of intractability are created equal” [58] was a main motivation for introducing the concept of parameterized complexity which is, among others, an alternative and new way to distinguish “tractable” from “intractable” problems. Downey and Fellows developed a theoretical basis for parameterized complexity, manifested in their monograph [58]. Most importantly for our work here, parameterized complexity also provides tools, among them bounded search trees, which can guide the algorithm designer in the development of efficient algorithms for, in particular, NP-hard problems that are tractable in the parameterized sense. Notably, the approach of parameterized complexity results in algorithms with guaranteed optimal results and worst-case bounds on their running time. This distinguishes parameterized complexity from other approaches to solve NP-hard problems like approximations (which do not produce guaranteed optimal results) or heuristics (which do not give guarantees either on the optimality of the results or on the running time). This chapter will give a brief survey of the fundamentals of parameterized complexity while focusing on algorithm design. Section 3.1 introduces our notion of parameterized tractability, so-called fixed-parameter tractability, while also pointing out its limitations. Section 3.2 introduces the concept of parameterized intractability, also discussing how such negative results can guide the search for algorithms. Section 3.3 exhibits techniques for designing fixed-parameter algorithms by giving examples from computational biology. Note that bounded search trees, the central design technique to obtain fixed-parameter algorithms in this work, are discussed in the following Chapter 4.

Two main example problems that will be used in this section are Vertex

Cover and Clique, which are introduced in the following. Both problems are NP-complete but turn out to be not equal concerning their tractability from the view of parameterized complexity.

> Vertex Cover
> **Given:** Graph $G = (V, E)$, and a non-negative integer k.
> **Question:** Does G have a set of vertices $V' \subseteq V$ with $|V'| \leq k$ such that, for every $u, v \in V$ with $\{u, v\} \in E$, either $u \in V'$, $v \in V'$, or both? If the answer is yes, then V' is called a *vertex cover* of size at most k.

Note that Vertex Cover also has applications in computational biology, e.g., for conflict resolution: Assume a set of experiments which are, with small probability, subject to errors. Due to these errors, we observe pairs of experiments whose results contradict each other. Assuming that the number of erroneous experiments is small, it can, in this situation, be desirable to ask for the minimum number of experiments such that, after deleting their results, no two results contradict each other. This is, in essence, Vertex Cover. For details on the application of Vertex Cover for the conflict resolution when computing multiple sequence alignments refer to [186].

> Clique
> **Given:** Graph $G = (V, E)$, and a non-negative integer k.
> **Question:** Does G have a set of vertices $V' \subseteq V$ with $|V'| \geq k$ such that for all $u, v \in V'$ we have $\{u, v\} \in E$? If the answer is yes, then V' is called a *clique* of size at least k.

Vertex Cover and Clique are chosen as examples because they are easy to understand and can be defined without much background information. Also, they became the two flagship problems of parameterized complexity.

3.1 Fixed-Parameter Tractability

The main idea of parameterized complexity is to consider the input as consisting of two parts, and to include the size of both parts in the complexity analysis. Therefore, we use an appropriate framework to define problems.

Definition 3.1.1. *An instance of a* **parameterized problem** *is given as* $(x, k) \in \Sigma^* \times \mathbf{N}$*, where the second component is called the* parameter.

In general, the second component, representing the parameter, can also be drawn from Σ^*; however, for most cases and, in particular, in this work, assuming the parameter to be a non-negative integer is sufficient. E.g., in Vertex

Cover and Clique the first component is the input graph coded over some alphabet Σ and the second component, the parameter, is the size of the relevant vertex set. Moreover, the non-negative integer k denoting the second component may, in general, be replaced by a *vector* of non-negative integers $(k_1, \ldots, k_r)$ for some integer $r > 0$; we call this an *aggregate* parameter. In this work, we will encounter examples of aggregate parameters. For the remainder of this chapter, however, we will, for simplicity, restrict ourselves to non-aggregate parameters, i.e., an aggregate parameter with $r = 1$. It is straightforward to generalize the following concepts to aggregate parameters, simply by replacing, in the definitions, the parameter k by an aggregate parameter $(k_1, \ldots, k_r)$.

Definition 3.1.2. *A parameterized problem is called* **fixed-parameter tractable** *iff there is an algorithm that computes, given an instance* (x, k) *of the problem, a solution in* $f(k) \cdot |x|^{O(1)}$ *running time, where* f *may be an arbitrarily fast growing function depending only on* k*. The complexity class* FPT *consists of all fixed-parameter tractable problems.*

Vertex Cover is *fixed-parameter tractable*: There are algorithms solving it in time less than $O(kn + 1.29^k)$ [43, 154]. By way of contrast, consider the also NP-complete Clique problem. Clique appears to be *fixed-parameter intractable*: It is *not* known whether it can be solved in $f(k)n^{O(1)}$ time for any function f depending only on k [58]. Thus, in some sense, the handling of the parameter seems to be more sophisticated for Clique compared to Vertex Cover. Moreover, unless P = NP, the well-founded conjecture is that no such algorithm exists. The best known algorithm solving Clique runs in $O(n^{ck/3})$ time [146], where c is the exponent on the time bound for multiplying two integer $n \times n$ matrices (the currently best known value for c is 2.38, see [49]). Note that an $O(n^{k+2})$ time algorithm for Clique is trivial, by trying all $O(n^k)$ many size-k subsets of vertices and testing, for each of them, in $O(n^2)$ time whether it is a clique. The decisive point is that k appears in the exponent of n, and there seems to be no way "to shift the combinatorial explosion only into k," independent from n [58, 59, 60].

Fixed-parameter algorithms allow to solve instances of the problem in which the parameter value is sufficiently small, which is the case for many applications. In particular, problems of computational biology often exhibit a variety of possible parameters which are bounded in practice, e.g., the alphabet size in sequence analysis, the maximum number of data errors, or the number of objects under consideration.

Note, however, that the definition of FPT allows the function $f(k)$ to take unreasonably large values, e.g.,

$$f(k) := 2^{2^{2^{2^{2^k}}}} .$$

Thus, showing that a problem is a member of the class FPT does not necessarily imply an efficient algorithm (not even for small k). In fact, many problems that are classified fixed-parameter tractable still wait for efficient and practical algorithms. In this sense, we strongly have to distinguish two different aspects of fixed-parameter tractability: The theoretical part which consists in classifying problems in the parameterized sense, i.e., proving membership in FPT or proving parameterized hardness (see Section 3.2), and the algorithmic component of actually finding efficient algorithms for problems inside the class FPT.

Useful tools for proving membership in FPT, which did, to our best knowledge, not lead to implemented algorithms used in practice, include the Graph Minor Theorem by Robertson and Seymour [58, chapter 7] or the *color coding* technique introduced by Alon *et al.* [5]. In addition to these techniques, we introduce, in Section 5.1.3, a new tool for FPT classification, integer linear programs (ILP's) with a fixed number of variables. This classification can be useful for practical purposes: Showing, in this way, that a problem is fixed-parameter tractable indicates that it is worth the effort to spend more time into the search for more practical fixed-parameter algorithms.

Techniques for designing practical fixed-parameter algorithms—in particular those for which we can find examples in computational biology—will be discussed in Section 3.3.

3.2 Parameterized Intractability

Showing that a problem is unlikely to be fixed-parameter tractable requires an appropriate framework analogous to the theory of NP-completeness [74]. Downey and Fellows [58] developed such a completeness program for showing parameterized intractability. In contrast to the theory of NP-completeness, the completeness theory of parameterized intractability involves significantly more technical effort (as will also become clear when following the hardness proofs presented in Chapter 5.2). We very briefly sketch some integral parts of this theory in the following, starting with an appropriate concept for reductions.

Definition 3.2.1. *Let* $L, L' \subseteq \Sigma^* \times \mathbf{N}$ *be two parameterized problems. We say that* L *reduces to* L' *by a standard* **parameterized m-reduction** *(or, for sake of abbreviation, by a parameterized reduction) if there are functions* $k \mapsto k'$ *and* $k \mapsto k''$ *from* $\mathbf{N}$ *to* $\mathbf{N}$ *and a function* $(x, k) \mapsto x'$ *from* $\Sigma^* \times \mathbf{N}$ *to* Σ^* *such that*

1. $(x, k) \mapsto x'$ *is computable in* $k''|x|^c$ *time for some constant* c *and*
2. $(x, k) \in L$ *iff* $(x', k') \in L'$.

Example. Reducing Clique **to** Independent Set**.** Given a graph $G = (V, E)$, a set $V' \subseteq V$ is an *independent set* iff for all $u, v \in V'$, $\{u, v\} \notin E$. The Independent Set problem is, given a graph $G = (V, E)$ and a non-negative integer k, the question whether G has an independent set of size k.

We outline a reduction from Clique to Independent Set. Given a Clique instance $G = (V, E)$ with parameter k, we generate an Independent Set instance $G' = (V, E')$ with parameter $k' := k$ by setting, for all $u, v \in V$, $\{u, v\} \in E'$ iff $\{u, v\} \notin E$. It can easily be shown that $G = (V, E)$ has a clique of size k iff G' has an independent set of size k'.

The displayed reduction is parameterized since the reduction can be computed in polynomial time and it is parameter-preserving, i.e., the value of k' depends only on the value of k. □

Note that, seemingly, most reductions from classical complexity turn out *not* to be parameterized ones:

Example. Reducing Clique **to** Vertex Cover**.** Given a Clique instance $G = (V, E)$ with parameter k, we generate a Vertex Cover instance $G' = (V, E')$ with parameter k' by setting, for all $u, v \in V$, $\{u, v\} \in E'$ iff $\{u, v\} \notin E$, and by setting $k' = |V| - k$. It can easily be shown that $G = (V, E)$ has a clique of size k iff G' has a vertex cover of size k'.

This reduction is *not* parameterized since it is not parameter-preserving: The value of k' depends not only on the value of k, but also on the value of $|V|$, i.e., on the total size of the input. □

The basic reference point for parameterized intractability can be defined as the class of parameterized languages that are equivalent to the Short Turing Machine Acceptance problem (also known as the k-Step Halting problem). Here, we want to determine, for an input consisting of a nondeterministic Turing machine M (with unbounded nondeterminism and alphabet size), and a string x, whether M has a computation path accepting x in at most k steps. This can trivially be solved in $O(n^{k+1})$ time by exploring all k-step computation paths exhaustively, and it is not expected that this can be much improved. Therefore, this problem is the parameterized analogue of the Turing Machine Acceptance problem that is the basic generic NP-complete problem in classical complexity theory.

Definition 3.2.2. *A parameterized problem is in the complexity class* W[1] *if it is reducible to* Short Turing Machine Acceptance *by a parameterized* m*-reduction. A parameterized problem is* **W[1]-hard** *iff* Short Turing Machine Acceptance *is reducible to the given problem by a parameterized* m*-reduction. A parameterized problem is* **W[1]-complete** *iff it is* W[1]*-hard and it is in* W[1]*.*

Hardness for W[1] is our criterion that a problem is unlikely to be in FPT. It is shown by giving a parameterized m-reduction from any W[1]-complete problem. These problems (there are many) include, besides Short Turing Machine Acceptance, e.g., Clique and Independent Set [57]; the monograph of Downey and Fellows [58] provides a compendium of problems listed by their parameterized complexity, including many W[1]-complete ones, and

Cesati [40] gives an update of this compendium. Thus, for an algorithm designer not being able to show fixed-parameter tractability of a problem, it may be helpful to give a reduction from CLIQUE or some other W[1]-hard problem to the given one using a standard parameterized m-reduction; by this way, the algorithm designer can show that a fixed-parameter algorithm is unlikely and other techniques to solve the problem have to be found.

The conjecture that FPT $\neq$ W[1] is very much analogous to the conjecture that P $\neq$ NP. As an important structural result of parameterized complexity, it could be shown that FPT $\neq$ W[1] under the *exponential-time hypothesis* [58] that there is no sub-exponential-time algorithm for 3-SAT, i.e., no algorithm for 3-SAT with $c^{o(n)}$ running time for constant c and n denoting the number of variables in the input formula [103, 104].[1] The currently best known deterministic algorithm for 3-SAT has a running time of $O(1.481^n)$ when n is the number of variables [53] and the existence of an algorithm with $c^{o(n)}$ running time for constant c is open. Since the exponential-time hypothesis implies P $\neq$ NP but not vice versa, the exponential-time hypothesis is more restrictive than assuming P $\neq$ NP but nevertheless it is still considered a very plausible assumption: Impagliazzo, Paturi, and Zane [104] identify a whole class, called SNP, of problems, for each of which no sub-exponential time algorithm is known, but each of which would have a sub-exponential time algorithm if there is one for 3-SAT.

We only mention briefly that there is a hierarchy of parameterized intractability classes, W[1] only being the lowest level. In general, the classes $W[t]$ are defined based on "logical depth" (i.e., the number of alternations between And- and Or-gates of unbounded fan-in) in boolean circuits. We omit any further details in this direction and refer to the monograph of Downey and Fellows [58]. From a practical point of view, W[1]-hardness gives a concrete indication that a parameterized problem with parameter k problem is unlikely to allow for an algorithm with a running time of the form $f(k) \cdot n^{O(1)}$. This identification of problems which are unlikely to have fixed-parameter algorithms is what is relevant in this work. We conclude by giving two examples of parameterized intractability in computational biology:

Example: LONGEST COMMON SUBSEQUENCE (LCS) is, given k strings s_1, $s_2, \ldots, s_k$ over an alphabet Σ, and a non-negative integer l, the question whether there is a string $s \in \Sigma^l$ that is a subsequence of $s_1, s_2, \ldots, s_k$. LCS can be used in the computation of multiple sequence alignments. In this application, we align, for each character of the longest common subsequence, the corresponding characters in each input string against each other. This gives the basis for a further refinement of this partial alignment using other techniques for processing those (possibly small) parts of the input strings which do not belong to the common subsequence.

For the case of unbounded alphabet, it was shown W[t]-hard for all $t \geq 1$ when the parameter is k, W[2]-hard when the parameter is l, and W[1]-complete when

[1] We adopt here this use of the term *sub-exponential* from [103, 104]

both k and l are parameters [28, 29]. Recently, these results were complemented by a hardness result for fixed alphabet size, which is the biologically more relevant case, showing that LCS is W[1]-hard for parameter k even with binary alphabet [166]. For fixed alphabet size, when the parameter is l (or an aggregate parameter which involves l) we can easily give an fixed-parameter algorithm by enumerating all length-l strings in Σ^l and testing, for each of them, whether it is a subsequence of all input strings. □

Example: Order-Preserving Submatrix (OPSM) is a problem introduced by Ben-Dor *et al.* [18] in the context of analyzing gene expression data. Results from gene expression experiments can be represented by a $n \times m$ matrix M with real-valued entries (e.g., rows corresponding to the genes and columns corresponding to tissues). Such a $n \times m$ matrix is called *order-preserving* if there is a permutation of columns such that, after the permutation, the entries within every row are strictly increasing; more formally, the matrix is order-preserving if there is a permutation $r_1, r_2, \ldots, r_m$ of the values $1, 2, \ldots, m$ such that $M(i, r_j) < M(i, r_{j+1})$ for all $1 \leq i \leq n$ and $1 \leq j \leq m-1$. Given an $n \times m$ matrix M with real-valued entries, and two integers k_1 and k_2, then OPSM is the question whether M has an order-preserving $k_1 \times k_2$ submatrix; a $k_1 \times k_2$ submatrix is obtained by deleting all but k_1 rows and all but k_2 columns. Informally, OPSM is the question of a subset of rows exhibiting a common pattern over a subset of the columns. As was pointed out by Chor [47], in a practical setting, the input matrix has thousands of rows (one for each gene), but less than 50 columns (one for each tissue) which implies that k_2, the number of columns in the submatrix, is bounded by 50. Therefore, a natural direction seems to be the question for the fixed-parameter tractability with respect to parameter k_2.

In response to this question, we can, analogously to the NP-completeness proof shown in [18], give a parameterized m-reduction from Clique where k denotes the clique size to OPSM and $k_1 = k_2 = k$. Thus, as one of the rare examples, this NP-completeness proof turns out to be parameter-preserving, showing that OPSM is W[1]-hard both with respect to parameter k_1 and with respect to parameter k_2. This insight shows us that, in particular, a fixed-parameter with respect to parameter k_2 is unlikely (as well as with respect to parameter k_1 and with respect to the aggregate parameter (k_1, k_2)). Therefore, to obtain fixed-parameter algorithms with respect to these parameters, it seems necessary to investigate special cases of the problem or to consider other parameters. □

3.3 Design of Fixed-Parameter Algorithms

In this section, we highlight some common algorithmic techniques which already have been used for the design of fixed-parameter algorithms in the analysis of genomic data, namely kernelization (Section 3.3.1), enumeration (Section 3.3.2), and dynamic programming (Section 3.3.3). For each of these techniques, we point to problems from computational biology to which they were applied.

An additional design technique not presented in this section are bounded search trees. Since they are the central design technique to obtain fixed-parameter algorithms in this work, we discuss them separately in Chapter 4.

3.3.1 Kernelization

Given a computational problem L, a *reduction rule* translates a given instance I of the problem in polynomial time $\text{poly}(|I|)$ to a simplified instance I' of the same problem such that $I \in L$ iff $I' \in L$. Now consider a parameterized problem with parameter k such that an instance (I, k) is reduced to an instance (I', k'). The problem is reducible to a *problem kernel* (or, for sake of abbreviation, it has a problem kernel) iff, after the application of the reduction rule, we are guaranteed that $|I'| = f(k)$ where $f(k)$ is a function depending only on k but not on $|I|$. For example, if $|I'| = O(k)$, then the problem has a *linear* problem kernel. A problem kernel directly implies a fixed-parameter algorithm: We reduce a problem instance to the problem kernel and then apply exhaustive search. Naturally, the reduction rule may itself consist of a set of separate rules which are applied repeatedly, as long as one of them applies; important is only that, in total, the application of the reduction rules can be done in polynomial time. Reduction rules are especially powerful and useful if they are easy and efficient to compute, e.g., in linear time, and if they are often applicable. In this sense, reduction rules are also valuable as heuristic elements of an algorithm, even if they do not yield a problem kernel.

Example: Sorting by Reversals. In the context of genome rearrangements, Sorting by Reversals is a well-known problem. Background is the observation that, due to gene mutation events during evolution (see Section 2.5), the order of genes often differs between the genomes of related species. For some classes of species, reversals are considered to be the most frequent gene mutation event. The question to compute a value of similarity between two given genomes based on their gene order leads to the following problem. Here, an *ordering* of n elements is a permutation of $\{1, 2, \ldots, n\}$ which denote the genes. E.g., an ordering is given by $\pi = \langle\, e_1 \,\ldots\, e_n \,\rangle$ where $\{e_1, \ldots, e_n\} = \{1, 2, \ldots, n\}$ (for details on this model see Chapter 7). Given an ordering π, a *reversal* inverts the order of a subset of successive elements. E.g., given π as shown before, a reversal of elements e_i to e_j transforms π to $\langle\, e_1 \,\ldots\, e_{i-1}\, e_j\, e_{j-1} \,\ldots\, e_i\, e_{j+1} \,\ldots\, e_n \,\rangle$. Given two orderings π_1 and π_2 on n elements and a non-negative integer d, Sorting by Reversals is the question whether we can find a sequence of d reversals that transforms π_1 to π_2.

We distinguish two important cases: The elements are assigned an orientation or not; the problem is, then, referred to as *Signed* or *Unsigned* Sorting by Reversals, respectively. While Signed Sorting by Reversals is solvable in polynomial time [113] (the decision problem even in linear time [8]), Unsigned

Sorting by Reversals is NP-complete. In the unsigned case, we call two elements e_i and e_j *adjacent* in an ordering π iff one of them follows the other, no matter whether e_i follows e_j or e_j follows e_i. Given two unsigned orderings π_1 and π_2, both on n elements, we call a pair of elements (e_i, e_j) a *breakpoint* in π_1 with respect to π_2 iff e_i and e_j are adjacent in π_1 but are not adjacent in π_2. Moreover, a *singleton* is an element $e \in \{1, \ldots, n\}$ such that none of the two elements adjacent to e in π_1 is adjacent to e in π_2. Hannenhalli and Pevzner [98] give a fixed-parameter algorithm for Unsigned Sorting by Reversals: They show that it can be solved in $O(2^k n^3 + n^4)$ time, where k is the number of *singletons*; since one reversal can remove at most two breakpoints in π_1 with respect to π_2, the number of singletons is upperbounded by $2d$. Central for this result was that they could show that there is an optimal sequence of reversals that does not "cut long strips": Here, a *strip* is a set of three or more elements $e'_1, \ldots, e'_r \subseteq \{1, \ldots, n\}$ such that e'_i is adjacent to e'_{i+1} both in π_1 and in π_2, for all $i = 1, \ldots, r-1$. A reversal *cuts* this strip if there is $i \in \{1, \ldots, r-1\}$ such that e'_i is not adjacent to e'_{i+1} after the reversal. Therefore, the above observation also enables us to show a problem kernel for Unsigned Sorting by Reversals: We reduce the input instance by replacing each long strip of arbitrary length by a long strip of length three. Omitting the details, we find the following: If, after this replacement, the number of elements is more than $6d + 3$, then we cannot find a solution with at most d reversals. Therefore, we can assume that a reduced instance has at most $6d + 3$ elements. □

Example: Tree distances. In phylogenetics, it is, in some situations, of interest to compute a distance value for two (unrooted and binary) phylogenetic trees with the same leaf set. This can be useful, e.g., to obtain a notion of the "neighborhood" of a tree. The distance value can be based on the number of operations that are necessary to transform one tree into the other, where the kind of allowed operations are dependent on the model we use. An example is *tree bisection and reconnection (TBR)*, where the allowed operations are to remove an edge of the tree, resulting in two unconnected trees, and reconnect these trees by creating a new edge between the midpoints of two arbitrary edges, one in each tree. Allen and Steel [4] point out that it is NP-complete to compute the TBR distance between two trees. However, given two trees T_1 and T_2 and a non-negative integer d, they show a fixed-parameter algorithm with respect to parameter d based on two reduction rules by which they replace isomorphic subtrees which occur both in T_1 and T_2 by single leaves. Thus, they show that, after applying their kernelization rules until none of them is applicable any more, the trees either have size at most $28d$ or the instance can be rejected. □

3.3.2 Enumeration

In many cases, fixed-parameter algorithms can be easily obtained by enumerating all possible objects which are candidates for solutions and by testing, for each of them, whether it satisfies the problem's requirements. Here, however, we have to be especially aware that enumeration techniques often have severe limitations and that classifying an algorithm as fixed-parameter tractable using enumeration does not necessarily imply that it works efficiently in practice. We mention two examples and point out their limitations.

Example: Phylogenetic Trees. In phylogenetics, when searching a phylogenetic tree for a given set of k taxa, there are algorithms which simply inspect the space of all possible trees over k taxa [142, 174]. On the one hand, the number of these trees is bounded by a function depending only on k. Formally, this shows fixed-parameter tractability with respect to k. On the other hand, this function is given by $\prod_{j=3}^{k}(2j-5)$, which is growing fast with increasing k and, usually, only allows to compute solutions for $k < 15$. □

Example: Motif Search. Enumeration can be applied, where the solution is a DNA string of limited length l. In *motif search*, the CLOSEST SUBSTRING problem is the question whether, given a set of length n strings $s_1, s_2, \ldots, s_k$ over alphabet $\Sigma = \{A, C, G, T\}$ and non-negative integers d and l, whether there is a length-l string over Σ such that all $s_1, s_2, \ldots, s_k$ contain length-l substrings $s'_1, s'_2, \ldots, s'_k$, respectively, with $\max_{i=1,\ldots,k}(s, s_i) \leq d$. This problem can be solved by enumerating all length-l strings over alphabet $\Sigma = \{A, C, G, T\}$ (this can be done in $O(4^l)$ time) and by testing, for each of them, whether there is a matching substring in each of the input strings (this can be done in $O(n + l)$ time for each input string). Such an approach turns out to be useful in practice: Blanchette *et al.* [27] show for the related PHYLOGENETIC FOOTPRINTING problem, how to make the enumeration more efficient by using a dynamic programming approach and, thereby, they give a celebrated algorithm for this problem with $O(k \cdot \min(n \cdot 3l^{d/2}, l \cdot 4^l + n))$ running time, essentially a fixed-parameter algorithm for parameter l. However, even with these improvements, for larger values of l (which, usually, go along with larger values for d), such an solution is not applicable anymore. E.g., Blanchette *et al.* [27] report about finding motifs up to $l = 12$ when $d = 5$ or up to $l = 30$ when $d = 1$. Therefore, we will revisit CLOSEST SUBSTRING in Section 5.2. □

3.3.3 Dynamic Programming

Often, enumeration alone does not lead to fixed-parameter algorithms or the running time is not satisfactory. Here, in some cases, special ways of dynamic programming are used. Dynamic programming is a bottom-up method which starts with computing solutions for small subproblems, storing their solutions in a dynamic programming table. Solutions for the next-larger subproblems are computed using already computed table entries. This is repeated until we have finally computed all table entries such that we can determine the solution to the overall problem. When the size of the table is only polynomial in the input size, dynamic programming is a tool to obtain polynomial-time algorithms. For a parameterized problem with parameter k, a dynamic programming algorithm is a fixed-parameter algorithm with respect to k

- if the number of table entries is $O(f(k) \cdot n^{O(1)})$ for n denoting the size of the input instance, k denoting the parameter, and an arbitrary function f, and
- if a table entry is computed in $O(g(k) \cdot n^{O(1)})$ time for an arbitrary function g.

Below, we present a fixed-parameter algorithm for a problem in the context of SNP haplotyping; the algorithm is based on enumeration but, here, enumeration alone would not be sufficient to show fixed-parameter tractability. We have already mentioned, in Subsection 3.3.2, another example of this kind by the algorithm of Blanchette *et al.* [27] for motif search: The PHYLOGENETIC FOOTPRINTING problem could, in contrast to the closely related CLOSEST SUBSTRING problem, not be shown fixed-parameter tractable by using enumeration alone, but was shown fixed-parameter tractable using additional dynamic programming. In RNA structure comparison, Evans [64, 65] uses dynamic programming to compute the longest arc-preserving subsequence of two arc-annotated sequences, both with crossing arc-structures; for the so-called "cut-width" as parameter she, thus, derives a fixed-parameter algorithm.

Example: SNP Haplotype Assembly problem. A single nucleotide polymorphism (SNP) is a genetic variation between two individuals of the same species, consisting of particular positions in the DNA (the SNP positions) at which one of two possible nucleotides (the SNP states) is observed in different individuals. As humans have two copies of each chromosome, the copies may carry different SNP states at the SNP positions. Then, a *haplotype* denotes the sequence of SNP states of one chromosome copy. SNP's cause problems when sequencing the chromosomes since one can only sequence fragments of the chromosomes and one is faced with fragments coming from both chromosome copies as well as with data errors. The question to partition the set of fragments coming from the process of sequencing the chromosome into two sets, one for each haplotype, leads to a set of problems called SNP Haplotype Assembly problems, one of which we address below.

The input of SNP Haplotype Assembly problems is a $m \times n$ matrix M over alphabet $\{0, 1, -\}$, where rows of this matrix represent fragments coming from the process of sequencing the chromosome, columns of this matrix represent SNP positions, and the matrix represents an alignment of the fragments. The goal is, roughly, to partition the set of columns into two sets, such that every of these sets represents one haplotype. Entry $M[f, s]$ denotes the SNP state of fragment f has at SNP position s. If fragment f contains SNP position s then $M[f, s]$ is given by one of two possible states of SNP s, denoted by 0 and 1. If fragment f does not contain SNP position s then $M[f, s] =$"$-$". When $M[f, s] =$"$-$" for a SNP position s within fragment f, i.e., there is a SNP position s_1 preceding s with $M[f, s_1] \neq$"$-$" and a SNP position s_2 succeeding s with $M[f, s_2] \neq$"$-$", then we say that f has a *gap* and, for one fragment f, the number of such SNP positions which are gaps is called the *total gap length* of f.

Two fragments f and g are *in conflict* if there is a SNP position s such that $M[f,s], M[g,s] \in \{0,1\}$ and $M[f,s] \neq M[g,s]$. For example, consider the matrix

$$\begin{pmatrix} - & 0 & 1 & - & 1 & 1 & - \\ - & - & 1 & 1 & 0 & 1 & - \\ - & - & - & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & - & - & - \\ - & 1 & 0 & - & 0 & 0 & - \end{pmatrix}$$

Here, the fragment denoted by the first row and the fragment in the last row each have a gap of length 1. E.g., the fragments represented by the first two rows are in conflict since they exhibit a different SNP state at SNP position 5.

The objective of Single Individual SNP Haplotyping problems is now, informally, to partition the fragments into two maximally consistent sets, each one corresponding to one SNP haplotype, while facing sequence errors as well as alignment errors. In the following we describe one specific Single Individual SNP Haplotyping problem.

The Minimum Fragment Removal (MFR) problem asks for a minimum number of fragments such that after their removal the data matrix is *error-free*, i.e., we can find a partition of fragments into two sets such that no two fragments within a set are in conflict. Informally, this addresses the goal of Single Individual SNP Haplotyping by trying to find fragments which are wrongly sequenced or wrongly aligned by assuming that their number is small. For example, in the matrix shown above, we can find a bipartition of fragments when deleting the last row: Then, neither rows 1 and 3 nor rows 2 and 4 are in conflict.

We distinguish the "gapless case" in which no fragments has a gap from the "gap case" otherwise. In the gapless case, MFR is solvable in $O(m^2n + m^3)$ time by dynamic programming [170]. The gap case is, however, of practical relevance but is NP-hard [122]. Rizzi et al. [170] present a fixed-parameter algorithm for MFR when parameter k denotes the maximal total length of gaps of one fragment. Their basic idea is to try, for every fragment, all ways to fill the gaps in the fragment with one of two possible states. For one fragment, there are at most 2^k ways to fill the gaps. In this way, they extend their dynamic programming algorithm for the gapless case and show a total running time of $O(2^{2k}m^2n + 2^{3k}m^3)$ to solve MFR in the gap case. Note that a trivial enumeration alone would not give a fixed-parameter algorithm since we would have to enumerate all possible ways to fill the gaps in *all* fragments with one of two possible states (these are $O(2^{mk})$ many possibilities) and invoke, for each of them, the algorithm for the gapless case. □

3.3.4 Bounded Search Trees

Bounded search trees are the central algorithm design technique in this work and are, therefore, discussed separately in the following Chapter 4.

Chapter 4

Search Tree Algorithms

This section provides an introduction into the design and the analysis of fixed-parameter search tree algorithms. We illustrate our explanations using the NP-complete Vertex Cover problem as a running example.

> Vertex Cover
> **Given:** Graph $G = (V, E)$, and a non-negative integer k.
> **Question:** Does G have a set of vertices $V' \subseteq V$ with $|V'| \leq k$ such that, for every $u, v \in V$ with $\{u, v\} \in E$, either $u \in V'$, $v \in V'$, or both? If the answer is yes, then V' is called a *vertex cover* of size at most k.

We decided to use this example since it is an easy-to-understand problem and it allows to explain a number of features that are desirable for good search tree algorithms. In computational biology, Vertex Cover finds application, e.g., for conflict resolution [186]. Moreover, the algorithmic strategies for Vertex Cover described in this chapter were generalized to the 3-Hitting Set problem by Niedermeier and Rossmanith [153]; their algorithm was the point of origin for our results described in Chapter 6.

Note that, up to today, only very few examples of non-trivial search tree algorithms can be found in the computational biology literature. This shortcoming is addressed by this thesis: We explore the design and the applicability of search tree algorithms for the analysis of DNA and RNA data.

4.1 Branching

A recursive procedure constitutes the core part of a search tree algorithm. It mainly consists of two parts: A set of *branching rules* and a *stop criterion*, as explained in the following.

Given a problem instance, a branching rule determines how to create a set of simplified instances such that the original instance has a solution iff one of the created simplified instances has a solution. Instead of *one* branching rule, we can also use a set of *several* branching rules, and decide, depending on the given instance, which of them to use. With several rules, it is only important to provide a case distinction that chooses, given an instance, exactly one of the branching rules which is to apply. The stop criterion specifies when to stop the recursion because the instance does not have a solution. In search trees for parameterized problems, this will usually be the case when the parameter value is 0 and no solution has been found.

Basically, a search tree procedure goes through the following three steps:

1. We test whether we found a solution; if we found a solution, we store or output this solution and backtrack in the recursion, indicating that a solution has been found.
2. We test whether the stop criterion applies; if it applies then we backtrack without finding a solution in this branch of the search tree.
3. We determine which branching rule to apply and create simplified instances according to the chosen branching rule. On each of these instances, we recursively invoke the search tree procedure until either one of them finds a solution or all instances have been processed; in the latter case, no solution is found in this branch of the search tree. (If we are not only interested in *one* solution but in as many as possible then we invoke, in any case, the search tree procedure on *all* instances.)

The resulting algorithm is termed a *search tree* algorithm since the dependency of recursive invocations (or calls) of this procedure can be depicted by a tree. The set of simplified instances that are generated by a branching rule are also called *subcases* and if we invoke the search tree procedure on a simplified instance, we *branch into a subcase.* The *search tree size* denotes the number of times the search tree procedure is invoked.

The algorithm is termed *bounded* search tree algorithm iff the search tree size can be upperbounded by a function depending only on the parameter.

Example: A simple branching rule for **VERTEX COVER.** Assume that we are given an instance of VERTEX COVER, consisting of a graph $G = (V, E)$ and a non-negative integer k, complemented by a set V' which, initially, is empty and by which we intend to successively construct a valid vertex cover. We choose an arbitrary $\{u, v\} \in E$. At least one of u and v has to be in the vertex cover. Therefore, we create two simplified instances, one of them by putting u into the vertex cover, the other by putting v into the vertex cover. The simplified instances are, then, denoted by a graph $G_{new} = (V_{new}, E_{new})$, a non-negative integer k_{new}, and a set V'_{new} containing the vertices which are already in the vertex cover:

Put u into the vertex cover. We set $V'_{new} := V' \cup \{u\}$, we decrease the parameter value k by one, i.e., $k_{new} := k - 1$, and we delete all edges in E that contain u, i.e., $E_{new} := E - \{\{u, z\} \in E \mid z \in V\}$.

Put v into the vertex cover. We set $V'_{new} := V' \cup \{v\}$, $k_{new} := k - 1$, and $E_{new} := E - \{\{v, z\} \in E \mid z \in V\}$.

We invoke the search tree procedure recursively on each of these two instances, replacing E, k, and V' by E_{new}, k_{new}, and V'_{new}, respectively. With this branching rule, we have found a solution if $k \geq 0$ and E is empty; then, V' is a vertex cover for the initial instance. If, however, $k = 0$ and E is non-empty, i.e., there are still uncovered edges, then we backtrack without further recursion: there is no chance of extending V' to a vertex cover of size at most k.

With the described search tree procedure, every application of the branching rule implies two recursive calls of the procedure. Each recursive call decreases the value of k by one. After a sequence of k consecutive calls, the value of k is zero and the recursion stops. Therefore, $O(2^k)$ is an upper bound for the search tree size and, thus, this algorithm is a bounded search tree algorithm. □

4.2 Simplification

Rules that simplify a given problem instance in polynomial time such that the simplified instance has a solution iff the non-simplified instance has a solution are called *reduction rules*. Usually a reduction rule, firstly, tests whether the instance has a specific property and, secondly, performs a simplification based on the property; to yield a correct reduction rule in our context, both testing the property as well as the reduction have to be doable in polynomial time, measured in the size of the instance. If we have a set of reduction rules for a problem, we invoke, given a problem instance, these reduction rules as long as one of them is applicable. We call the instance *reduced* if none of the reduction rules applies. But even if an instance is reduced, the reduction rules may become applicable again after a modification of the instance due to branching. Therefore, the application of reduction rules is interleaved with the application of branching rules: After each recursive call of the search tree procedure, we apply the reduction rules until the instance is reduced. Then, we continue with the search tree procedure.

Applying reduction rules has several advantages: Firstly, they have the potential to drastically reduce the size of the search tree in practice and, thereby, improve the performance of the algorithm. Secondly, they often allow one to make assumptions about the reduced instance. This can be important when specifying a set of branching rules. We only have to provide branching rules for situations that are possible in a reduced instance. Situations that lead to unfavorable branchings, because, e.g., they are complex or a bottleneck for the analysis of the search tree size, can, in this way, be avoided when they cannot occur in a reduced instance.

Example: Reduction rules for Vertex Cover**.**

1. If there is a $u \in V$ such that there is no $v \in V$ with $\{u, v\} \in E$, then we can delete u from V: $V_{new} := V - \{u\}$.

   Correctness of this rule. Since u is endpoint of no arc in E, it would not make sense to put u into the vertex cover.

2. If there is a $u \in V$ such that there is only one $v \in V$ with $\{u, v\} \in E$, then we can take v into the vertex cover, decrease the value of k by one, and delete all edges $\{v, z\} \in E$, $z \in V$, from E since they are covered: $V'_{new} := V' \cup \{v\}$, $E_{new} := E - \{\{v, z\} \mid z \in E\}$, and $k_{new} = k - 1$.

   Correctness of this rule. We have to show that if the instance before the application of the rule has a vertex cover of size k then the instance after the application has a vertex cover of size $k - 1$ (the reverse is obvious). To cover $\{u, v\} \in E$, at least one of u and v has to be in the vertex cover. Assume that u belongs to an optimal vertex cover, but not v. Then, we could as well replace u in the vertex cover by v: $\{u, v\}$ is still covered and there are no other edges $\{u, z\} \in E$, $z \in V$, with $z \neq v$ that would remain uncovered. Therefore, it is at least as good to put v into the vertex cover as to put u into the vertex cover: this choice covers $\{u, v\} \in E$ and, moreover, all possibly existing edges $\{v, z\} \in E$, $z \in V$.

After invoking these reduction rules as often as possible, we can assume that every $u \in V$ has at least two "neighbors," i.e., there are $v, z \in V$, $v \neq z$, with $\{u, v\} \in E$ and $\{u, z\} \in E$. Since every rule application removes at least one vertex or one edge, the number of possible applications is linear in the size of the original instance. Every application can be done in linear time. Therefore, reducing a Vertex Cover instance according to these rules can easily be done in quadratic time (and can, by a more thoughtful organization of the rules, even be done in linear time). □

4.3 Kernelization

A special kind of reduction rules are those leading to a problem kernel, as explained in Section 3.3, i.e., the size of the reduced instance is bounded by a function depending only on the parameter and not by the size of the input instance. In the lucky situation that a reduction to a problem kernel of bounded size is known, it is, obviously, useful to, firstly, compute a problem kernel, and, secondly, invoke the search tree algorithm on the reduced instance. This combination can result in especially useful algorithms. Although we already obtain a fixed-parameter algorithm by exhaustive search on the computed problem kernel, this, often, may be still too time-consuming. Bounded search trees can considerably improve the function measuring the exponential growth of running time and, thus, they can be the key to a good performance in practice.

Example: A problem kernel for VERTEX COVER**.** For VERTEX COVER, we can find a problem kernel of size $O(k^2)$ based on the following rule which is used in addition to the two reduction rules shown above:

Problem Kernel Rule (due to Buss and Goldsmith [37]) If there is a $u \in V$ such that there are more than k edges in E of which u is an endpoint, then u has to be in the vertex cover, i.e., $V'_{new} := V' \cup \{u\}$. Thus, we can delete all edges in E of which u is an endpoint:

$$E_{new} := E - \{(u, v) \in E \mid v \in V\}.$$

The parameter value k is decreased by one: $k_{new} := k - 1$.

Correctness of this rule: Assume that u would not be in the vertex cover. Then, *all* vertices that are connected to u by an edge in E would have to be in the vertex cover, and, thus, the vertex cover would be larger than k. Therefore, the instance has a vertex cover of size k prior to the application of this rule iff the instance after the application has a vertex cover of size $k - 1$.

Analogously to the other reduction rules mentioned above, we invoke the problem kernel rule as long as it is applicable. Notably, a given graph G with parameter k can be reduced with respect to this rule in $O(|G|)$ time. The resulting graph G_{new} with parameter k_{new} contains no vertex of degree larger than k_{new} and G has a size-k vertex cover iff G_{new} has a size-k_{new} vertex cover. However, we can find a vertex cover of size at most k_{new} only if the reduced instance contains at most k^2_{new} edges; if the reduced instance contains more than k^2_{new} edges, then the instance does not have a solution and can be rejected. Since $k_{new} \leq k$, this shows a problem kernel of size $O(k^2)$ which is computed in linear running time.

Besides this straightforward problem kernel, Nemhauser and Trotter [144] even give a problem kernel of size $O(k)$ for VERTEX COVER, which, however, relies on a more complex reduction rule. □

The computation of a problem kernel can be *interleaved* with the application of branching rules in a search tree algorithm: The problem kernel reduction is invoked not only at the beginning of the algorithm, but after every branching step. This makes sense when branching into a subcase can possibly give rise to new application of the problem kernel rule. Niedermeier and Rossmanith [151] presented this *interleaving technique* and show, in particular, that it allows an improved analysis of running time: Let a search tree algorithm have running time $O(n^{O(1)} + k^{O(1)}c^k)$ for an instance of size n with parameter k, where the factor $k^{O(1)}$ denotes the time needed to process the problem kernel in every search tree node. Then, interleaving the branching steps with the problem kernel reduction yields a running time of $O(n^{O(1)} + c^k)$. For the Vertex Cover example this means that the running time analysis for the algorithm outlined above can be improved from $O(|G| + k^2 \cdot 2^k)$ to $O(|G| + 2^k)$ (with interleaving).

4.4 Recognizing Easy Instances

Sometimes, we can avoid to invoke reduction and branching rules until the instance is empty. We may encounter instances that have special easy-to-check properties such that they can be solved in polynomial, or even linear, time. This can be, on the positive side, an instance for which a solution can be computed in polynomial time, or, on the negative side, a solution for which we can immediately determine that it has no solution. Specifying these kind of instances and developing fast strategies to solve them is a task of its own which can considerably speed-up the search tree algorithm in practice, since it can avoid to traverse large parts of the search tree. Unfortunately, it seems difficult to exploit this technique for improving the bounds on the search tree size.

Example: An easy instance of Vertex Cover**.** Assume an instance of Vertex Cover in which all vertices $u \in V$ have exactly two neighbors, i.e., there are exactly two vertices $v, z \in V$, $v \neq z$, such that $\{u, v\} \in E$ and $\{u, z\} \in E$. Although this instance is reduced with respect to the reduction rules shown above, we can solve it without further branching. In this situation, the edges in E form one or several edge-disjoint *cycles* of vertices; a cycle is a set $\{v_1, \ldots, v_r\} \in V$ of vertices such that $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, r$ and $\{v_r, v_1\} \in E$. It is easy to verify that, for a cycle with an even number of vertices, an optimal vertex cover of that cycle contains every second vertex of that cycle. For a cycle with an odd number of vertices, an optimal vertex cover of that cycle contains two adjacent vertices of the cycle and, apart from that, every second vertex of that cycle. With this choice, we can find an optimal solution in linear time. □

4.5 Analysis

The running time of our algorithm is, asymptotically, mainly dominated by the size of the search tree. Thus, we are looking for an upper bound on the search tree size which is determined by the number of children of each node and the height of the tree. To achieve a fixed-parameter algorithm, our goal is to bound the search tree size by a function depending only on the parameter k of the problem.

Branchings are easy to analyze when in each subcase the value of the parameter is decreased by one. E.g., in the example shown in Section 4.1, the branching has two subcases, each time decreasing the parameter by one, and, thus, we concluded a bound on the search tree size of $O(2^k)$. However, search tree algorithms can apply more complicated branchings, decreasing in some of the subcases the parameter value by more than one. In the following we show how to give upper bounds on the search tree size in the case of these more general types of branchings.

Example. A new branching rule for Vertex Cover. With the rules for Vertex Cover presented in Sections 4.2, 4.3, and 4.4, we can make certain assumptions about a Vertex Cover instance which is reduced with respect to these rules. E.g., we conclude that G contains no vertices of degree zero and one (since, otherwise, one of the reduction rules presented in Section 4.2 would apply), we conclude that there is at least one vertex of degree more than 2 (otherwise, the rule presented in Section 4.4 would apply), and we conclude that all vertices have degree at most k (since, otherwise, the problem kernel rule presented in Section 4.3 would apply). Therefore, we can, instead of the branching rule described in Section 4.1, also use the following branching rule:

Branching rule: Choose an $u \in V$ with maximum degree d; u has degree at least three and there are pairwisely different $v_1, v_2, \ldots, v_d$, with $\{u, v_1\} \in E$, $\{u, v_2\} \in E, \ldots, \{u, v_d\} \in E$. Then, we branch into the following two subcases:

Put u into the vertex cover. $V'_{new} := V' \cup \{u\}$, $E_{new} := E - \{\{u, z\} \in E \mid z \in V\}$, and $k_{new} := k - 1$.

Put $v_1, v_2, \ldots, v_d$ into the vertex cover. $V'_{new} := V' \cup \{v_1, v_2, \ldots, v_d\}$, $E_{new} := E - \{\{v, z\} \in E \mid v \in \{v_1, v_2, \ldots, v_d\}, z \in V\}$, and $k_{new} := k - d$.

We invoke the search tree procedure recursively on each of these three instances, replacing E, k, and V' by E_{new}, k_{new}, and V'_{new}, respectively. In the following, we will see how we can obtain upper bounds on the size of the resulting search tree which are better than $O(2^k)$. □

Consider a node of the search tree branching into r subcases. Branching into subcase i, $1 \leq i \leq r$, let k_i be the value by which the we reduce the value of k. Then, we assign to the node the *branching vector* $(k_1, k_2, \ldots, k_r)$ which abbreviates the recurrence relation

$$T_k = T_{k-k_1} + T_{k-k_2} + \ldots + T_{k-k_r}, \tag{4.1}$$

where we set

$$T_0 = T_1 = \ldots = T_{\max(k_1, k_2, \ldots, k_r)-1} = 1.$$

This recurrence relation computes the number T_k of leaves of a search tree handling an instance with parameter k, assuming that all nodes apply this branching rule and disregarding additional reduction rules. Reducing in one branching subcase the parameter value by k_i, we have to solve a resulting instance with parameter value $k - k_i$ and the number of leaves in the search tree handling this instance is given by T_{k-k_i}. The recursion stops when $k < \max(k_1, k_2, \ldots, k_r)$ since we can, for example, maintain a dictionary in which we store the optimal solutions for small instances of constant size such that looking up the solution in the dictionary constitutes a leaf in the search tree without further recursion. The number of leaves of a search tree, however, is, up to a constant factor, an upper bound on the total search tree size since the search tree procedure branches into at least two subcases, and, therefore, inner nodes of the search tree have at least two children. To continue our analysis of the search tree size, we shall briefly consider the solving of recurrences.

Recurrence relation (4.1) is called *linear* since all T_{k-k_i} occur to the first power and it is called *homogeneous* since there is no additive constant term. The corresponding polynomial in z

$$p(z) := z^k - z^{k-k_1} - z^{k-k_2} - \ldots - z^{k-k_r} \tag{4.2}$$

is referred to as the *characteristic polynomial* of the recurrence. It is a polynomial of power k and it has k roots, which are called *characteristic roots*. In general, among these roots there can be repeated roots and some roots may be complex numbers.

We search for an easy way how to compute an upper bound for T_k (as defined in recurrence (4.1)) without recursion. In the remainder of this subsection, we outline some results helping us to achieve this goal; proofs are omitted with references where they can be found. We can find a characterization of T_k in recurrence (4.1) based on the characteristic roots, e.g., as shown in Chapter 5.1 of [171].

Definition 4.5.1. *We denote the distinct characteristic roots of polynomial (4.2) by α_1, $\alpha_2, \ldots, \alpha_q$, and we use m_i to denote the multiplicity of root α_i, $i = 1, \ldots, q$. Then, the set of* basic solutions *of recurrence (4.1) is the union of*

$$\alpha_i^k, k\alpha_i^k, k^2\alpha_i^k, \ldots, k^{m_i-1}\alpha_i^k$$

for all $i = 1, \ldots, q$.

Using this terminology, we can express T_k in terms of the basic solutions of recurrence (4.1):

Theorem 4.5.2. *([171], Theorem 5.2) Suppose that a linear homogeneous recurrence (4.1) with constant coefficients has basic solutions b_1, $b_2, \ldots, b_p$. Then*

$$T_k = \lambda_1 b_1 + \lambda_2 b_2 + \ldots + \lambda_p b_p \tag{4.3}$$

for some constants λ_1, $\lambda_2, \ldots, \lambda_p$.

The right-hand term in equation (4.3) is mainly determined, asymptotically and up to a polynomial factor in k, by a *maximum characteristic root* of polynomial (4.2), i.e., the characteristic root having a maximum absolute value. Therefore, by Theorem 4.5.2, a maximum characteristic root gives us an asymptotic approximation for the solution of recurrence (4.1): Let α denote the maximum characteristic root of polynomial (4.2) and let m be the multiplicity of this root, then equation (4.3) yields

$$T_k = O(k^{m-1}\alpha^k)$$

since both the number p of basic solutions as well as all λ_1, $\lambda_2, \ldots, \lambda_p$ in equation (4.3) are constants.

For recurrence (4.1), due to its special form, it is easy to determine a maximum characteristic root, as we will describe in the following. We will, firstly, show that exactly one of the characteristic roots is real and positive, and, moreover, that this root is a single root. Secondly, we will show that this real and positive root is a maximum characteristic root.

Lemma 4.5.3. *Let* $p(z) := z^k - z^{k-k_1} - z^{k-k_2} - \ldots - z^{k-k_r}$ *for integer* $r > 1$*, integer* $k \geq 1$*, and positive integers* $k_1, k_2, \ldots, k_r$ *with* $k_i < k$*,* $i = 1, \ldots, r$*. Then,* $p(z)$ *has exactly one positive, real root* α*. Additionally,* $\alpha > 1$ *and* α *is a single root.*

Proof. Instead of $p(z)$, we consider

$$q(z) := 1 - z^{k_1} - z^{k_2} - \ldots - z^{k_r}.$$

The relation between $p(z)$ and $q(z)$ is, since $p(z)/z^k = 1 - (1/z)^{k_1} - (1/z)^{k_2} - \ldots - (1/z)^{k_r}$, given by

$$q(1/z) = p(z)/z^k.$$

Therefore, β is a positive, real root of $q(z)$ iff $1/\beta$ is a positive, real root of $p(z)$. In the following, we examine the positive, real roots of $q(z)$ and, using this detour, draw conclusions about the positive, real roots of $p(z)$.

We have $q(0) = 1$ and $q(1) = 1 - r$. Since $r \geq 2$ and $k_i \geq 1$ for all $i = 1, \ldots, r$, we have $q(1) \leq -1$ and, moreover, $q(z) \leq -1$ for $z > 1$. Since $q(z)$ is continuous and strictly decreasing for $z > 0$, $q(z)$ has exactly one real root β with $0 < \beta < 1$. Since $q(z)$ is negative for $z \geq 1$, this is the only positive, real root of $q(z)$. Consequently, also $p(z)$ has only one real positive root, namely $\alpha := 1/\beta$, and, since $0 < \beta < 1$, we have $\alpha > 1$.

Considering the derivation

$$q'(z) = -k_1 z^{k_1 - 1} - k_2 z^{k_2 - 1} - \ldots - k_r z^{k_r - 1},$$

we have $q'(z) < 0$ for $z > 0$ and, therefore, $q'(\beta) < 0$. This shows that β is a single root since a multiple root β' would, due to standard derivation rules, imply $q'(z) = 0$. Consequently, $\alpha := 1/\beta$ is also a single root of $p(z)$. □

It remains to show that α as determined in Lemma 4.5.3 is a maximum characteristic root. To this end, we introduce *generating functions* which play a central role in the solution of recurrence relations [177]. Let us consider the elements T_k computed by the recurrence relation (4.1) as a sequence $(T_k)_k$. With the sequence elements as coefficients we build the power series

$$t(z) = \sum_{k \geq 0} T_k z^k. \tag{4.4}$$

It has a positive radius of convergence r which is given by $\frac{1}{r} = \lim_{k \to \infty}(\sqrt[k]{T_k})$ [31]. We call such a power series with positive radius of convergence the *generating function* of sequence $(T_k)_k$. There is a close relation between generating function (4.4) and the characteristic polynomial (4.2):

Theorem 4.5.4 ([177], Theorem 3.3). *The generating function*

$$t(z) = \sum_{k \geq 0} T_k z^k$$

is a rational function

$$t(z) = f(z)/q(z),$$

where $q(z) = 1 - z^{k_1} - z^{k_2} - \ldots - z^{k_r}$ *and* $f(z)$ *is determined by the initial values* $T_0, T_1, \ldots, T_{d-1}$ *where* $d = \max(k_1, k_2, \ldots, k_r) - 1$. □

Polynomial $q(z)$ as defined in Theorem 4.5.4 has a root β iff the characteristic polynomial $p(z)$ as defined in (4.2) has a root $1/\beta$ (for an explanation refer to the proof of Lemma 4.5.3). Moreover, a root β of $q(z)$ is a *singularity* of generating function $t(z)$, i.e., at β the function is not complex-differentiable.

Summarizing, on the one hand, recurrence (4.1) corresponds to a characteristic polynomial $p(z)$ as given in (4.2). On the other hand, recurrence (4.1) corresponds to a generating function $t(z)$ as given in (4.4) and $t(z) = f(z)/q(z)$ for a polynomial $q(z)$ with the property that $q(1/z) = p(z)/z^k$ (for details see proof of Lemma 4.5.3). Consequently, a root α of polynomial $p(z)$ corresponds to a root $1/\alpha$ of polynomial $q(z)$ and, thus, to a singularity of generating function $t(z)$. Moreover, a maximum characteristic root α of the characteristic polynomial (4.2) corresponds to a *dominant singularity* of generating function $t(z)$, i.e., to a singularity with minimum absolute value. The following classical "Pringsheim" theorem concludes our argumentation:

Theorem 4.5.5 ([72], Theorem 4.3). *A generating function with finite radius of convergence and non-negative coefficients has a dominant singularity that is real and positive.* □

With Lemma 4.5.3 and Theorem 4.5.5, we conclude that the unique positive, real root α of polynomial 4.2 corresponds to a dominant singularity and, therefore, is a maximum characteristic root. Thus, we have $T_k = O(\alpha^k)$. We call α the *branching number* for the branching vector $(k_1, k_2, \ldots, k_r)$.

In summary, this section showed how to compute the branching number from the branching vector: We compute the unique positive, real root α of the characteristic polynomial (4.2), e.g., using the Newton method as explained in [79]. Given a set of branching rules, we determine the branching number for every rule. Then, the branching rule with the largest branching number determines the worst-case upper bound on the search tree size since, as the worst case, we assume that all branchings are of this kind.

Example. A new branching rule for Vertex Cover **(continued).** With the new knowledge we can give a new upper bound on the search tree based on the branching rule presented above. The number of leaves in the search tree is given by recurrence

$$T_k = T_{k-1} + T_{k-d},$$

with $d \geq 3$. Therefore, the worst-case branching vector of this branching is $(1, 3)$ and we compute a corresponding branching number of approximately 1.466 (branching vectors $(1, d)$ with $d > 3$ yield larger branching numbers): 1.466^k is an upper bound on the search tree size.

Recently, there has been active research on search tree algorithms for Vertex Cover [9, 43, 150, 154, 187], now giving upper bounds on the search tree size better than 1.29^k [43, 154]. □

Chapter 5

Consensus of Sequences

Finding signals in DNA is a major problem in computational biology. A recently intensively studied facet of this problem is based on consensus word analysis [164, Section 8.6]. Central problems herein are Closest String and Closest Substring defined as follows; recall that we use $d_H(s, s_i)$ to denote the Hamming distance between strings s and s_i:

> Closest String
> **Input:** Strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a non-negative integer d.
> **Question:** Is there a string s of length L such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?

> Closest Substring
> **Input:** Strings $s_1, s_2, \ldots, s_k$ over alphabet Σ, and non-negative integers d and L.
> **Question:** Is there a string s of length L such that, for every $i = 1, \ldots, k$, there is a length-L substring s'_i of s_i with $d_H(s, s'_i) \leq d$?

Closest String[1] is also known as Consensus String, Center String, or Minimum Radius problem; here, we adopt the notation of [128, 129]. The solution string s which has Hamming distance at most d to all given strings will, in the following, be referred to as *center string*.

While Closest Substring has its main application in motif search [27, 35, 165], Closest String, besides being a special case and a subproblem of

[1] From a linguistic point of view, a "closest" string would mean a string for which the maximum Hamming distance to one of the given strings is as small as possible and, thus, the term would refer to the optimization version of the problem. Here, however, we refer to the decision version of the problem. Identifying the parameterized decision version of a problem with a name more appropriate for its optimization version is commonly used in complexity theory, e.g., for Longest Common Subsequence (Section 3.2) and many other problems [58].

Closest Substring, can, e.g., be used for the unbiased representation of sequences [19] and it is a special case of a problem in which all internal nodes of a given evolutionary tree have to be labeled [75].

What is currently known about these two problems as well as some main results from this chapter are summarized in the following and in Table 5.1.

1. Closest String and Closest Substring are NP-complete even for binary alphabet [73, 124].
2. On the positive side, Closest String and Closest Substring admit polynomial-time approximation schemes (PTAS's), where the objective function is the minimum Hamming distance d which allows to find a center string or substring of length L, respectively [129].
3. In the PTAS's for both Closest String and Closest Substring, the exponent of the polynomial bounding the running time depends on the goodness of the approximation. These are not *efficient* PTAS's (EPTAS's) in the sense of [41] and therefore are probably not useful for bioinformatics practice. Whether EPTAS's are possible for these approximation problems, currently remains open.
4. Closest String is solvable in $L^{O(k)}$ time [76] and solvable in linear time for $d = 1$ [188].

Our Main Results in this Chapter.

1. Closest String is *fixed-parameter tractable* with respect to parameter d, and it can be solved in $O(kL + kd \cdot d^d)$ time, even for unbounded alphabet size (Subsection 5.1.2); this algorithm generalizes to closely related versions of the problem.
2. Closest String is also fixed-parameter tractable with respect to the parameter k, even for unbounded alphabet size, but here the exponential parametric function is much faster growing, and the algorithm is probably of less practical use (Subsection 5.1.3).
3. For unbounded alphabet size, Closest Substring is W[1]-hard for the combined parameters L, d, and k (Subsection 5.2.2).
4. Closest Substring is W[1]-hard with respect to parameter k, i.e., the number of input strings, even in case of a binary alphabet (Subsection 5.2.3); our hardness results for Closest Substring can also be adapted for the related Consensus Patterns problem (see Section 5.2).

Thus, this work gives the first strong theory-based support for the common intuition that Closest Substring (W[1]-hard) seems to be a much harder

	Closest String	Closest Substring
NP-completeness theory	NP-complete	NP-complete
Approximation theory	PTAS [129]	PTAS [129]
Parameterized complexity		
w.r.t. parameter d	FPT$^{(*)}$ (Sect. 5.1.2)	(open)
w.r.t. parameter k	FPT$^{(*)}$ (Sect. 5.1.3)	W[1]-hard$^{(*)}$ (Sect. 5.2.3)

Table 5.1: Overview on old and new results for Closest String and Closest Substring. Results from this work are marked by $(*)$. Here, we assume the practical case of constant alphabet size, although the mentioned FPT results hold also for unbounded alphabet size, and the W[1]-hardness result even holds for binary alphabet.

problem than Closest String (in FPT). Notably, this could *not* be expressed by "classical complexity measures" since both problems are NP-complete as well as both do have a PTAS (also see Table 5.1).

This chapter is organized as follows. Section 5.1 contains our results for Closest String and some closely related problems. Section 5.2 contains our hardness results for Closest Substring.

5.1 Part I: Closest String and Related Problems

In this section, we study Closest String and related problems, namely the d-Mismatch problem (which generalizes Closest String in the way that we look for center strings of *aligned* substrings of a given set of strings) [188, 189] and the Distinguishing String Selection problem [124].[2] For a brief overview on biological applications concerning signal finding and primer design, refer to, e.g., [124]. All these problems are, in general, NP-hard [73, 124]. Hence, polynomial-time algorithms are out of reach.

Despite their hardness, these problems need to be solved in practice. One line of research is thus to study their approximability. Improving previous work [75, 124], Li *et al.* [129] finally came up with a polynomial-time approximation scheme (PTAS) for Closest String. The constants and degrees of polynomials occurring in the running time, however, make this result of little practical value. Another very promising approach is to study the pa-

[2] We follow the recent work of Deng *et al.* [56] which tells apart the problems Distinguishing String Selection (where all given input strings and the goal string have exactly the same length L) and the computationally still harder Distinguishing Substring Selection (which is defined relative to substrings and the given input strings may have varying lengths). Note, however, that Lanctot *et al.* [124] who originally proposed these problems did not make this distinction and referred to both problems as Distinguishing String Selection.

rameterized complexity with a focus on the two most natural parameters of Closest String: the maximum Hamming distance d allowed and the number k of given input strings. Under the natural assumption that either d or k is (very) small (in particular, in biological applications it is appropriate to assume small d, e.g., d smaller than 10 [67, 124]), it is important to ask whether efficient polynomial or, even better, linear-time algorithms are possible when d or k are constant.

We present the following results:

1. Closest String can be solved in $O(kL + kd \cdot d^d)$ time, yielding a linear-time search tree algorithm for constant d. This answers the open question of Evans and Wareham [67] for the parameterized complexity of Closest String with parameter d.

2. d-Mismatch can be solved in linear time for constant d, which improves work by Stojanovic *et al.* [188] who gave a linear-time algorithm for only $d = 1$; thus, we positively answer their open question for generalizing their result to $d > 1$.

3. Our result is also extendible to Distinguishing String Selection, for which we can derive a linear-time algorithm in case of constant distance parameters and constant alphabet size. (Note that, here, we clearly distinguish, in analogy to the difference between Closest String and Closest Substring, between Distinguishing String Selection and Distinguishing Substring Selection as it is also suggested in [56]; for more details refer to Subsection 5.1.2.3).

4. Using a new ILP formulation of the problem, we also show that Closest String is fixed-parameter tractable with respect to k, i.e., the number of input strings (answering an open question of [67]). To our best knowledge, this is the first time that ILP's are used to show fixed-parameter tractability, while, up to now, it was not possible to obtain this result in some other way.

5.1.1 Preliminaries on Closest Strings

In this section, we give definitions and small results that will be useful later on.

Given a set of k strings, each of length L, we can think of these strings as a $k \times L$ character matrix. By *columns* of the set of strings, we refer to the columns of this matrix.

A string s is an *optimal center string* for S iff there is no string s' with $\max_{i=1,\dots,k} d_H(s', s_i) < \max_{i=1,\dots,k} d_H(s, s_i)$. By way of contrast, s is an *optimal median string* for S iff there is no string s' with $\sum_{i=1,\dots,k} d_H(s', s_i) < \sum_{i=1,\dots,k} d_H(s, s_i)$.

Note that, given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, an optimal median string can be easily computed by choosing in every column a letter occurring most often. If a letter is chosen in this way, we call it *majority vote*; it is, however, not necessarily unique.

In the following, we state that after reordering the columns of the Closest String instance, we can easily obtain solutions for the original instance from solutions for the reordered instance. For reordering the columns, we introduce a permutation on strings as follows. Given a string $s = c_1c_2 \ldots c_L$ of length L with $c_1, \ldots, c_L \in \Sigma$ for alphabet Σ and a permutation $\pi\colon \{1, \ldots, L\} \to \{1, \ldots, L\}$. Then, $\pi(s) = c_{\pi(1)}c_{\pi(2)} \ldots c_{\pi(L)}$. The following lemma is obvious.

Lemma 5.1.1. *Given a set of strings* $S = \{s_1, s_2, \ldots, s_k\}$*, each of length* L*, and a permutation* $\pi\colon \{1, \ldots, L\} \to \{1, \ldots, L\}$*. Then* s *is an optimal center string for* $\{s_1, s_2, \ldots, s_k\}$ *iff* $\pi(s)$ *is an optimal center string for* $\{\pi(s_1), \pi(s_2), \ldots, \pi(s_k)\}$*.* □

Several columns can be identified as equivalent due to *isomorphism*. The reason for this is the fact that the columns are independent from each other in the sense that the distance from the center string is measured columnwise. For instance, consider the case of the two columns $(a, a, b)^t$ and $(b, b, a)^t$ when $k = 3$. Clearly, these two columns are isomorphic because they express the same structure: The first and the second column entry are equal and they differ from the third column entry. For finding the optimal center string, however, only this kind of structure matters. More precisely, we consider two length k columns $\vec{c}$ and $\vec{c}'$ as isomorphic if, for all column entries $\vec{c}[i]$ and $\vec{c}'[j]$, $1 \leq i < j \leq k$, $\vec{c}[i] = \vec{c}[j]$ iff $\vec{c}'[i] = \vec{c}'[j]$. Isomorphic columns form *column types*.

This can be generalized as follows. Without loss of generality, let a always denote the letter that occurs most often in a column, let b always denote the letter that has the secondly most often occurrences and so on. This property of being *normalized*, as we will refer to it in the following, can be easily achieved by a simple linear-time preprocessing of the input instance. In addition, solving the normalized problem optimally, one again can compute the optimal solution of the original problem instance by simply reversing the above mapping done by the preprocessing. Hence:

Lemma 5.1.2. *To compute an optimal center string, it is sufficient to solve a normalized and reordered instance. From this, the solution of the original instance can be derived in linear time.* □

In the following, we call two input instances *isomorphic* if there is a one-to-one correspondence between the columns of both instances such that each thus determined pair of columns is isomorphic. The following lemma shows that it is sufficient to solve an instance with alphabet size $|\Sigma'| \leq k$.

Lemma 5.1.3. *A* Closest String *instance with arbitrary alphabet* Σ*,* $|\Sigma| > k$*, is isomorphic to a* Closest String *instance with alphabet* Σ'*,* $|\Sigma'| = k$*.*

Proof. A normalized instance has automatically at most k symbols. □

With the following observation by Evans and Wareham [67], we find that it is sufficient to solve instances containing at most kd columns. This yields a problem kernel with respect to parameters k and d together, which allows an efficient preprocessing of the input instance. We call a column *dirty* iff it contains at least two different symbols from alphabet Σ. Clearly, all the work in solving Closest String concentrates on the dirty columns of the input instance.

Lemma 5.1.4. *Given a* Closest String *instance with* k *strings* $s_1, \dots, s_k$ *of length* L *and a non-negative integer* d*. If the resulting* $k \times L$ *matrix has more than* kd *dirty columns then there is no string* s *with* $\max_{i=1,\dots,k} d_H(s, s_i) \leq d$.

Proof. Each of the k input strings differs from a solution in at most d positions. These mismatches give rise to dirty columns. Since there are k strings with at most d mismatches each, an instance can have a solution only when there are at most kd dirty columns. □

5.1.2 A Linear-Time Solution for Constant Distance Parameter

We show that Closest String, although NP-complete in general, is solvable in linear time for constant d and, in particular, that it is fixed-parameter tractable with respect to the distance parameter d. In the first subsection, we give the basic algorithm and then we show that it has a linear running time for constant d. In the subsequent two subsections, we present heuristic improvements of the strategy and enhancements, e.g., solving the more general d-Mismatch and the Distinguishing String Selection problems.

5.1.2.1 Bounded Search Tree Algorithm

In Fig. 5.1, we outline a recursive procedure solving Closest String. For the correctness of the algorithm we need the following simple observation.

Lemma 5.1.5. *Given a set of strings* $S = \{s_1, s_2, \dots, s_k\}$ *and a positive integer* d*. If there are* $i, j \in \{1, \dots, k\}$ *with* $d_H(s_i, s_j) > 2d$ *then there is no string* s *with* $\max_{i=1,\dots,k} d_H(s, s_i) \leq d$.

Proof. The Hamming distance satisfies the triangle inequality. If $d_H(s_i, s_j) > 2d$ and we are given an arbitrary string s then we know that $d_H(s, s_i) + d_H(s, s_j) > 2d$. It follows that $d_H(s, s_i) > d$ or $d_H(s, s_j) > d$ (or both). □

The idea of our strategy is to start with one of the given strings, e.g., s_1, as a "candidate string." If there is a string s_i, $i = 2, \dots, k$, that differs from

```
Recursive procedure CSd(s, Δd):
Global variables: Set of strings $S = \{s_1, s_2, \ldots, s_k\}$, non-negative integer $d$.
Input: Candidate string $s$ and integer $\Delta d$.
Output: A string $\hat{s}$ with $\max_{i=1,\ldots,k} d_H(\hat{s}, s_i) \leq d$ and $d_H(\hat{s}, s) \leq \Delta d$, if it
        exists, and "not found," otherwise.
Method:
(Case 0) if $(\Delta d < 0)$ then return "not found";
(Case 1) if $(d_H(s, s_i) > d + \Delta d)$ for some $i \in \{1, \ldots, k\}$ then
                                                        return "not found";
(Case 2) if $(d_H(s, s_i) \leq d)$ for all $i = 1, \ldots, k$ then return $s$;
(Case 3) choose any $i \in \{1, \ldots, k\}$ such that $d_H(s, s_i) > d$:
        $P := \{ p \mid s[p] \neq s_i[p] \}$;
        choose any $P' \subseteq P$ with $|P'| = d + 1$;
        for all $p \in P'$ do
            $s' := s$;
            $s'[p] := s_i[p]$;
            $s_{ret} :=$ CSd$(s', \Delta d - 1)$;
            if $s_{ret} \neq$"not found" then return $s_{ret}$;
        return "not found"
```

Figure 5.1: **Algorithm** CSd. Inputs are a Closest String instance consisting of a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ of length L, and an integer d. First, we perform a preprocessing performing the reduction to a problem kernel as shown in Lemma 5.1.4: We select the dirty columns. If there are more than kd many then we reject the instance. If there are at most kd many then we invoke the recursion with *CSd*(s_1, d).

the candidate string in more than d positions, we recursively try several ways to move the candidate string "towards" s_i; moving closer here means that we select a position in which the candidate string and s_i differ and set this position in the candidate string to the character of s_i at this position. We stop either if we moved the candidate "too far away" from s_1 or if we found a solution. By a careful selection of subcases of this recursion we can limit the size of this search tree to $O(d^d)$, as will be shown in the following theorem.

Theorem 5.1.6. *Given a set of strings* $S = \{s_1, s_2, \ldots, s_k\}$*, and an integer* d*, Algorithm* CSd *(Fig. 5.1) determines in* $O(kL + kd \cdot d^d)$ *time whether there is a string* s *with* $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$ *and it computes such an* s *if one exists.*

Proof. **Running time.** Prior to the recursion, we perform the reduction to a problem kernel as described in Lemma 5.1.4. This preprocessing reduces the size of the input instance to $O(kd)$ and can be done in $O(kL)$ time.

Now, we consider the recursive part of the algorithm. Parameter Δd is initialized to d. Every recursive call decreases Δd by one. The algorithm stops when

$\Delta d < 0$. Therefore, the algorithm builds a search tree of height at most d. In one step of the recursion, the algorithm chooses, given the current candidate string s, a string s_i such that $d_H(s, s_i) > d$. It creates a subcase for $d + 1$ of the positions in which s and s_i disagree (there are more than d but at most $2d$ such positions). This yields an upper bound of $(d+1)^d$ on the search tree size.[3]

Each step of the recursion needs only $O(kd)$, i.e., linear, time: Before starting the recursion, we build a table containing the distances of the candidate s_1 to all other given strings. Using this table, instructions (Case 1) and (Case 2) can be done in $O(k)$ time. In instruction (Case 3), we need $O(k)$ time to select the s_i for branching and $O(kd)$ time (observe Lemma 5.1.4) to find the positions in which s and s_i differ. For $d+1$ of those differing positions (there are at most $2d$ many) we modify the candidate string, update the table of distances, and call the procedure recursively. Since we changed only one position, we can update the table of distances in $O(k)$ time.

Correctness. We have to show that Algorithm CSd will find a string s with $\max_{i=1,\dots,k} d_H(s, s_i) \leq d$, if such an s exists. The preprocessing is correct by Lemma 5.1.4. Regarding the search tree, we explicitly show only the correctness of the first recursive step; the correctness of the algorithm then follows with an inductive application of the argument.

In the situation that s_1 satisfies $\max_{i=1,\dots,k} d_H(s_1, s_i) \leq d$, we immediately find a solution, namely s_1. If s_1 is not a solution but there exists a center string s for this instance with distance value d, then there is a string s_i, $i = 2, \dots, k$, such that $d_H(s_1, s_i) > d$. For branching, we consider the positions where s_1 and s_i differ, i.e., $P := \{\, p \mid s_1[p] \neq s_i[p] \,\}$. Algorithm CSd successively creates subcases for $d + 1$ positions p from P in order to create a new candidate by altering the respective position p from $s_1[p]$ to $s_i[p]$. Such a "move" is correct if we choose a position p from $P_{s_1 \neq s = s_i} := \{\, p \mid s_1[p] \neq s[p] = s_i[p] \,\}$. Now, we show that (at least) one of our $d + 1$ moves is a correct one. We observe that P is the disjoint union of $P_{s_1 \neq s = s_i}$ and $P_{s \neq s_i} := \{\, p \mid s[p] \neq s_i[p] \,\}$. Since $d_H(s, s_i) \leq d$ we know that $|P_{s \neq s_i}| \leq d$. Therefore, at least one of our $d + 1$ subcases will try a position from $P_{s_1 \neq s = s_i}$.

Regarding instruction (Case 1), we can analogously to Lemma 5.1.5 observe that it is correct to omit those branches where the candidate string s satisfies $d_H(s, s_i) > d + \Delta d$ for some string s_i of the given strings $s_1, \dots, s_k$. □

With Algorithm CSd, we find a solution if one exists. Within the given time bounds, we can easily modify Algorithm CSd to find *all* solutions if the given distance parameter d is optimal, described as follows. Firstly, we do, in (Case 3),

[3] If there are two strings s_i, s_j with $d_H(s_i, s_j) = 2d$, we can use a better strategy: We know that a solution has to differ from both s_i and s_j in d positions. We can search a solution by trying all ways to partition the set of positions p with $s_i[p] \neq s_j[p]$ into two sets of size d. In the candidate, we give to one set of positions the characters of s_i, to the second set the characters of s_j. The resulting running time is $O(kL + kd \cdot 2^{2d})$.

not stop the recursion as soon as a solution string is found but perform a recursive call for *every* $p \in P'$, returning not before the **for**-loop has been processed for every $p \in P'$. Secondly, it may occur that we have found, in (Case 2), a solution s while $\Delta d > 0$. Then, however, there has to be s_j, $j \in \{1, \ldots, k\}$ with $d_H(s, s_i) = d$ since, otherwise d would not be optimal. For a solution $s' \neq s$ it has to hold that $s'[q] = s_i[q]$ for at least one $q \in Q$ where $Q = \{\, p \mid s[q] \neq s_i[q] \,\}$. Therfore, we invoke, for every $q \in Q$, a recursive call CSd$(s', \Delta d - 1)$ where s' is obtained from s by changing $s[q]$ to $s_i[q]$. Since $|Q| \leq d$, this does not sacrifice the time bounds of the algorithm given in Theorem 5.1.6. Analogously as in the proof of Theorem 5.1.6, it is shown that, by these modifications, we find *all* solutions if the given distance parameter d is optimal. However, we cannot, within these time bounds, necessarily find all solutions to a given instance when d is not optimal as we explain in the first paragraph of Section 5.1.2.3.

We can find the optimal d by starting from $d = 0$ and increasing it by 1 until a solution is found. In contrast to knowing d in advance this costs only a constant factor in the running time.

Note that the interleaving technique described in [151] to build a new problem kernel on every level of the search tree seems not to be of use here because changing levels of recursion do not influence the size of the problem kernel.

5.1.2.2 Heuristic Improvements

Avoiding the multiple traversal of subtrees. Since the search tree size is the critical factor in the algorithm's running time, the goal is to keep it as small as possible. In particular, there is no use in visiting the same "configurations" twice or more times. In the following, we describe two ways to avoid visiting such useless branches of the search tree.

Keeping in mind the initial candidate string, there is no use in changing a position that has already been changed before. We can avoid this by testing for each position considered for a change whether it still equals the position in the initial candidate. We branch only if it does.

Still, we will run into situations where the same candidate string is considered multiple numbers of times. For instance, when the solution differs from the candidate in positions p_1 and p_2, we find it by changing first p_1 and then p_2, and we can also find it by changing first p_2 and then p_1. We can avoid finding the same solutions several times in the following way. Recall that the search tree is traversed in a depth-first manner. We maintain, for every position p, the character set C_p such that $c \in C_p$ if a length-L string s with $s[p] = c$ is candidate string in a search tree node that is an ancestor of the current node (including the current node itself) or an already traversed sibling of these nodes (see Fig. 5.2). For branching, we consider only changing a position p in the candidate to a character c if $c \notin C_p$. In the following, we sketch why

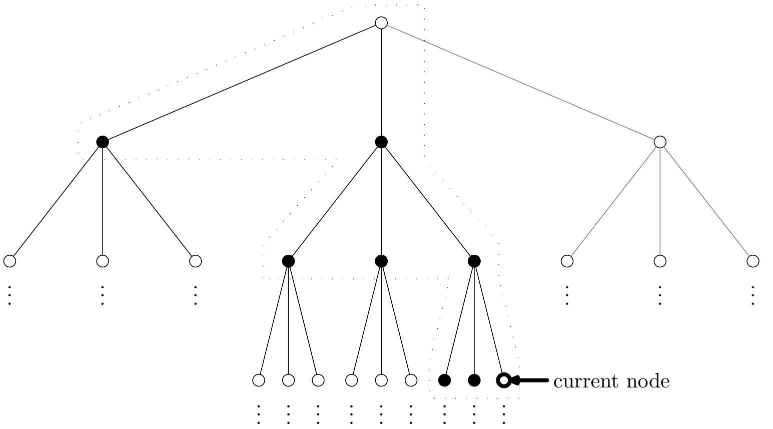


Figure 5.2: Illustration of our strategy to avoid computing the same solution several times. Outlined is a search tree which is traversed depth-first. For the current search tree node, we store those changes of the candidate string which are made in the node's ancestors (including the node itself) and already traversed siblings of these nodes. Hence, in the current node, when branching, we omit to make changes to the candidate string that have already been tried in one of the black nodes.

we can omit the choice of $c \in C_p$. Without loss of generality, we assume that we started the algorithm with the initial candidate string s_1. Given a search tree node with candidate string s, we call a solution s^* *descendant* of s if it satisfies, for all $1 \leq p \leq L$, either $s_1[p] = s[p] = s^*[p]$, or $s_1[p] \neq s[p] = s^*[p]$, or $s_1[p] = s[p] \neq s^*[p]$. In analogy to the proof of Theorem 5.1.6, we can, then, show that a solution s^* which is a descendant of a candidate string s can be found in the part of the search tree rooted at the search tree node having s as a candidate string. Using this observation, we can prove the following claim: Assuming that we find a solution by changing a position p in the candidate to a character c with $c \in C_p$, this solution is a descendant of a candidate string corresponding to an already traversed search tree node, more precisely, corresponding to an already traversed sibling of an ancestor of the current search tree node. Therefore, this solution has already been found by the search tree. Moreover, it can also be shown that, by this strategy, no solution is found more than once. A detailed proof for these claims can be derived in a similar way as in the proof of Theorem 5.1.6.

Note that there are at most k many characters in C_p for a position p, and, therefore, this enhancement can be done within the time bounds given for Algorithm CSd.

Selection of the candidate string to start with. In Algorithm CSd, we initialized the candidate string s to s_1. We are, however, free to choose the initial candidate string from all s_i, $i = 1, \ldots, k$. Since we search the solution in the neighborhood of the initial candidate string s, a good choice is an s which is presumably close to the (unknown) solutions. A possible strategy is to select the string with a minimum median distance to all other strings.

Improved problem kernel. In Subsection 5.1.1 we explained how to compute a problem kernel of size $O(kd)$. Taking into account the number of different symbols occurring in a dirty column, we can improve this result in a heuristic way as follows. For the ith column $\vec{c_i}$ of a given instance, we denote the number of different symbols occurring in $\vec{c_i}$ by $\#(\vec{c_i})$. In analogy to the proof of Lemma 5.1.4, we can easily show that the given instance cannot have a solution if $\sum_{i=1}^{L}(\#(\vec{c_i}) - 1) > kd$. Taking additionally into account the number of occurrences of a symbol in a dirty column, even more refined versions of a problem kernel rule are possible but we omit further details here.

5.1.2.3 Enhancements and Related Problems

Finding all solutions. So far, we assumed that Algorithm CSd is invoked with a minimal distance parameter d or the minimal d is found. In some situations, it may, however, be desirable for the user to know all solutions that can be found with a non-minimal d. We already mentioned in Subsection 5.1.2.1 that Algorithm D, invoked with non-optimal d, does not necessarily compute *all* solutions for a given instance. This is not possible in the claimed time bound since there may be more than $O(kd \cdot d^d)$ solutions, e.g., for an instance containing only one input string of length L and allowing $d = 1$ mismatches. We can, however, extend the given algorithm such that, even for non-minimal d, it expands the set of found solutions to the set of all solutions. If we find a solution after changing d' positions in the initial candidate string, we are allowed to change $\Delta d = d - d'$ further positions. We can try all ways to change Δd of those positions in which the solution still equals the initial candidate string. A recursive strategy similar as the one in the main algorithm allows to further prune this search space, e.g., we only have to consider changes that do not increase the distance to one of the input strings to a value larger than d. The upper bound for the number of branching subcases is L, yielding a worst-case time bound of $O(L^{\Delta d}|\Sigma^{\Delta d}|)$ for the expansion of a solution. In practice, we will often find the solutions with a small value of Δd. Therefore, for finding *all* solutions, our strategy of "expanding" the solutions of Algorithm D will still be superior to a brute-force check of all strings of length L in $O(|\Sigma|^L \cdot kL)$ time.

Solving the d-MISMATCH problem. Let $s_{i,p,L}$ denote the length-L substring of a given string s_i starting at position p, i.e., $s_{i,p,L} = s_i[p]s_i[p+1] \ldots s_i[p+L-1]$. Then, given strings $s_1, s_2, \ldots, s_k$ of length n and non-negative integers k and L, the d-MISMATCH problem is the question of whether there is a string s of length L and a position p with $1 \leq p \leq n - L + 1$, such that $d_H(s, s_{i,p,L}) \leq d$ for all $i = 1, \ldots, k$. For $n = L$, this problem is equivalent to CLOSEST STRING.

Stojanovic *et al.* [188, 189] give a linear-time algorithm for the 1-MISMATCH problem and report about its use in practice. For constant $d > 1$, they explicitly ask for a polynomial-time algorithm.

We can solve d-MISMATCH using Algorithm CSd in linear time for constant d. A trivial solution is to invoke the algorithm for each of the $n - L + 1$ possible positions in the $n \times k$ matrix which is induced by the input. This yields a worst-case running time of $O(n \cdot (kL + kd \cdot d^d))$, i.e., a quadratic[4] running time for constant d. Improving this, we achieve a linear running time for constant d as follows. We use the problem kernel of size kd for CLOSEST STRING as given in Lemma 5.1.4. Considering only the first L columns of the $n \times k$ matrix, we can, in $O(kL)$ time, build a FIFO queue of dirty columns. We update this queue in $O(k)$ time when shifting the window of L consecutive columns under consideration from position p (containing columns p to $p+L-1$) to position $p+1$ (containing columns $p + 1$ to $p + L$), $p = 1, \ldots, n - L$. (1) If column p is dirty, we delete it from the front end of the queue. (2) If the "new" column $p + L$ is dirty, we append it to the back end of the queue.

Thus, we can maintain the queue of dirty columns, at each position taking only $O(k)$ time. After a one-position-shift in the $n \times k$ matrix, Algorithm CSd is invoked on the columns in the queue only if the queue contains at most kd columns. The described strategy yields the following result.

Theorem 5.1.7. d-MISMATCH *is solvable in* $O(kL+(n-L)kd \cdot d^d)$ *time which is* $O(n \cdot k)$ *for constant* d. □

Solving the DISTINGUISHING STRING SELECTION **(DSS) problem.** Following Lanctot *et al.* [124], in this problem, we are given "bad" strings $s_1, \ldots, s_{k_1}$, "good" strings $s'_1, \ldots, s'_{k_2}$ all of same length L, and non-negative integers d_1, d_2. We are looking for a solution string s that is close to the bad strings, i.e.,

$$\max_{i=1,\ldots,k_1} d_H(s, s_i) \leq d_1,$$

and far away from the good strings, i.e.,

$$\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2.^5$$

We follow Deng *et al.* [56] in the sense that we tell apart DISTINGUISHING SUBSTRING SELECTION (DSSS) (where the input strings may differ in lengths) and the special case DSS.[6] This distinction is not made by Lanctot *et al.* [124], who

[4] Note that the input size is nk and that $L \leq n$.

[5] This terminology of "good" and "bad" introduced by Lanctot *et al.* [124] has its motivation in the application scenario of designing genetic markers in order to distinguish the sequences of harmful bacteria (to which the markers should bind) from human sequences (to which the markers should not bind).

[6] Actually, according to Deng *et al.* in DISTINGUISHING SUBSTRING SELECTION only the "bad" strings do have varying lengths, the "good" strings as well as the goal string all being of length exactly L, see [56] for details.

give a polynomial-time factor-2 approximation algorithm for the more general DSSS problem. We can adapt Algorithm CSd for Closest String to solve DSS.

We start with an observation similar to the one made in Lemma 5.1.5.

Lemma 5.1.8. *Given sets of strings* $S_1 = \{s_1, \ldots, s_{k_1}\}$ *and* $S_2 = \{s'_1, \ldots, s'_{k_2}\}$, *and positive integers* d_1 *and* d_2. *If there are* $i \in \{1, \ldots, k_1\}$ *and* $j \in \{1, \ldots, k_2\}$ *with* $d_H(s_i, s'_j) < L - (d_1 + d_2)$, *then there is no string* s *satisfying both* $\max_{i=1,\ldots,k_1} d_H(s, s_i) \leq d_1$ *and* $\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2$.

Proof. Assume that there are strings s, s_i, s'_j such that (1) s is close to s_i, i.e., $d_H(s, s_i) \leq d_1$, (2) s is far away from s'_j, i.e., $d_H(s, s'_j) \geq L - d_2$, and (3) s_i and s'_j satisfy the lemma's premise, i.e., $d_H(s_i, s'_j) < L - d_1 - d_2$. Since the Hamming distance satisfies the triangle inequality, we know that $d_H(s, s'_j) \leq d_H(s, s_i) + d_H(s_i, s'_j)$. Together with assumptions (1) and (3) this yields $d_H(s_i, s'_j) < L - d_2$. This, however, contradicts assumption (2) and shows that there are no s, s_i, s'_j that meet all three assumptions at the same time. □

In what follows, we describe how to modify Algorithm CSd in order to solve DSS. Using Lemma 5.1.8, we can detect instances that cannot have a solution, i.e., instances where a bad and a good string have Hamming distance less than $L - (d_1 + d_2)$. For this reason, we can extend instruction (Case 1) in Algorithm CSd by returning not only when $d_H(s, s_i) > d_1 + \Delta d_1$ for the candidate s and a bad string s_i but also when $d_H(s, s'_j) < L - (d_2 + \Delta d_1)$ for a good string s'_j.

Of course, a solution in instruction (Case 2) is now found, when the new goal is met, i.e., $\max_{i=1,\ldots,k_1} d_H(s, s_i) \leq d_1$ *and* $\min_{j=1,\ldots,k_2} d_H(s, s'_j) \geq L - d_2$.

Also instruction (Case 3) has to be extended. As long as the branching shown in (Case 3) applies, we still use it: If there is a bad string s_i which our candidate s is too far away from, i.e., $d_H(s, s_i) > d_1$, we branch on $d_1 + 1$ many positions in which s and s_i differ.

When the candidate s satisfies $d_H(s, s_i) \leq d_1$ for all $i = 1, \ldots, k_1$ but it is too close to one of the good strings s'_j, i.e., $d_H(s, s'_j) < L - d_2$, we introduce a new branching. We have to increase $d_H(s, s'_j)$ by changing in s a position p with $s[p] = s'_j[p]$. Since a solution $s^\star$ can have at most d_2 many positions p with $s^\star[p] = s_j[p]$ it is sufficient to branch on $d_2 + 1$ positions with $s[p] = s'_j[p]$. We do, however, not know to which character $s[p]$ should be set. Trying all characters in this situation gives us an upper bound of $(d_2 + 1) \cdot |\Sigma - 1|$ for the subcases to branch into.

Regarding the search tree size, we now have, in every search tree node, one of two possible branchings: In one case, we branch into at most $d_1 + 1$ subcases as in Algorithm CSd. In the second new case, we branch into at most $(d_2+1)|\Sigma-1|$

subcases. The search tree height is limited by d_1. Therefore, the search tree size is at most $(\max(d_1 + 1, (d_2 + 1)|\Sigma - 1|))^{d_1}$.

This yields the following result.

Theorem 5.1.9. DSS *is solvable in* $O((k_1 + k_2)L \cdot (\max(d_1 + 1, (d_2 + 1)(|\Sigma| - 1)))^{d_1})$ *time.* □

By way of contrast, note that very recently parameterized hardness results were obtained for Distinguishing Substring Selection [83].

5.1.3 Constant Number of Input Strings

In this section, we show that Closest String is solvable in linear time for a bounded number k of input strings by using *integer linear programming* [145, 176]. There is a famous result of H. W. Lenstra [125][7] that applies to fixed-parameter algorithms (also see [112, 121] for more details). Lenstra's result basically says that integer linear programs (ILP's for short) with a constant number of variables can be solved in linear time. More precisely, with Kannan's [112] improvements we have the following.

Theorem 5.1.10 (Lenstra). *The integer programming feasibility problem can be solved with* $O(p^{9p/2}L)$ *arithmetic operations with integers of* $O(p^{2p}L)$ *bits in size, where* p *is the number of ILP variables and L is the number of bits in the input.* □

Note that the fixed-parameter result also needs space exponential in the parameter p.

The goal is to give an ILP formulation for Closest String such that the number of variables solely depends on the parameter value k, the number of input strings. The key to this lies in the notion of column types.

Example 5.1.11. *For* $k = 3$*, the set of all possible column types for a* Closest String *instance consists of*

$$(a, a, a)^t, (a, a, b)^t, (a, b, a)^t, (b, a, a)^t, (a, b, c)^t.$$

□

Generally, the number of column types for k strings depends only on k (namely, it is given by the Bell number $B(k) \leq k!$, cf. [156]). Using the column types, Closest String can be formulated as an ILP having only $B(k) \cdot k$ variables. Let the underlying alphabet be Σ. The ILP can be formulated as follows. Since

[7] It won the Fulkerson Prize 1985 as an outstanding paper in the area of discrete mathematics.

we may assume that the instance is normalized (Lemma 5.1.2), every column contains at least one occurrence of symbol a. The ILP uses $B(k) \cdot k$ variables $x_{t,\varphi}$, where t denotes a column type and $\varphi \in \Sigma$. The value of $x_{t,\varphi}$ denotes the number of columns of column type t whose corresponding character in the desired solution string of Closest String is set to φ. Thus, the ILP seeks to minimize

$$\max_{1 \leq i \leq k} \sum_{t} \sum_{\varphi \in (\Sigma - \{\varphi_{t,i}\})} x_{t,\varphi},$$

where $\varphi_{t,i}$ denotes the alphabet symbol at the ith entry of column type t. The following two constraints have to be fulfilled when minimizing the above function.

1. All variables $x_{t,\varphi}$ have to be non-negative integers.
2. Let $\#_t$ denote the number of columns of type t in the input instance (taking into account isomorphism as described before). Then,

   $$\sum_{\varphi \in \Sigma} x_{t,\varphi} = \#_t$$

   for every column type t.

Actually, Theorem 5.1.10 refers to the integer linear programming feasibility problem and, moreover, a Closest String instance also gives the maximum distance d allowed. Thus, we may obtain the following "feasibility formulation" where the above two constraints remain unchanged but the goal function that had to be minimized now translates into a third set of constraints, namely:

$$\sum_{t} \sum_{\varphi \in (\Sigma - \{\varphi_{t,i}\})} x_{t,\varphi} \leq d$$

for every string i, $1 \leq i \leq k$. Altogether, this yields fixed-parameter tractability for Closest String with respect to parameter k. Note, however, that the combinatorial explosion in k is huge and this approach appears to be impractical for large k.

The above ILP approach, however, at least serves as a tool to help deciding whether a problem is fixed-parameter tractable and maybe later it is possible to come up with a more efficient, direct approach to solve the given problem. As to Closest String, the ILP approach is the only one known to us that yields fixed-parameter tractability with respect to parameter k. In [92], a direct combinatorial approach (avoiding ILP's) was given for $k = 3$ but already $k = 4$ remained open due to the enormous combinatorial complexity. Finally, note that there is an alternative ILP formulation for Closest String given by Ben-Dor *et al.* [19], where the variables have only binary values but the number of variables is $|\Sigma| \cdot L$ (for alphabet Σ and string length L). Hence, this ILP formulation does *not* imply the fixed-parameter tractability of Closest String with respect to parameter k. In conclusion, it remains open to give further examples besides Closest String where the described ILP approach turns out to be fruitful. More generally, it would be interesting to see more connections between fixed-parameter algorithms and integer linear programming.

5.1.4 Empirical Results

We report about tests on random instances which were generated as follows. Given as parameters the string length L, the number of strings k, the distance parameter d and the alphabet size $|\Sigma|$, we created a random instance by first computing a random string s of length L. Then, we computed k strings differing from s in d positions, each time switching d randomly chosen positions in s. The displayed results are average results taken from a range of 25 such random instances. We used alphabet size $|\Sigma| = 4$ if not indicated otherwise.

5.1.4.1 Solving Closest String with Algorithm CSd

We implemented Algorithm CSd using the programming language C, including the heuristic improvements discussed in Subsection 5.1.2.2 and also with the optional extension to solve d-Mismatch and DSS as shown in Subsection 5.1.2.3. The following tests were performed on a Linux PC with 750 MHz processor and 192 MB main memory.

For the tests, we used the algorithm that scans the whole search tree for possible solutions, i.e., it does not stop when the first solution is found. Reasons are that one could want to find as many solutions as possible, and that, in this way, the running time is not affected by the location of a solution in the search tree.

Length/mismatch ratio. Our experiments with randomly generated data show that not only the number of mismatches but, moreover, the ratio of string length to the number of mismatches has a major impact on the difficulty of solving the problem. The results from Fig. 5.3(a) show that an increasing length L and a thereby increasing L/d ratio for a fixed value of d, can, in a limited range, significantly decrease the running time of the problem. This can be explained as follows. Due to the random generation of the input, with a larger L the mismatches in the strings are more distributed over the columns than for a smaller L. Therefore, with large L/d ratio, the number of search tree branches for which solutions seem possible will be smaller. In contrary, we encountered large search tree sizes for a small L/d ratio (e.g., we found an average search tree size of 40205 for $L = 20, d = 10, k = 25$ compared to 2733 for $L = 30, d = 10, k = 25$). At some point, of course, this decrease in running time is outweighed by the linear time factor needed for the reduction to the problem kernel.

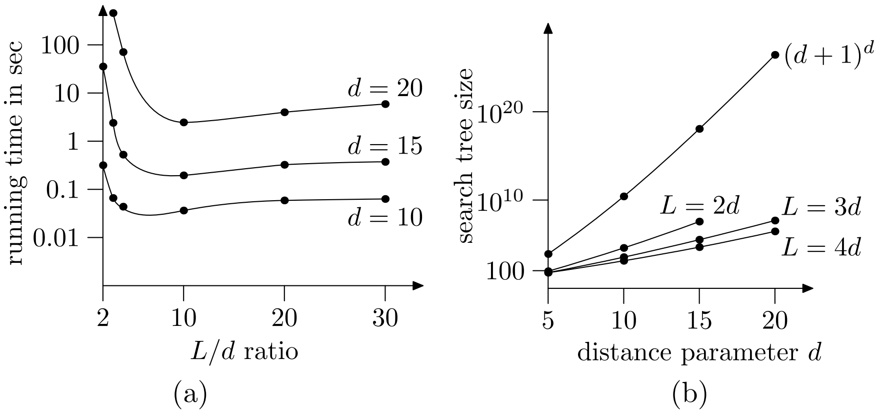


Figure 5.3: (a) Comparing, on a logarithmic scale, the running time of Algorithm CSd on Closest String instances for differing length/mismatch ratio L/d, $|\Sigma| = 4$. (b) Comparing, on a logarithmic scale, search tree sizes of Algorithm CSd with the theoretical upper bound of $(d+1)^d$. Each line displays results for one fixed L/d ratio.

Thus, when considering the values of mismatches d for which we can process Closest String instances in practice, we also have to take the L/d ratio into account. For instance, for a "hard" ratio of 2, i.e., the string length is twice the number of mismatches, we solved instances with $d = 15$ ($L = 30$, $k = 50$) in 200 seconds, and for an "easier" ratio of 3 we solved instances with $d = 20$ ($L = 60$, $k = 50$) in 98 seconds.

Number of input strings. When considering the running times for a variable number of input strings (and fixed values of L, d), we observe two competing factors. On the one hand, an increase in the number of strings means an increase in the linear time factor which has to be spent in every node of the search tree. On the other hand, a growing number of strings means a growing number of constraints on the solutions and, therefore, a decreasing size of the search tree. Our experience with random data sets shows a high running time for small numbers of strings, decreasing with growing number of strings up to some turning point. From then on running time increases again since the linear factor spent in each search tree node becomes crucial. An example: for $L = 24$, $d = 12$, we needed 4.8 seconds for $k = 10$ (search tree size 1305137), 2.7 seconds for $k = 100$ (search tree size 164503), and 6.8 seconds for $k = 400$ (search tree size 55602).

Search tree size. In Fig. 5.3(b), we compare the size of the search tree for given instances with the theoretical upper bound of $(d+1)^d$. Note that the actual search trees are by far smaller than the worst-case bound predicts.

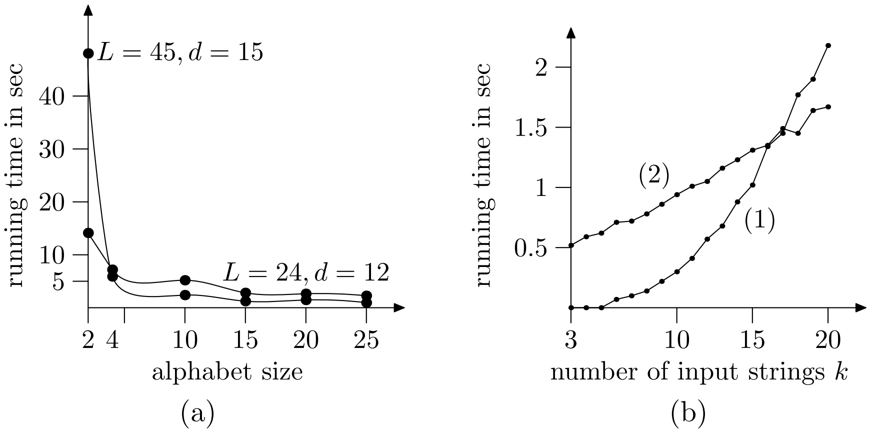


Figure 5.4: (a) Comparing the running times of Algorithm CSd for increasing alphabet size ($k = 50$). (b) Comparing the running times of a heuristic ILP solver using a branch-and-bound strategy for the ILP formulation given in [19] (2) and the one given in this paper (1); we used random instances with $|\Sigma| = 4$, $L = 256$ and $d = 32$.

Alphabet size. Fig. 5.4(a) shows examples of the influence of the alphabet size onto the running time. A very small alphabet size turns out to be harder than a large alphabet size. The reason is the way in which our input instances were generated. With larger alphabet size, the random mismatches that are implanted into the original string will be chosen from a larger set of alphabet symbols and will, therefore, be more "obvious" for the algorithm.

5.1.4.2 Solving Closest String with ILPs

We developed a C++ program using the GNU GLPK library to test the performance of a heuristic ILP solver using a branch-and-bound strategy on ILPs generated from Closest String instances. This strategy guarantees to find an optimal solution but it is heuristic in the sense that it does not give guarantees on the worst-case running time. For our tests, we randomly generated random Closest String instances and translated them both to the ILP formulation given in [19] and to the ILP formulation presented in this section. In Figure 5.4(b), we display the running times of the GNU GLPK solver on instances with $L = 256$ and $d = 32$, measured on a SUN Blade workstation with Ultrasparc IIe processor (500 MHz) and 512 MB main memory. It is no surprise that the ILP formulation presented here turns out to be preferable in case of a small number of long input strings.

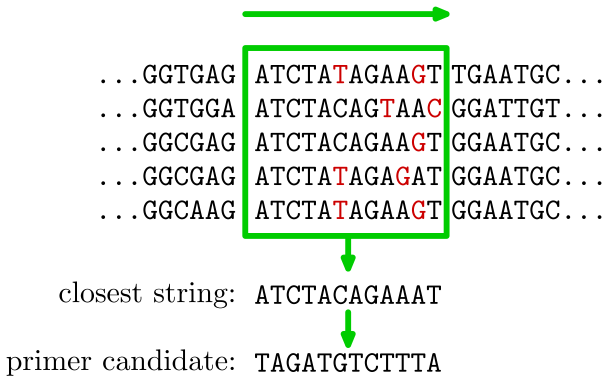

...GGTGAG ATCTATAGAAGT TGAATGC...
...GGTGGA ATCTACAGTAAC GGATTGT...
...GGCGAG ATCTACAGAAGT GGAATGC...
...GGCGAG ATCTATAGAGAT GGAATGC...
...GGCAAG ATCTATAGAAGT GGAATGC...
closest string: ATCTACAGAAAT
primer candidate: TAGATGTCTTTA

Figure 5.5: Strategy for designing primer candidates.

5.1.4.3 Applications

Primer design. Primers are short sequences of nucleotides which are designed such that the primer hybridizes to a given DNA sequence (or, in our case, to all of a given set of DNA sequences) in order to provide a start point for DNA strand synthesis by PCR (polymerase chain reaction). The hybridization of primers depends on complex thermodynamic rules, but is largely determined by the number of "mismatching" positions which should be as small as possible. Designing candidates for primers is a task often done by biological experts using the output of multiple alignment programs which is evaluated by hand. This task can be automatized in the same way as proposed by Stojanovic *et al.* [189] who used it for searching regulatory elements. Here, we also propose it for searching primers of length L which are required to bind to each of a set of homologous sequences with at most d mismatches. This strategy, depicted in Fig. 5.5, is outlined as follows.

1. Compute an alignment of these sequences [15].

2. "Slide" a length L window over all the aligned strings, solving a Closest String instance for every window position.

Using our algorithm for d-Mismatch, (2) can be done in $O(kL + (n-L)kd \cdot d^d)$ time (i.e., linear time for fixed d) where n is the length of the alignment.

On the one hand, determining the quality of a primer only based on the number of mismatching positions is a crude simplification of biological complexity. On the other hand, using our algorithm, we can also incorporate additional properties that are desirable for primers. We can adjust the search to find primers without mismatches in initial positions or by preferring solutions with long common substrings as proposed in [167]. As additional filtering, we can select solutions that are favorable in terms of melting temperature.

In an example experiment, we were faced with a slightly more general setting than described above. Given homologous sequences of parasite DNA as well as host DNA, the goal was to design primers that exclusively bind to the parasite sequences but not to the host sequences. The given data provided by Michael Weiss (group for Systematic Biology and Mycology, Tübingen) in this example were an alignment of length 715 with five sequences of parasite DNA and four sequences of host DNA. The parasite DNA was taken from some *Basidiomycetes* species which grow parasitically on plants, e.g., *Christiansenia pallida* and *Tremella exigua*. We approached the problem by solving a combination of d-MISMATCH and DSS on the parasite sequences as set of good strings and host sequences as set of bad strings: We slided a length-L window over all aligned strings in the same way as was explained for d-MISMATCH in Subsection 5.1.2.3, solving for every window position a DSS instance. Since we were given an alignment of the sequences, it was sufficient to solve DISTINGUISHING STRING SELECTION instances instead of a DISTINGUISHING SUBSTRING SELECTION instance. The desired length L of primers was between 15 to 20. Since the primers should have as few mismatches as possible, we considered here only $d_1 \leq 3$. E.g., with $L = 15$, $d_1 = 2$, the minimum value for which we found a primer candidate was $d_2 = 7$. For $L = 25$, we found a candidate with $d_1 = 2$ and $d_2 = 18$, or with $d_1 = 3$ and $d_2 = 15$. The advantage of the algorithm in this application is that it quickly (all runs were done in less than a second) finds all positions where primers are possible (and also finds if certain values of L, d_1 and d_2 do no allow a solution), whereas the human expert needs a lot of time and may find only obvious candidates.

Motif search. Another application of CLOSEST STRING algorithms can be found in motif search (for more details and references also refer to Section 5.2). A motif is a string that occurs approximately preserved, i.e., with changes in at most d positions for a fixed integer d, as a substring in several DNA sequences. Motifs are candidates for substrings of non-coding parts of the DNA sequence that have functions related to, e.g., gene expression. A formal definition of the motif search problem leads to the CLOSEST SUBSTRING problem. This problem has recently been addressed by a number of heuristic approaches, e.g. [35, 165]. Algorithms as [35] or [165] compute a candidate set of substrings, every of these substrings in a different given string, without actually checking whether this candidate set is (or can be extended to) a set of substrings, one in each given string, that satisfies the CLOSEST STRING property. Here, the CLOSEST STRING algorithms can be used to perform this check. The computation of the candidate set is, as a rule, much more time consuming than the CLOSEST STRING check and, therefore, the additional time needed by the CLOSEST STRING algorithm for practically relevant values like $L < 50$ and small values of d can easily be tolerated [85].

5.2 Part II: Motif Search Problems

Motif search problems are of central importance for sequence analysis in computational molecular biology. These problems have applications in fields such as genetic drug target identification or signal finding (see [35, 124, 128, 129, 165] and the references cited therein for more details and further applications). Two core problems in this context are CLOSEST SUBSTRING [129] (we recall here its definition already given in the introduction of this chapter) and CONSENSUS PATTERNS [128]:

> **Input:** Strings $s_1, s_2, \ldots, s_k$ over alphabet Σ and non-negative integers d and L.
>
> **Question in case of** CLOSEST SUBSTRING**:** Is there a string s of length L, and for every $i = 1, \ldots, k$, a length-L substring s'_i of s_i such that, for all $i = 1, \ldots, k$, $d_H(s, s'_i) \leq d$?
>
> **Question in case of** CONSENSUS PATTERNS**:** Is there a string s of length L, and, for every $i = 1, \ldots, k$, a length-L substring s'_i of s_i such that $\sum_{i=1}^{k} d_H(s, s'_i) \leq d$?

What is known regarding CLOSEST SUBSTRING has already been discussed in the introduction of this chapter, results regarding CONSENSUS PATTERNS are as follows:

1. CONSENSUS PATTERNS is NP-complete and remains so for the restriction to a binary alphabet [128].
2. CONSENSUS PATTERNS admits a PTAS [128], where the objective function is to minimize d, i.e., the maximum Hamming distance of the string s to one of the given strings.
3. The known PTAS for CONSENSUS PATTERNS is not an EPTAS, and whether an EPTAS is possible currently remains open.

Algorithms applied in practice to solve motif search problems closely related to CLOSEST SUBSTRING and CONSENSUS PATTERNS use heuristics [35, 165] or enumerative approaches [172, 27].

The key distinguishing point between CLOSEST SUBSTRING and CONSENSUS PATTERNS lies in the definition of the distance measure d between the solution string s and the substrings of the k input strings. Whereas CLOSEST SUBSTRING uses a "maximum distance" metric, CONSENSUS PATTERNS uses the "sum of distances" metric. This is of particular importance when discussing values of parameter d occurring in practice. Whereas it makes good sense for many applications to assume that d is a fairly small number in case of Closest Substring, this is much less reasonable in the case of Consensus Patterns. This will be of some importance when discussing our result for Consensus Patterns.

Here, we investigate the parameterized complexity of Closest Substring and Consensus Patterns. Unfortunately, our main results are negative ones: we show that Closest Substring and Consensus Patterns are W[1]-hard with respect to the number k of input strings, even in case of a binary alphabet.

1. For unbounded alphabet size, we show that Closest Substring is W[1]-hard for the combined parameters L, d, and k (Subsection 5.2.2).
2. For binary alphabet size, we show that Closest Substring is W[1]-hard for the combined parameters L, d, and k (Subsection 5.2.3).
3. We show that these results can be extended to Consensus Patterns (Subsection 5.2.4).

In the case of constant alphabet size, the parameterized complexity of the problems remains open when parameterized by d and k together, or by d alone. Note that in the case of Consensus Patterns our hardness result gains particular importance because here the distance parameter d usually is not small, whereas assuming that k is small makes sense. Until now, it was known only that if one additionally considers the substring length L as a parameter, then running times exponential in L can be achieved [27, 67, 172]. An overview on the old and new parameterized complexity results for Closest Substring and Consensus Patterns is given in Table 5.2.[8]

We achieve our results by giving parameterized many-one reductions from the W[1]-complete Clique problem to the respective problems. It is important here to note that parameterized reductions are much more fine-grained than conventional polynomial-time reductions used in NP-completeness proofs since parameterized reductions have to take care of the parameters. Establishing that Closest Substring and Consensus Patterns are W[1]-hard with respect to the parameter k requires significantly more technical effort than the already known demonstrations of NP-completeness [73].

Notably, based on the constructions presented in this section, the slightly more general Distinguishing Substring Selection problem [56, 124] (already mentioned in Subsection 5.1.2.3) was shown to be W[1]-hard also with respect to the distance parameters [83]. In particular, this implies that the recently presented PTAS for Distinguishing Substring Selection [56] cannot be improved into an EPTAS unless FPT =W[1] (see [83] and [41, 69] for details).

[8] Note that for unbounded alphabet size similar results were independently obtained by Evans *et al.* [66].

parameter	constant size alphabet	unbounded alphabet
d	(open)	W[1]-hard$^{(*)}$
k	W[1]-hard$^{(*)}$	W[1]-hard$^{(*)}$
d, k	(open)	W[1]-hard$^{(*)}$
L	FPT	W[1]-hard$^{(*)}$
d, k, L	FPT	W[1]-hard$^{(*)}$

Table 5.2: Overview on the parameterized complexity of Closest Substring and Consensus Patterns with respect to different parameterizations, where k is the number of given strings, L is the length of the substrings we search for, and d is the Hamming distance allowed. Results from this work are marked by $(*)$. The FPT results for constant size alphabet can be achieved by enumerating all length L strings over Σ.

Due to this result, it is to be expected that our constructions might be useful in further hardness proofs concerning string problems.

The remainder of this section is organized as follows. In Subsection 5.2.1, we give a brief overview on related computational biology results. Afterwards, in Subsection 5.2.2, we present a parameterized reduction of Clique to Closest Substring in case of unbounded input alphabet size. Then, in Subsection 5.2.3, this is specialized to the case of binary input alphabet. Finally, Subsection 5.2.4 gives similar constructions and results for Consensus Patterns.

5.2.1 Motivation and Previous Results

Applications for the consensus word analysis of DNA, RNA, or protein sequences include locating binding sites and finding conserved regions in unaligned sequences for genetic drug target identification, for designing genetic probes, and for universal PCR primer design. These problems can be regarded as various generalizations of the common substring problem, allowing errors (see [124, 128, 129] and references there). This leads to Closest Substring and Consensus Patterns, where errors are modeled by the (Hamming) distance parameter d.

There is a straightforward factor-2 approximation algorithm for Closest Substring, sketched as follows: With a generalization of Lemma 5.1.5 we conclude that, if a solution for the given instance exists, then there is a length-L substring s_i' of input string s_i for all $i = 1, \ldots, k$ such that the pairwise Hamming distance between every two substrings is at most $2d$. Therefore, we can test, for every length-L substring s_1' of the first input string, whether we find such a length-L substring s_i' of input string s_i for all $i = 1, \ldots, k$. If this holds for one substring s_1' of s_1 then s_1' yields a factor-2 approximation. If, otherwise, we cannot find such a substring s_1' then the input instance cannot have a solution.

The first better-than-2 approximation with factor $2-2/(2|\Sigma|+1)$ was given by Li *et al.* [127]. As mentioned at the begin of this section, there are PTAS's for Consensus Patterns [128] as well as for Closest Substring [129].

Concerning exact (parameterized) algorithms, we only briefly mention that, e.g., Sagot [172] studies motif discovery by solving Closest Substring, Evans and Wareham [67] give FPT algorithms for the same problem, and Blanchette *et al.* [27] developed a so-called phylogenetic footprinting method for a slightly more general version of Consensus Patterns. All these results, however, make essential use of the parameter "substring length" L and the running times show exponential behavior with respect to L. To circumvent the computational limitations for larger values of L, many heuristics were proposed, e.g., Pevzner and Sze [165] present algorithms called WINNOWER (with respect to Closest Substring) and SP-STAR (with respect to Consensus Patterns), and Buhler and Tompa [35] use random projections to find closest substrings. Our analysis makes a first step towards showing that, for exact solutions, we have to include L in the exponential growth; namely, we show that it is highly unlikely to find algorithms with a running time exponential *only* in k.

5.2.2 Closest Substring: Unbounded Alphabet

We first describe a reduction from the W[1]-hard Clique problem to Closest Substring which is a parameterized m-reduction with respect to the aggregate parameter (L, d, k) in case of unbounded alphabet size.

5.2.2.1 Reduction from Clique to Closest Substring

A Clique instance is given by an undirected graph $G = (V, E)$, with a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices, a set E of m edges, and a positive integer k denoting the desired clique size. We describe how to generate a set S of $\binom{k}{2}$ strings such that G has a clique of size k iff there is a string s of length $L := k+1$ such that every $s_i \in S$ has a substring s_i' of length L with $d_H(s, s_i') \leq d := k-2$. If a string $s_i \in S$ has a substring s_i' of length L with $d_H(s, s_i') \leq d$, we call s_i' a *match*. We assume $k > 2$ because $k = 1, 2$ are trivial cases.

Alphabet. The alphabet of the produced instance is given by the disjoint union of the following sets:

- $\{\, \sigma_i \mid v_i \in V\,\}$, i.e., an alphabet symbol for every vertex of the input graph; we call them *encoding symbols*;
- $\{\, \varphi_j \mid j = 1, \ldots, \binom{k}{2}\,\}$, i.e., a unique symbol for every of the $\binom{k}{2}$ produced strings; we call them *string identification symbols*;
- $\{\#\}$ which we call the *synchronizing symbol*.

This makes a total of $n + \binom{k}{2} + 1$ alphabet symbols.

Choice strings. We generate a set of $\binom{k}{2}$ *choice strings* $S_c = \{c_{1,2}, \dots, c_{1,k}, c_{2,3}, c_{2,4}, \dots, c_{k-1,k}\}$ and we assume that the strings in S_c are ordered as shown. *Every* choice string will encode the whole graph; it consists of m concatenated strings, each of length $k+1$, called *blocks*; by this, we have one block for every edge of the graph. The blocks will be separated by *barriers*, which are length k strings consisting of k identification symbols corresponding to the respective string. A choice string $c_{i,j}$, which, according to the given order, is the i'th choice string in S_c, is given by

$$c_{i,j} := \langle \text{block}(i, j, e_1) \rangle \, (\varphi_{i'})^k \, \langle \text{block}(i, j, e_2) \rangle \, (\varphi_{i'})^k \dots (\varphi_{i'})^k \, \langle \text{block}(i, j, e_m) \rangle,$$

where $e_1, e_2, \dots, e_m$ are the edges of G and $\langle \text{block}() \rangle$ will be defined below. The solution string s will have length $k+1$, which is exactly the length of one block.

Block in a choice string. Every block is a string of length $k+1$ and it encodes an edge of the input graph. Every choice string contains a block for every edge of the input graph; different choice strings, however, encode the edges in different positions of their blocks: For a block in choice string $c_{i,j}$, positions i and j are called *active* and these positions encode the edge. Let e be the edge to be encoded and let e connect vertices v_r and v_s, $1 \le r < s \le n$. Then, the ith position of the block is σ_r in order to encode v_r and the jth position is σ_s in order to encode v_s. The last position of a block is set to the synchronizing symbol #. Let $c_{i,j}$ be the i'th choice string in S_c; then, all remaining positions in the block are set to $c_{i,j}$'s identification symbol $\varphi_{i'}$. Thus, the block is given by

$$\langle \text{block}(i, j, (v_r, v_s)) \rangle \; := \; (\varphi_{i'})^{i-1} \, \sigma_r \, (\varphi_{i'})^{j-i-1} \, \sigma_s \, (\varphi_{i'})^{k-j} \, \#.$$

Values for L and d. We set $L := k+1$ and $d := k-2$.

Example 5.2.1. Let $G = (V, E)$ be an undirected graph with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$ (as shown in Fig. 5.6(a)) and let $k = 3$. Using G, we exhibit the above construction of $\binom{k}{2} = 3$ choice strings c_1, c_2, and c_3 (as shown in Fig. 5.6(b)). Note that, in the described construction, the strings were called $c_{1,2}$, $c_{1,3}$, and $c_{2,3}$ but, here, for the ease of presentation, we call them c_1, c_2, and c_3. We claim that (which will be proven in the following subsection) there exists a clique of size k in G iff there is a string s of length $L := \binom{k}{2} + 1 = 4$ such that, for $i = 1, 2, 3$, each c_i contains a length 4 substring s_i with $d_H(c_i, s_i) \le d := k - 2 = 1$.

The choice strings are over an alphabet consisting of $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ (the encoding symbols, i.e., one symbol for every node of G), $\{\varphi_1, \varphi_2, \varphi_3\}$ (the string identification symbols), and $\{\#\}$ (the synchronizing symbol). Every string c_i, $i = 1, 2, 3$ consists of four blocks, each of which encodes an edge of the graph. Every block is of length $\binom{k}{2} + 1 = 4$ and has # at its last position. The blocks are separated by barriers consisting of $(\varphi_i)^k = (\varphi_i)^3$.

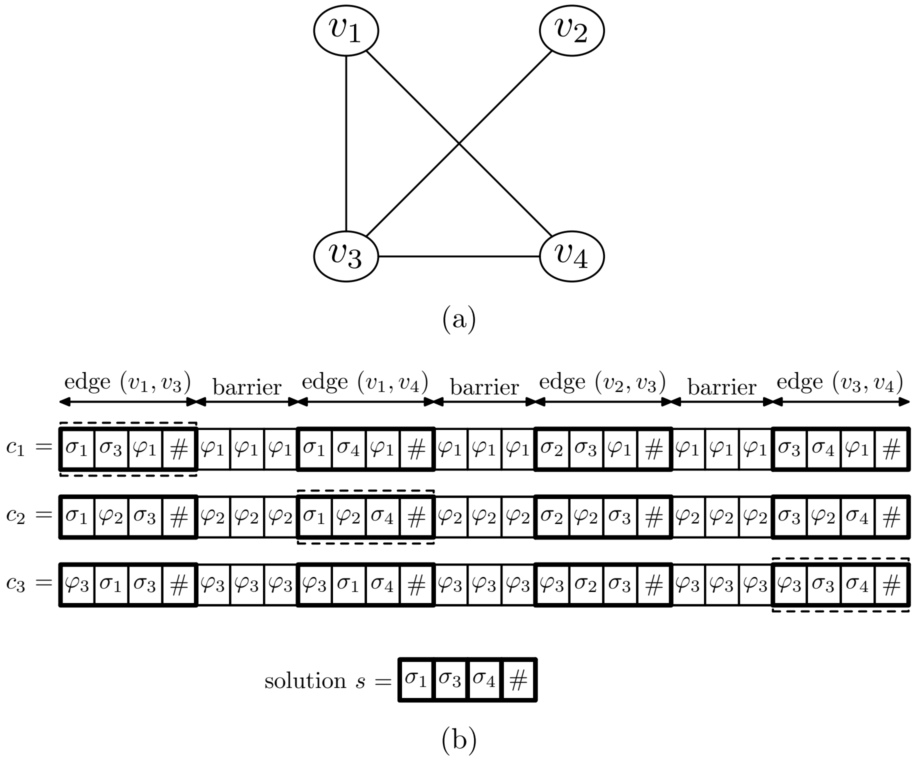

v_1
v_2
v_3
v_4
(a)
edge (v_1, v_3)
barrier
edge (v_1, v_4)
barrier
edge (v_2, v_3)
barrier
edge (v_3, v_4)
$c_1 =$ $\sigma_1\sigma_3\varphi_1\#\ \varphi_1\varphi_1\varphi_1\ \sigma_1\sigma_4\varphi_1\#\ \varphi_1\varphi_1\varphi_1\ \sigma_2\sigma_3\varphi_1\#\ \varphi_1\varphi_1\varphi_1\ \sigma_3\sigma_4\varphi_1\#$
$c_2 =$ $\sigma_1\varphi_2\sigma_3\#\ \varphi_2\varphi_2\varphi_2\ \sigma_1\varphi_2\sigma_4\#\ \varphi_2\varphi_2\varphi_2\ \sigma_2\varphi_2\sigma_3\#\ \varphi_2\varphi_2\varphi_2\ \sigma_3\varphi_2\sigma_4\#$
$c_3 =$ $\varphi_3\sigma_1\sigma_3\#\ \varphi_3\varphi_3\varphi_3\ \varphi_3\sigma_1\sigma_4\#\ \varphi_3\varphi_3\varphi_3\ \varphi_3\sigma_2\sigma_3\#\ \varphi_3\varphi_3\varphi_3\ \varphi_3\sigma_3\sigma_4\#$
solution $s =$ $\sigma_1\sigma_3\sigma_4\#$
(b)

Figure 5.6: Example for the reduction from a Clique instance G with $k = 3$ (shown in (a)) to a Closest Substring instance with bounded alphabet (shown in (b)) as explained in Example 5.2.1. In (b), we display the constructed strings c_1, c_2, and c_3 (the contained blocks are highlighted by bold boxes) and the solution string s that is found since G has a clique of size $k = 3$; s is a string of length $k + 1 = 4$ such that c_1, c_2, and c_3 have length 4 substrings (indicated by dashed boxes) that have Hamming distance at most $k - 2 = 1$ to s.

In string c_1, positions 1 and 2 within a block are active and encode the corresponding edge (in c_2 positions 1 and 3, and, in c_3 positions 2 and 3 within a block are active). All of the first k positions of a block in string c_i, $i = 1, 2, 3$ which are not active, contain the φ_i symbol. Thus, e.g., the block in c_1 encoding the edge (v_1, v_3) is given by $\sigma_1\sigma_3\varphi_1\#$. Further details can be found in Fig. 5.6.

The closest substring that corresponds to the k-clique in G consisting of vertices v_1, v_3, and v_4 is $\sigma_1\sigma_3\sigma_4\#$. The corresponding matches are $\sigma_1\sigma_3\varphi_1\#$ in c_1 (encoding the edge (v_1, v_3)), $\sigma_1\varphi_2\sigma_4\#$ in c_2 (encoding the edge (v_1, v_4)), and $\varphi_3\sigma_3\sigma_4\#$ in c_3 (encoding the edge (v_3, v_4)).

5.2.2.2 Correctness of the Reduction

To prove the correctness of the proposed reduction, we have to show an equivalence, consisting of two directions. The easier one is to see that a k-clique implies a closest substring fulfilling the given requirements.

Proposition 5.2.2. *For a graph with a k-clique, the construction in Subsection 5.2.2.1 produces an instance of* Closest Substring *which has a solution, i.e., there is a string s of length L such that every $c_{i,j} \in S_c$ has a substring $s_{i,j}$ with $d_H(s, s_{i,j}) \leq d$.*

Proof. Let the input graph have a clique of size k. Let $h_1, h_2, \ldots, h_k$ denote the indices of the clique's vertices, $1 \leq h_1 < h_2 < \ldots < h_k \leq n$. Then, we claim that a solution for the produced Closest Substring instance is

$$s := \sigma_{h_1} \sigma_{h_2} \ldots \sigma_{h_k} \#.$$

Consider choice string $c_{i,j}$, $1 \leq i < j \leq k$. As the vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$ form a clique, we have an edge connecting v_{h_i} and v_{h_j}. Choice string $c_{i,j}$ contains a block $s_{i,j} := \langle \text{block}(i, j, (v_{h_i}, v_{h_j})) \rangle$ encoding this edge:

$$s_{i,j} \; := (\varphi_{i'})^{i-1} \, \sigma_{h_i} \, (\varphi_{i'})^{j-i-1} \, \sigma_{h_j} \, (\varphi_{i'})^{k-j} \#,$$

where i' is the number (according to the given order) of the choice string in S_c. We have $d_H(s, s_{i,j}) = k-2$, and we can find such a block for every $c_{i,j}$, $1 \leq i < j \leq k$. □

For the reverse direction, we show in Proposition 5.2.5 that a solution in the produced Closest Substring instance implies a k-clique in the input graph. For this, we need the following two lemmas, which show that a solution to the instance constructed in Subsection 5.2.2.1 has encoding symbols at its first k positions and the synchronizing symbol # at its last position.

Lemma 5.2.3. *A closest substring s contains at least two encoding symbols and at least one synchronization symbol.*

Proof. Let s be a solution of the Closest Substring instance produced by the construction in Subsection 5.2.2.1. Let $A_\varphi(s)$ be the set of string identification symbols from $\{ \varphi_i \mid 1 \leq i \leq \binom{k}{2} \}$ that occur in s. Let $S_\varphi(s) \subseteq S_c$ be the subset of choice strings that do *not* contain a symbol from $A_\varphi(s)$.

Since s is of length $k+1$, we have $|A_\varphi(s)| \leq k+1$. Therefore, for $k \geq 4$, there are at least $\binom{k}{2} - (k+1)$ choice strings in $S_\varphi(s)$. We show that with less than two encoding symbols and no synchronizing symbol, we cannot find matches for s (with maximally allowed Hamming distance $d = k-2$) in the choice strings of $S_\varphi(s)$. Observe that, in every choice string, because of the barriers, every length $k+1$ substring contains at most two encoding symbols and at most one symbol #. Observe further that, taken a choice string from $S_\varphi(s)$, positions with symbols from $\{ \varphi_i \mid 1 \le i \le \binom{k}{2} \}$ cannot coincide with the corresponding positions in s. Therefore, s has a match in such a string only if s has two encoding symbols and one symbol # that all coincide with the corresponding positions in the selected substring. This proves the claim for $k \ge 4$. Regarding $k = 3$, if $|A_\varphi(s)| < 3$, then the above argument applies here, too. If, however, $|A_\varphi(s)| = 3$, a length 4 substring in every choice string has at least two positions that do not coincide with the corresponding positions in s. □

Based on Lemma 5.2.3, we can now exactly specify the numbers and positions of the encoding and synchronizing symbols in the closest substring.

Lemma 5.2.4. *A closest substring* s *contains encoding symbols at its first* k *positions and a symbol # at its last position.*

Proof. Let $n_\#(s)$ denote the number of symbols # in s, let $n_\varphi(s)$ denote the number of string identification symbols in s, and let $n_\sigma(s)$ denote the number of encoding symbols in s. Let $S_\varphi(s) \subseteq S_c$ be the subset of choice strings whose string identification symbol does not occur in s. In the following, we establish a lower bound on the number of strings in $S_\varphi(s)$ and an upper bound on the number of strings from $S_\varphi(s)$ in which we can find a match for s. Comparing these bounds, we will show that, if $n_\#(s) > 1$, then there are choice strings in $S_\varphi(s)$ in which we cannot find a match; we will conclude that $n_\#(s) = 1$. Then, we will show that, if $n_\sigma(s) < k$, then again there are strings in $S_\varphi(s)$ without a match; we will conclude that $n_\sigma(s) = k$.

Regarding the size of $S_\varphi(s)$, a lower bound on its size is $|S_\varphi(s)| \ge \binom{k}{2} - n_\varphi(s)$. To explain the upper bound on the number of strings from $S_\varphi(s)$ in which we can find a match for s, we recall that such matches must contain two encoding symbols and one symbol # that all coincide with the corresponding positions in s. On the one hand, the synchronizing symbol of a block must coincide with a symbol # in s. On the other hand, in all blocks of a choice string, its encoding symbols are in fixed positions relative to the block's synchronizing symbol, e.g., in choice string $c_{1,2}$, the encoding symbols are located only at the first and second position and # at the last position of a block in $c_{1,2}$. For these two reasons, one symbol # in s can provide matches in at most $\binom{n_\sigma(s)}{2}$ choice strings from $S_\varphi(s)$. Consequently, $n_\#(s)$ many symbols # in s can provide matches in at most $n_\#(s) \cdot \binom{n_\sigma(s)}{2}$ choice strings from $S_\varphi(s)$.

Summarizing, we have at least $\binom{k}{2} - n_\varphi(s)$ choice strings in $S_\varphi(s)$ and we can find matches in at most $n_\#(s) \cdot \binom{n_\sigma(s)}{2}$ many of them. Thus, we find matches for s in all choice strings only if

$$n_\#(s) \cdot \binom{n_\sigma(s)}{2} \ge \binom{k}{2} - n_\varphi(s). \tag{5.1}$$

In order to show that s contains exactly one synchronizing symbol, we assume that $n_\#(s) > 1$ (we know that $n_\varphi(s) \geq 1$ by Lemma 5.2.3) while $k > 2$, and show that inequality 5.1 is violated.

We know that $k + 1 = n_\sigma(s) + n_\varphi(s) + n_\#(s)$ and, by Lemma 5.2.3, that $n_\sigma(s) \geq 2$. Using these, we conclude, on the one hand, that $n_\#(s) \cdot \binom{n_\sigma(s)}{2} \leq n_\#(s) \cdot \binom{k+1-n_\#(s)}{2}$ and, since $n_\#(s) \geq 2$, that $n_\#(s) \cdot \binom{k+1-n_\#(s)}{2} \leq 2 \cdot \binom{k-1}{2}$. On the other hand, we have that $\binom{k}{2} - n_\varphi(s) \geq \binom{k}{2} - (k-1-n_\#(s))$ and, since $n_\#(s) \geq 2$, $\binom{k}{2} - (k-1-n_\#(s)) \geq \binom{k}{2} - (k-3)$. For $k \geq 3$, however we have $\binom{k}{2} - (k-3) > 2 \cdot \binom{k-1}{2}$. Thus,

$$\begin{array}{rcl} n_\#(s) \cdot \binom{n_\sigma(s)}{2} & \leq & n_\#(s) \cdot \binom{k+1-n_\#(s)}{2} \\ & < & \binom{k}{2} - (k-1-n_\#(s)) \quad \leq \quad \binom{k}{2} - n_\varphi(s), \end{array}$$

i.e., there are choice strings in $S_\varphi(s)$ which contain no match for s, a contradiction. Since (Lemma 5.2.3) $n_\#(s) \geq 1$, we conclude that $n_\#(s) = 1$.

In order to show that s contains exactly k encoding symbols, we assume that $n_\sigma(s) < k$ while $k > 2$ and $n_\#(s) = 1$, and show that inequality 5.1 is violated. Since $k+1 = n_\sigma(s)+n_\varphi(s)+n_\#(s) = n_\sigma(s)+n_\varphi(s)+1$, we have $\binom{k}{2} - n_\varphi(s) = \binom{k}{2} - (k - n_\sigma(s))$ and, thus,

$$\binom{n_\sigma(s)}{2} < \binom{k}{2} - (k - n_\sigma(s)) \leq \binom{k}{2} - n_\varphi(s),$$

i.e., again, some strings in $S_\varphi(s)$ have no match for s, a contradiction. Thus, on the one hand, we have $n_\sigma(s) \geq k$, and, on the other hand, we have $n_\#(s) = 1$ and, therefore, $n_\sigma(s) \leq k$.

Note that, if an encoding symbol is located *after* the synchronizing symbol in s, then, due to the barriers, it is not possible that both # and this encoding symbol coincide with the respective positions in a choice string from $S_\varphi(s)$. Therefore, symbol # is located at the last position of s. □

Proposition 5.2.5. *The first* k *characters of a closest substring correspond to* k *vertices of a clique in the input graph.*

Proof. By Lemma 5.2.4, a closest substring s has encoding symbols at its first k positions and a synchronizing symbol at its last position. Consequently, the blocks are the only possible matches of s in the choice string. Now, assume that $s = \sigma_{h_1}\sigma_{h_2}\ldots\sigma_{h_k}\#$ for $h_1, h_2, \ldots, h_k \in \{1, \ldots, n\}$. Consider any two h_i, h_j, $1 \leq i < j \leq k$, and choice string $c_{i,j}$. Recall that in this choice string, the blocks encode edges at their ith and jth position, they have # at their last position, and all their other positions are set to a string identification symbol unique for this choice string. Thus, we can only find a block that is a match if there is a block with σ_{h_i} at its ith position and σ_{h_j} at its jth position. We have such a block only if there is an edge connecting v_{h_i} and v_{h_j}. Summarizing, the closest substring s implies that there is an edge between every pair of $\{v_{h_1}, v_{h_2}, \ldots, v_{h_k}\}$; these vertices form a k-clique in the input graph. □

Propositions 5.2.2 and 5.2.5 establish the following hardness result. Note that hardness for the combination of all three parameters also implies hardness for each subset of the three.

Theorem 5.2.6. CLOSEST SUBSTRING *with unbounded alphabet is* W[1]*-hard for every combination of the parameters* L, d, *and* k. □

5.2.3 Closest Substring: Binary Alphabet

We modify the reduction from Subsection 5.2.2 to achieve a CLOSEST SUBSTRING instance with binary alphabet proving a W[1]-hardness result also in this case. In contrast to the previous construction, we cannot encode every vertex with its own symbol and we cannot use a unique symbol for every produced string. Also, we have to find new ways to "synchronize" the matches of our solution, a task previously done by the synchronizing symbol #. To overcome these problems, we construct an additional "complement string" for the input instance and we lengthen the blocks in the produced choice strings considerably.

5.2.3.1 Reduction from Clique to Closest Substring

Number strings. To encode integers between 1 and n, we introduce *number strings* ⟨number(*pos*)⟩, which have length n and which have symbol "1" at position *pos* and symbol "0" elsewhere: $0^{pos-1}\,1\,0^{n-pos}$. In contrast to the reduction from Subsection 5.2.2, now we use these number strings to encode the vertices of a graph.

Choice strings. As in Subsection 5.2.2, we generate a set of $\binom{k}{2}$ *choice strings* $S_c = \{c_{1,2}, c_{1,3} \ldots, c_{k-1,k}\}$. Again, every choice string will consist of m *blocks*, one block for every edge of the graph. The choice string $c_{i,j}$ is given by

$$c_{i,j} \;:=\; \langle\text{block}(i,j,e_1)\rangle\langle\text{block}(i,j,e_2)\rangle \ldots \langle\text{block}(i,j,e_m)\rangle,$$

where $e_1, e_2, \ldots, e_m$ are the edges of the input graph and ⟨block()⟩ is defined below. The length of a closest substring will be exactly the length of one block.

Block in a choice string. Every block consists of a front tag, an encoding part, and a back tag. A block in choice string $c_{i,j}$ encodes an edge e; let e be an edge connecting vertices v_r and v_s, $1 \leq r < s \leq n$, and let $c_{i,j}$ be the (according to the given order) i'th string in S_c. Then, the corresponding block is given by

$$\langle\text{block}(i,j,(v_r,v_s))\rangle \;:=\langle\text{front\_tag}\rangle\langle\text{encode}(i,j,(v_r,v_s))\rangle\langle\text{back\_tag}(i')\rangle.$$

Front tags. We want to enforce that a closest substring can only match substrings at certain positions in the produced choice strings, using front tags:

$$\langle\text{front\_tag}\rangle \;:=\; (1^{3nk}0)^{nk},$$

i.e., a front tag has length $(3nk + 1) \cdot nk$. By this arrangement, the closest substring s and every match of s start (as will be shown in Subsection 5.2.3.2) with the front tag.

Encoding part. The encoding part consists of k sections, each of length n. The encoding part corresponds to the blocks used in Subsection 5.2.2. As a consequence, in $\langle\text{block}(i, j, e)\rangle$ the ith and jth section are called *active* and encode edge $e = (v_r, v_s)$, $1 \leq r < s \leq n$; section i encodes v_r by $\langle\text{number}(r)\rangle$ and section j encodes v_s by $\langle\text{number}(s)\rangle$. The other sections except for i and j are called *inactive* and are given by $\langle\text{inactive}\rangle := 0^n$. Thus,

$$\begin{aligned}&\langle\text{encode}(i, j, (v_r, v_s))\rangle := \\ &\qquad (\langle\text{inactive}\rangle)^{i-1}\,\langle\text{number}(r)\rangle\,(\langle\text{inactive}\rangle)^{j-i-1}\,\langle\text{number}(s)\rangle\,(\langle\text{inactive}\rangle)^{k-j}.\end{aligned}$$

Back tag. The back tag of a block is intended to balance the Hamming distance of the closest substring to a block, as will be explained later. The back tag consists of $\binom{k}{2}$ sections, each section has length $nk-2k+2$. The i'th section consists of symbols "1," all other sections consist of symbols "0":

$$\langle\text{back\_tag}(i')\rangle \;:=\; 0^{(i'-1)(nk-2k+2)}1^{nk-2k+2}0^{(\binom{k}{2}-i')(nk-2k+2)}$$

Template string. The set of choice strings is complemented by one *template string.* It consists, in analogy to the blocks in the choice strings, of three parts: A front tag of length $(3nk + 1) \cdot nk$, followed by a length nk string of symbols "1," followed by a length $\binom{k}{2}(nk - 2k + 2)$ string of symbols "0." Thus, the template string has the same length as a block in a choice string, i.e., $(3nk + 1) \cdot nk + nk + \binom{k}{2}(nk - 2k + 2)$.

Values for d and L. We set $L := (3nk + 1) \cdot nk + nk + \binom{k}{2}(nk - 2k + 2)$ and $d := nk - k$. As we will show in Subsection 5.2.3.2, the possible matches for a string of this length are the blocks in the choice strings, and, concerning the template string, the template string itself.

Notation. For a closest substring s, we denote its first $(3nk + 1) \cdot nk$ symbols (the front tag) by s', the following nk symbols (its encoding part) by s'', and the last $\binom{k}{2}(nk - 2k + 2)$ symbols (its back tag), by s'''. Analogously, the three parts of the template string t are denoted t', t'', and t'''. A particular block of a choice string $c_{i,j}$, is referred to by $s_{i,j}$; its three parts are called $s'_{i,j}$, $s''_{i,j}$, and $s'''_{i,j}$.

Example 5.2.7. Let $G = (V, E)$ be the graph from Example 5.2.1, with vertices $V = \{v_1, v_2, v_3, v_4\}$ and edges $E = \{(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$ (as shown in Fig. 5.6(a)) and let $k = 3$. In the following, we outline the above construction of $\binom{k}{2} = 3$ choice strings c_1, c_2, and c_3 and one template string t over alphabet $\Sigma = \{0, 1\}$ as displayed in Fig. 5.7.

Every string c_1, c_2, and c_3 consists of four blocks corresponding to the four edges of G. Fig. 5.7(a) displays the first block of c_1 corresponding to edge (v_1, v_3). It

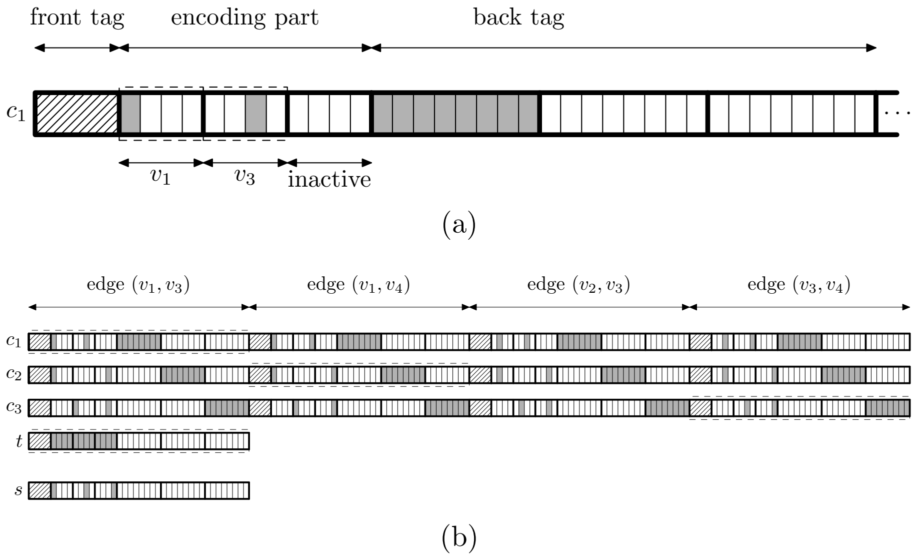


Figure 5.7: Example for the reduction from the Clique instance G (shown in Fig. 5.6(a)) to a Closest Substring instance with binary alphabet as explained in Example 5.2.7. When displaying the strings, we omit the details of the front tag parts and only indicate them shortened in their proportion to the other parts of the strings; all front tag parts in all strings are equal. In the encoding parts and the back tag parts, we indicate the symbols "1" of the construction by dark boxes, the symbols "0" by white boxes. In (a), we outline the first block of c_1. In its encoding part, sections 1 and 2 (sections are indicated by bold separating lines) are active (indicated by dashed boxes) and encode the first edge (v_1, v_3) of graph G; the remaining third section is inactive. In its back tag part, the first section is filled with symbols "1." In (b), we give an overview on all constructed strings, the choice strings c_1, c_2, and c_3, and the template string t. We also display the closest substring s that is found since G has a clique of size $k = 3$; its matches in c_1, c_2, c_3, and t are indicated by dashed boxes. A focus on the matches is given in Figure 5.8.

consists of a front tag, an encoding part, and a back tag. The front tag (not displayed in detail in the figure) is given by $\langle\text{front\_tag}\rangle := (1^{3nk}0)^{nk} = (1^{36}0)^{12}$; all front tags for all blocks in all constructed strings are the same. The back tag of the first block consists of $\binom{k}{2}$ sections; since the back tag is in the *first* string, the *first* section is filled with "1"s and the remaining sections are filled with "0"s. Thus, the back tag is given by $1^{nk-2k+2}0^{(\binom{k}{2}-1)(nk-2k+2)} = 1^8 0^{16}$, and all back tags for blocks in the first string are given like this. The encoding part consists of $k = 3$ sections, each section of length $n = 4$. In the blocks of string c_1, the first and the second section are active; in the first block they encode edge (v_1, v_3). Therefore, the first section is given by $\langle\text{number}(1)\rangle$ and the second one by $\langle\text{number}(3)\rangle$, the remaining inactive section is filled with "0"s.

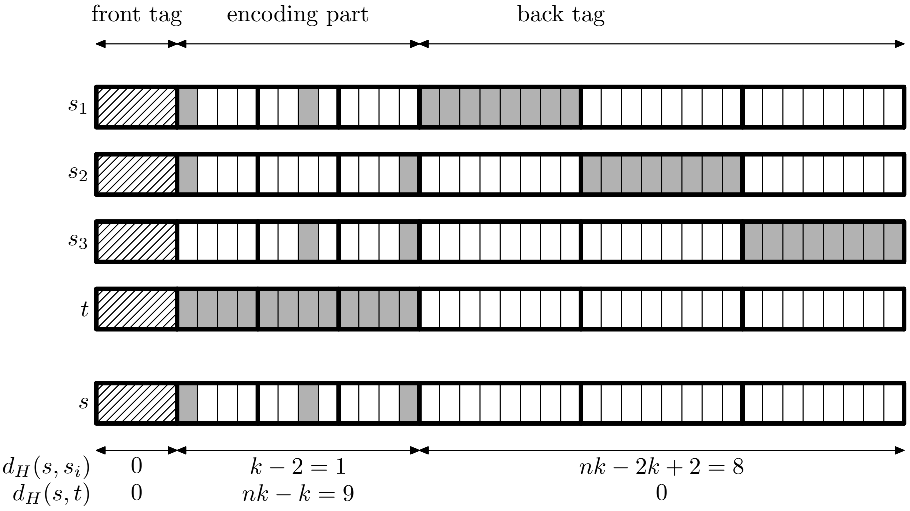


Figure 5.8: Continuation of Figure 5.7. We illustrate the reduction from the Clique instance G (shown in Fig. 5.6(a)) to a Closest Substring instance with binary alphabet as explained in Example 5.2.7. Here, we focus on the matches in the produced instance, which were marked by dashed boxes in Figure 5.7(b), together with the template string t and the solution string s. We state, separately for the front tag, the encoding, and the back tag part, the Hamming distances of s to a match s_i, $i = 1, 2, 3$ (the distances are equal for s_1, s_2, and s_3) and to the template string t.

Fig. 5.7(b) displays an overview on all constructed strings c_1, c_2, c_3, and t. In all strings, block i encodes the ith edge, $1 \leq i \leq 4$. However, the active sections of the encoding part and the back tags differ for different strings. The template string t consists only of one block, which has a front tag, a part corresponding to the encoding part, filled with "1"s, and a part corresponding to the back tag, filled with "0"s.

Since G has a k-clique for $k = 3$, consisting of vertices v_1, v_3, and v_4, we find a solution s for the constructed Closest Substring instance. This s has a front tag, and its back tag part is filled with "0" symbols. The encoding part encodes the vertices of the clique, it is given by $\langle\text{number}(1)\rangle\langle\text{number}(3)\rangle\langle\text{number}(4)\rangle$.

Fig. 5.7(h) gives a focus on the matches that are found in c_1, c_2, c_3, and t, which are, for the choice strings, referred to by s_1, s_2, and s_3, respectively. The front tag part s' has distance 0 to the front tags s'_1, s'_2, s'_3, and t'. The encoding part s'' contains $k = 3$ many "1"s; s''_1, s''_2, s''_3 have two "1"s each and, in each case, these "1"s coincide with "1"s in s''. Therefore, $d_H(s'', s_i) = k - 2 = 1$, $1 \leq i \leq 3$. The encoding part of the template string, t'', only consists of "1"s and, therefore, $d_H(s'', t'') = nk - k$. The back tag s''' only consists of "0"s; each back tag s'''_1, s'''_2, and s'''_3 contains $nk - 2k + 2 = 8$ many "1"s; therefore $d_H(s''', s'''_i) = 8$, $1 \leq i \leq 3$. The back tag of the template string, t''', contains only "0"s and, hence, $d_H(s''', t''') = 0$. Altogether, this shows that, for $1 \leq i \leq 3$, $d_H(s, s_i) = d_H(s, t) = nk - k = 9$ as required.

5.2.3.2 Correctness of the Reduction

To prove the correctness of the reduction, again the easier direction is to show that a k-clique implies a closest substring fulfilling the given requirements.

Proposition 5.2.8. *For a graph with a* k*-clique, the construction in Subsection 5.2.3.1 produces an instance of* Closest Substring *that has a solution, i.e., there is a string* s *of length* L *such that every* $c_{i,j} \in S_c$ *has a length* L *substring* $s_{i,j}$ *with* $d_H(s, s_{i,j}) \leq d$ *and* $d_H(s, t) \leq d$.

Proof. Let the graph have a clique of size k. Let $h_1, h_2, \ldots, h_k$ denote the indices of the clique's vertices, $1 \leq h_1 < h_2 < \ldots < h_k \leq n$. Then, we can find a closest substring s, consisting of three parts s', s'', and s''', as follows: its front tag s' is given by ⟨front_tag⟩; its encoding part s'' is given by ⟨number(h_1)⟩⟨number(h_2)⟩ ... ⟨number(h_k)⟩; its back tag s''' is $0^{\binom{k}{2}(nk-2k+2)}$. It follows from the construction that the choice strings have substrings that are matches for this s: For every $1 \leq i < j \leq k$, we produced choice string $c_{i,j}$ with a block $s_{i,j}$ encoding the edge between vertices v_{h_i} and v_{h_j}. For these blocks as well as for the template string, the following table reports the distance they have to the solution string, separately for each of their three parts and in total:

$d_H(\cdot, \cdot)$	s'	s''	s'''	s
match $s_{i,j}$ in choice string $c_{i,j}$	0	$k-2$	$nk-2k+2$	$nk-k$
template string t	0	$nk-k$	0	$nk-k$

As is obvious from these distance values, the indicated substrings in the choice strings all have Hamming distance $d = nk - k$ to the solution string and, therefore, are matches for s. □

For the reverse direction, we assume that the Closest Substring instance has a solution. We need the following statements:

Lemma 5.2.9. *A solution* s *and all its matches in the input instance start with the front tag.*

Proof. Since s is of length $L = (3nk+1) \cdot nk + nk + \binom{k}{2}(nk - 2k + 2)$, the only possible match in the template string is the template string itself. Therefore, s' can differ from t' in at most $d = nk - k$ symbols. We can show that the only substrings in a choice string $c_{i,j}$ that are possible matches for s with Hamming distance at most d start with the front tag, as we argue in the following.

Since s is a solution, there is a match in $c_{i,j}$ and we denote it by $s_{i,j}$. Denote the the first $(3nk+1) \cdot nk$ symbols of $s_{i,j}$ by $s'_{i,j}$. Since $d_H(s', s'_{i,j}) \leq nk - k$ and

$d_H(s', t') \leq nk - k$, we necessarily (triangle inequality for Hamming metric) have $d_H(s'_{i,j}, t') \leq 2(nk - k)$. We show that this is only possible when $s'_{i,j}$ coincides with a front tag of a block of $c_{i,j}$. Assuming that it does not, we will show that $d_H(s'_{i,j}, t') > 2(nk - k)$, a contradiction.

Firstly, assume that the starting position of $s'_{i,j}$ and the starting position of a front tag in $c_{i,j}$ differ by p positions, $1 \leq p \leq 3nk$. Then, at least $nk - 1$ symbols "0" of t' are aligned with symbols "1" of the front tag in $s'_{i,j}$ and at least $nk - 1$ symbols "1" of t' are aligned with symbols "0" of $s'_{i,j}$. This implies $d_H(s'_{i,j}, t') > 2nk - 2$. Secondly, assume that the starting position of $s'_{i,j}$ and the starting position of its closest front tag in $c_{i,j}$ differ by $p > 3nk$ positions. Then, a block of $3nk$ symbols "1" falls onto the encoding and/or the back tag part of $s'_{i,j}$. Since the encoding part and back tag contain together only $2 + (nk - 2k + 2) < nk$ (under the assumption that $k > 2$) many symbols "1", we have more than $2nk$ mismatching symbols and $d_H(s'_{i,j}, t') > 2(nk - k)$.

Summarizing, we conclude that $s'_{i,j}$ coincides with a front tag in choice string $c'_{i,j}$, i.e., $s'_{i,j} = t' = s' = \langle\text{front\_tag}\rangle$. □

Lemma 5.2.10. *The encoding part of* s *contains exactly* k *symbols "1".*

Proof. Assume that s has less than k symbols "1" in its encoding part, i.e., s'' contains less than k symbols "1". Then, because $t'' = 1^{nk}$, $d_H(s'', t'') \geq nk - k + 1$, implying $d_H(s, t) \geq nk - k + 1$, a contradiction.

Assume that s has more than k "1" symbols in its encoding part s''. Then, $d_H(s'', s''_{i,j}) > k - 2$ for the encoding part $s''_{i,j}$ of a match in every choice string $c_{i,j}$. Now consider the solution's back tag s'''. To achieve $d_H(s, s_{i,j}) \leq nk - k$, we need $d_H(s''', s'''_{i,j}) < nk - 2k + 2$ and s''' must contain one or more symbols "1". Every "1" symbol in s''' will decrease the value $d_H(s, s_{i,j})$ for a block $s_{i,j}$ of *one* choice string $c_{i,j}$ by one but will increase the solution's Hamming distance to the selected blocks of *all other* choice strings. No matter how many "1" symbols we have in the back tag of s, there will always be a choice string $c_{i,j}$ with $d_H(s''', s'''_{i,j}) \geq nk - 2k + 2$. In summary, we will always have a choice string $c_{i,j}$ with $d_H(s, s_{i,j}) = d_H(s'', s''_{i,j}) + d_H(s''', s'''_{i,j}) > nk - k$, a contradiction. □

Lemma 5.2.11. *Every section of the encoding part of* s *contains exactly one symbol "1".*

Proof. Assume that not every section in the encoding part of s contains exactly one "1" symbol. Then, there must be a section containing no symbol "1" since, by Lemma 5.2.10, the number of symbols "1" in the encoding part of s adds up to k. Let i', $1 \leq i' \leq k$, be the section containing no symbol "1". W.l.o.g., consider a choice string $c_{i',j}$, $i' < j \leq k$ or, if $i' = k$, a choice string $c_{i',j}$, $1 \leq j < i'$. In *every* block $s_{i',j}$ of $c_{i',j}$, sections i' and j of the encoding part are active and, therefore, contain exactly one symbol "1" each; these are the only symbols "1" in $s''_{i',j}$. Now consider the k symbols "1" in the encoding part of s: The "1"s in all sections of s'' except for section j are all aligned with "0"s in $s''_{i',j}$; within section j, only a single "1" of s'' can be matched to a "1" of $s''_{i',j}$. Therefore, $d_H(s'', s''_{i',j}) > k - 2$. As in the proof of Lemma 5.2.10, we conclude that s is no solution. □

Proposition 5.2.12. *The* k *symbols "1" in the solution string's encoding part correspond to a* k*-clique in the graph.*

Proof. Let s be a solution for the Closest Substring instance. Summarizing, we know by Lemma 5.2.9 that s can have as a match only one of the choice string's blocks. By Lemma 5.2.11, every section of the encoding part s'' contains exactly one "1" symbol; therefore, we can read this as an encoding of k vertices of the graph. Let $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$ be these vertices. Further, we know that the back tag s''' consists only of "0" symbols: By Lemma 5.2.10, the encoding part s' has only k "1"s; would s''' contain a "1", then we would have $d_H(s,t) > nk - k$. We have $d_H(s''', s'''_{i,j}) = nk - 2k + 2$ for *every* choice string match $s_{i,j}$ and, since every $s''_{i,j}$ contains only two "1" symbols, $d_H(s'', s''_{i,j}) \geq k-2$. Now consider some $1 \leq i < j \leq k$ and the corresponding choice string $c_{i,j}$. Since s is a solution, we know that there is a block $s_{i,j}$ with $d_H(s'', s''_{i,j}) = k - 2$. That means that the two "1" symbols in $s''_{i,j}$ have to match two "1" symbols in s''; this implies that the two vertices v_{h_i} and v_{h_j} are connected by an edge in the graph. Since this is true for all $1 \leq i < j \leq k$, vertices $v_{h_1}, \ldots, v_{h_k}$ are pairwisely interconnected by edges and form a k-clique. □

Propositions 5.2.8 and 5.2.12 yield the following main theorem:

Theorem 5.2.13. Closest Substring *is* W[1]*-hard for parameter* k *in the case of a binary alphabet.* □

5.2.4 Consensus Patterns

Our techniques for showing hardness of Closest Substring, parameterized by the number k of input strings, also apply to Consensus Patterns. Because of the similarity to Closest Substring, we restrict ourselves to explaining the problem and pointing out new features in the hardness proof.

Given strings $s_1, s_2, \ldots, s_k$ over alphabet Σ and integers d and L, the Consensus Patterns problem asks whether there is a string s of length L such that $\sum_{i=1}^{k} d_H(s, s'_i) \leq d$ where s'_i is a length L substring of s_i. Thus, Consensus Patterns aims for minimizing the *sum* of errors. Since errors are summed up over all strings, the value of d will, usually, not be a small and, therefore, the most significant parameterization for this problem seems to be the one by k. By reduction from Clique, we can show W[1]-hardness results as for Closest Substring given unbounded alphabet size. We omit the details here and focus on the case of binary input alphabet. We can apply basically the same ideas as were used in Subsection 5.2.3; however, some modifications are necessary.

5.2.4.1 Reduction from Clique to Consensus Patterns

Choice strings. As in Subsection 5.2.3.1, we generate a set of $\binom{k}{2}$ *choice strings* $S_c = \{c_{1,2},\ c_{1,2} \ldots, c_{k-1,k}\}$ with

$$c_{i,j} := \langle\text{block}(i,j,e_1)\rangle\langle\text{block}(i,j,e_2)\rangle \ldots \langle\text{block}(i,j,e_m)\rangle,$$

encoding the m edges of the input graph. This time, however, every block consists only of a front tag and an encoding part. No back tag is necessary. Therefore, we use $\langle\text{block}(i,j,(v_r,v_s))\rangle := \langle\text{front\_tag}\rangle\langle\text{encode}(i,j,(v_r,v_s))\rangle$, in which the encoding part $\langle\text{encode}(i,j,(v_r,v_s))\rangle$ is constructed as in Subsection 5.2.3.1. Before we explain the front tags, we already fix the distance value d.

Distance Value. We set the distance value $d := (\binom{k}{2} - (k-1))nk$.

Front tags. The front tag is now given by $(1^{nk^3}0)^{nk^3}0^{nk^3}$. Thus, the front tag has length $n^2k^6 + 2nk^3$. The front tag here is more complex than the one used in Subsection 5.2.3.1. The reason is as follows. Its purpose is to make sure that a substring which is not a block cannot be a match. To achieve this, the front tag lets such an unwanted substring necessarily have a distance value larger than d to a possible solution (as explained in the proof of Lemma 5.2.9). Since d has a higher value here compared to Subsection 5.2.3, we need the more complex front tag.

Solution length. We set the substring length to the length of one block, i.e., the sum of $n^2k^6 + 2nk^3$ (the length of the front tag) and nk (the length of the encoding part). Therefore, $L := n^2k^6 + 2nk^3 + nk$.

Template strings. In contrast to Subsection 5.2.3.1, we produce not only one but $\binom{k}{2} - (k-1)$ many template strings. All template strings have length L, i.e., the length of one block. The template strings are a concatenation of the front tag part (as given above) and an encoding part consisting of nk many symbols “1”.

In summary, the front tag ensures that only the block of a choice string can be selected as a substring matching a solution. Regarding the distribution of mismatches, we note that a closest substring's front tag part will not cause any mismatches. In its encoding part, every of its nk positions causes at least $\binom{k}{2} - (k-1)$ mismatches. It causes exactly $\binom{k}{2} - (k-1)$ mismatches for every position iff the input graph contains a k-clique.

5.2.4.2 Correctness of the Reduction

Proposition 5.2.14. *For a graph with a k-clique, the construction in Subsection 5.2.4.1 produces an instance of* Consensus Patterns *which has a solution, i.e., there is a string s of length L such that every $c_{i,j}$, $1 \le i < j \le k$, has a substring $s_{i,j}$ with $\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d_H(s, s_{i,j}) \le d$.*

Proof. Given an undirected graph G with n vertices and m edges, let $1 \leq h_1 < h_2 < \ldots < h_k \leq n$ be the indices of k-clique's vertices. Then, let string s consist of the front tag described in the above construction, concatenated with the encoding part $\langle \text{number}(h_1)\rangle\langle \text{number}(h_2)\rangle \ldots \langle \text{number}(h_k)\rangle$, which encodes all clique vertices. For every $1 \leq i < j \leq k$, we choose in choice string $c_{i,j}$ the block $s_{i,j}$ encoding the edge connecting vertices v_{h_i} and v_{h_j}. We will show that these blocks have exactly total Hamming distance $(\binom{k}{2} - (k-1))nk$ to s.

The front tags of s and of each $s_{i,j}$ coincide, their Hamming distance is 0. Recall from Subsection 5.2.3.1 that the encoding parts consist of k sections, each section of length n. We consider the encoding parts section by section and, within a section, columnwise. Given a section i', $1 \leq i' \leq k$, there are $k-1$ choice strings in which this section is active, and this section in these blocks encodes vertex $v_{h_{i'}}$. Consider the column at position $h_{i'}$ in this section, over all selected substrings and all template strings. We have $\binom{k}{2} - (k-1)$ "0" symbols from the choice strings in which this section is inactive; in all other strings, there is a "1" at this position. In s, this position is "1," causing $\binom{k}{2} - (k-1)$ mismatches. Now consider the remaining columns of section i'. In each of them, we have $\binom{k}{2} - (k-1)$ "1" symbols from the template strings; all $\binom{k}{2}$ choice strings have "0" at the corresponding position. In s, this position is "0," causing $\binom{k}{2} - (k-1)$ mismatches. Thus, we have $\binom{k}{2} - (k-1)$ mismatches at every of the n positions within a section, and this is true for all k sections of the encoding part. The sum of distances from s to the matches in choice strings and the template strings is $(\binom{k}{2} - (k-1))kn$; s is a solution. □

For the reverse direction, we use two lemmas to show important properties that a solution of the constructed instance has. The first lemma is proved in analogy to Lemma 5.2.9.

Lemma 5.2.15. *A solution s and all its matches in the input instance start with the front tag.* □

The second property of a solution, although also valid for the solutions in Subsection 5.2.3.2, is established in a different way here. It relies on the additional template strings that have been introduced in the construction of the Consensus Patterns instance.

Lemma 5.2.16. *A solution s contains exactly one symbol "1" in every section of its encoding part.*

Proof. Let s be a solution for the constructed Consensus Patterns instance. By Lemma 5.2.15, we know that s and all its matches in the choice strings start with the front tag. Consequently, the matches in the choice strings must be blocks.

Consider the encoding part of a solution s together with the encoding parts of its matches in the input strings. We note that we have at least $\binom{k}{2} - (k-1)$

mismatches for every column at positions p, $1 \leq p \leq nk$: On the one hand, all $\binom{k}{2} - (k-1)$ template strings have "1" symbols at position p. On the other hand, all $\binom{k}{2} - (k-1)$ choice strings in which position p's section is inactive have "0" at this position, no matter which blocks we chose in these choice strings. Since s is a solution and only a total of $(\binom{k}{2} - (k-1))nk$ mismatches are allowed, we have *exactly* $\binom{k}{2} - (k-1)$ mismatches for every position of the encoding part of s with the corresponding positions in the matches of s.

Now, consider an arbitrary section i', $1 \leq i' \leq k$, and consider all $k-1$ choice strings in which section i' is active. In these choice strings, section i' contains exactly one "1" symbol. We will show that in these choice strings' blocks that form the matches for s, the "1" in section i' must be at the same position in all matches because, otherwise, s is no solution. Assume that we chose blocks in which the "1" symbols of section i' are at different positions. We can easily check that this would cause more than $\binom{k}{2} - (k-1)$ mismatches for the columns corresponding to the positions of the "1" symbols; this contradicts the assumption that s is a solution. We conclude that, for all matches in choice strings, the "1" symbols of section i' must be at the same position. For columns in which we have "1" symbols in choice strings, there is a majority of "1" symbols, namely those in the $(k-1)$ choice strings in which section i' is active and those in the $\binom{k}{2} - (k-1)$ template strings. Therefore, the respective position in s must be "1." For all other columns, there is a majority of "0" symbols, namely those in all $\binom{k}{2}$ choice strings. Therefore, the respective position in s must be "0." □

These two lemmas allow us to show that also the reverse direction of the reduction is correct.

Proposition 5.2.17. *The k symbols "1" in the solution string's encoding part correspond to a k-clique in the graph.*

Proof. Let s be a solution for the constructed CONSENSUS PATTERNS instance. By Lemma 5.2.16, every section in the encoding part of s encodes a vertex of the input graph. In the following, we show that all encoded vertices are interconnected by edges.

Let $V_C = \{v_{h_1}, v_{h_2}, \ldots, v_{h_k}\}$ be the vertices encoded in the solution's encoding part. For every two sections $1 \leq i < j \leq k$, we select in choice string $c_{i,j}$ a substring in which the "1" symbols of sections i and j are at the same positions as the "1" symbols of sections i and j in the solution: Selecting another substring would result in a Hamming distance greater than $\binom{k}{2} - (k-1)$ in the h_ith and h_jth column and s could not be a solution. Hence, the selected block encodes the edge connecting v_{h_i} and v_{h_j}. Since we find such a substring for every $1 \leq i < j \leq k$, every pair of vertices in V_C is connected by an edge, V_C is a k-clique. □

Propositions 5.2.14 and 5.2.17 yield the following main result.

Theorem 5.2.18. CONSENSUS PATTERNS *is* W[1]*-hard for parameter* k *in case of a binary alphabet.* □

5.3 Conclusion and Open Questions

We conclude with some open questions and directions for further research:

1. We showed that, from the theoretical point of view, CLOSEST STRING is fixed-parameter tractable with respect to k. Our algorithm, however, suffers from huge constant factors in the running time, even for moderate values of k, that seem to make it impossible to find exact solutions with this algorithm for $k > 4$. Is it possible to give a fixed-parameter algorithm for parameter k that is usable for larger values of k and arbitrary values of L and d?

2. We considered CLOSEST STRING with respect to Hamming distance. What is, for constant alphabet size, the parameterized complexity of CLOSEST STRING with respect to parameter d when using edit distance instead, i.e., allowing insertions, deletions, and substitutions? Note that, for unbounded alphabet size, the NP-completeness result of Cascuberta and de la Higuera [55] implies that, CLOSEST STRING with respect to the parameter of edit distance is W[t]-hard for all $t > 0$. With respect to the number of input strings as parameter, Nicolas and Rivals [147] show that, even for binary alphabet, CLOSEST STRING is W[1]-hard when using edit distance.

3. What is the parameterized complexity of CLOSEST SUBSTRING with respect to parameter d and with respect to parameters d and k? Showing that CLOSEST SUBSTRING is W[1]-hard with respect to d would, with results from [41], imply that the problem does not have an efficient PTAS (EPTAS).

Chapter 6

Consensus of Quartets

To determine the evolutionary relationship of a set of taxa, e.g., based on DNA or protein sequence data, is an important question in computational biology. A common model for this relationship is an *evolutionary tree*, a binary tree T in which the leaves are bijectively labeled by the taxa.[1] In recent years, quartet methods for reconstructing evolutionary trees have received considerable attention [45, 108, 114]. Here, a *quartet* is a size four subset $\{a, b, c, d\}$ of the set of taxa and the *quartet topology* for $\{a, b, c, d\}$ induced by T simply is the four-leaved subtree of T for $\{a, b, c, d\}$.[2] The three possible quartet topologies for $\{a, b, c, d\}$ are $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$ as shown in Fig. 6.1—a fourth possible topology is the star topology which is not considered here because it is not binary. The fundamental goal of quartet methods is, given a set of quartet topologies, to reconstruct the corresponding evolutionary tree. Herein, the given set of quartet topologies can be incomplete, may contain errors or more than one topology for one quartet. Hence, to reconstruct (a good estimation of) the original evolutionary tree becomes an optimization problem, whose decision version generally turns out to be NP-hard.

In this chapter, we focus on the Minimum Quartet Inconsistency (MQI) problem.

> Minimum Quartet Inconsistency (MQI)
> **Input:** A set S of n taxa and a set Q_S of $\binom{n}{4}$ quartet topologies such that there is exactly one topology for *every* quartet corresponding to S and a non-negative integer k.
> **Question:** Is there an evolutionary tree T where the leaves are bijectively labeled by the elements from S such that the set of quartet topologies induced by T differs from Q_S in at most k quartet topologies?

[1] We follow, here, the commonly used demand for *binary* trees since evolutionary speciation events are thought to split up one species from another [158].

[2] Following a more graph-theoretical terminology, we can, equivalently, say that the quartet topology is the subtree induced by nodes a, b, c, and d in the tree.

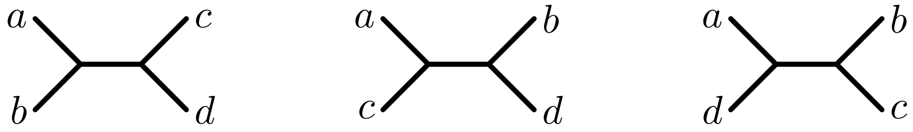


Figure 6.1: Possible quartet topologies for quartet $\{a, b, c, d\}$, which are (from left to right) $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$.

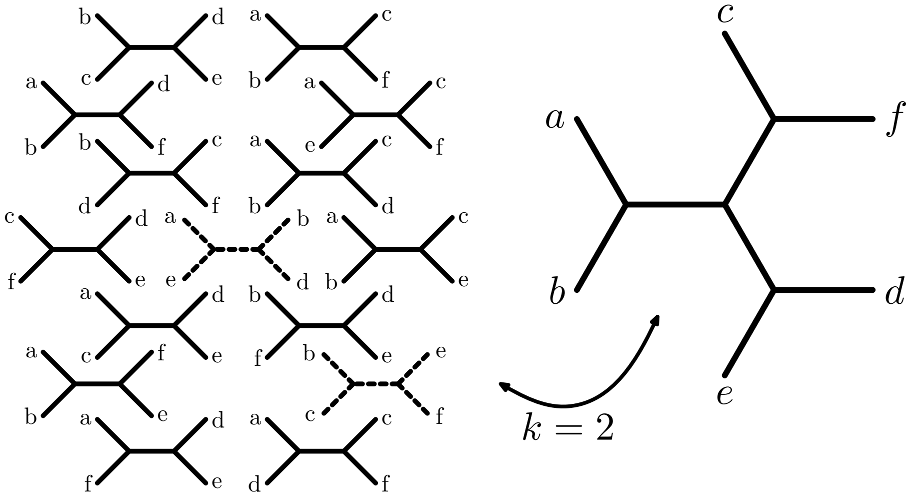


Figure 6.2: Example for an MQI instance consisting of 15 quartet topologies for taxa $\{a, b, c, d, e, f\}$ (left) such that, for $k = 2$, the tree (right) is the solution for MQI. The quartet topologies in which the given topologies differ from the topologies induced by the tree are indicated by dashed lines.

In Fig. 6.2, we display an example of a set containing 15 quartet topologies for 6 taxa and a tree which is a solution for the MQI instance with $k = 2$. MQI is NP-complete [25, 109]. It is worth noting, as was pointed out by Steel [185], that the quartet cleaning algorithm by Berry *et al.* [25] finds the optimal solution for instances with $k < (n-3)/2$. Therefore, MQI is NP-hard only for $k \geq (n-3)/2$. It is known that MQI is polynomial-time approximable with factor n^2 [108], and it is an open question of [108] whether MQI can be approximated with factor at most n or even with a constant factor. Heuristics for the problem include semidefinite programming [17] and the widely used *quartet puzzling* [190]. The parameterized complexity of MQI, however, so far, has not been discussed explicitly. We close this gap here and present the following results:

- We show that MQI can be solved exactly in worst-case $O(4^k n + n^4)$ time. Observe that the input size is $O(n^4)$.

- We discuss heuristic improvements to reduce the running time of our main algorithm in practical applications significantly.
- We exhibit several experiments on synthetic and real (fungi) data and, thereby, show that our algorithm (due to intensive tuning) in practice runs much faster than its theoretical (worst-case) analysis predicts.

To establish the correctness and the running time of our fixed-parameter algorithm, we exhibit some nice combinatorial properties of MQI. For instance, loosely speaking, we point out that "global conflicts" due to erroneous quartet topologies in fact can be led back to "local conflicts." The basis for this was laid by Colonius and Schultze [48], and by Bandelt and Dress [11]. This property is fundamental for our algorithms. Moreover, for minimal k, our approach makes it possible to construct *all* evolutionary trees that can be (uniquely) obtained from the given input by changing k quartet topologies. This puts the user of the algorithm in the position to select (e.g., based on additional biological knowledge) the probably best, most reasonable solution.

Note that the more general variant of MQI where the set Q_S is not required to contain a topology for every quartet (subsequently referred to as SPARSE MQI) is NP-complete even if $k = 0$ [184]. Hence, this excludes parameterized complexity studies and also implies inapproximability (with any factor).

6.1 Preliminaries on Quartet Methods

Quartet methods infer the evolutionary tree only for four taxa, called a *quartet*, at a time. Once having determined the evolutionary tree for every quartet of taxa, they try to combine these evolutionary trees involving four taxa, called *quartet topologies*, in order to obtain a tree containing all taxa at its leaves.

There are several reasons why quartet methods are widely used in practice. They are founded on the fact that an evolutionary tree is uniquely characterized by the quartet topologies for its size four sets of taxa [36]. From this set of topologies, we can efficiently compute the tree in $O(n^4)$ time [24]. Quartet methods clearly divide the tree construction process in two stages—we can use an arbitrary, even computationally expensive tree construction method for inferring the quartet topologies, while the recombination of topologies can be handled independently of the method chosen for inference. Another reason to use quartet methods is data disparity as discussed by Chor [45]: In practice, we often do not have the same amount of data for all considered taxa, e.g., not the same set of proteins for which the corresponding genes are known and sequenced. In general, tree construction methods cannot take advantage of information available only for a subset of taxa. Quartet methods, however, allow us to use the maximum amount of information available for the four taxa of a quartet when we compute its quartet topology.

The limitation of quartet methods in practice is caused by the process of quartet inference which can be erroneous. Therefore, we cannot be sure that there exists a tree inducing the inferred set of quartet topologies. Assuming that the number of errors is small compared to the number of correct topologies, we will try to overcome this problem by searching for a tree that matches the inferred topologies as "closely" as possible.

Recombination of quartet topologies without errors. Given exactly one quartet topology for every quartet of taxa, it is possible to decide in polynomial time whether there is a binary tree inducing all of the given quartet topologies, and, if so, to actually construct the tree [24]. Bryant and Steel [34] solve in $O(n^5)$ time a related problem in the case that we have one or two topologies for each quartet and the topologies are weighted; then, they ask for a tree inducing quartet topologies for which the sum of weights exceeds some given threshold. Both algorithms rely on the fact that the given set of topologies contains a topology for every quartet. In the more general situation in which we are not necessarily given a topology for every quartet, the problem of deciding whether there is a binary tree inducing all the given topologies is NP-complete [184].

Inferring strongly supported parts of the tree. There are situations in which there is no *binary* tree inducing the given set of topologies. In the following, we mention methods that infer bipartitions of taxa such that these bipartitions are particularly supported by the given quartet topologies and which can be seen as edges of a tree; these methods yield trees that are not necessarily fully resolved, i.e., that are not binary.

The Q* method by Berry and Gascuel [24] infers, for a complete set of quartet topologies, only the *completely supported* edges, an approach proposed by Buneman [36]: Given a tree with its leaves bijectively labeled by the given set S of taxa, an edge e in the tree defines a bipartition of S into sets A_e and B_e, each of them containing the taxa in the subtree rooted at one end of e. We call edge e completely supported if, for every $a, a' \in A_e$ and $b, b' \in B_e$, the corresponding quartet topology is $[aa'|bb']$. The set of quartet topologies induced by the completely supported edges is called Q*. Berry and Gascuel [24] present an algorithm computing Q* in $O(n^4)$ running time.

Another approach, called *quartet cleaning*, tries to correct obvious *quartet errors* if their number is bounded [23, 25, 107, 109, 194]. We distinguish *edge* and *vertex* quartet cleaning, as well as *global* and *local* algorithms. Edge quartet cleaning applies if the number of errors *across* one edge is smaller than some bound, in vertex quartet cleaning the number of errors across a vertex has to be bounded (for the definition of errors across an edge or a vertex, resp., see, e.g., [25]). Global quartet cleaning algorithms correct errors only if the errors are bounded for every edge or vertex. Local algorithms correct errors also when errors are bounded only for one edge or vertex. For an overview of quartet cleaning results refer to Della Vedova and Wareham [194]. The global edge quartet cleaning algorithm given by [25] computes the optimal tree if, for every edge e inducing a bipartition of taxa into sets A_e and B_e with $|A_e| \geq 2$ and $|B_e| \geq 2$, we have fewer than $(|A_e| - 1)(|B_e| - 1)/2$ quartet errors across this edge. This value is minimal for $|A_e| = 2$ and $|B_e| = n - 2$ and, therefore, quartet cleaning computes a guaranteed optimal solution if the total number of quartet errors is smaller than $(n - 3)/2$.

Minimum quartet inconsistency. In order to find the "best" binary tree for a given set of quartet topologies, we can ask for a tree that violates a minimum number of topologies. If we are given exactly one quartet topology for every set of four taxa, this question is the MQI problem. If there is not necessarily a quartet topology for every set of four taxa, the more general question is referred to as SPARSE MQI. Ben-Dor *et al.* [17] give an exact algorithm for the SPARSE MQI problem, based on dynamic programming. For every subset of i taxa, it computes the optimal tree for these taxa based on the optimal trees for the subsets of $i-1$ taxa, with i running up to the total number n of species. The resulting running time is $O(3^n \cdot m)$, where n is the number of species and m is the number of given quartet topologies, and the memory requirement is $\Theta(2^n)$.

Regarding heuristics, Ben-Dor *et al.* [17] use semidefinite programming to obtain, in polynomial time, possibly non-optimal solutions for SPARSE MQI. A widely used heuristic for MQI is *quartet puzzling* by Strimmer and von Haeseler [190]. Its main idea is to build the tree incrementally, starting with four taxa and adding one taxon at a time in a greedy way. To avoid local traps the algorithm repeats this process and, finally, constructs a (possibly non-binary) consensus of the single trees.

Not much is known about the approximability of MQI. Jiang *et al.* [108] mention a factor-n^2 approximation, at the same time asking for better approximation results. Note that the complement problem of MQI, where one tries to find a tree T that *maximizes* the number of given quartet topologies induced by T, possesses a polynomial-time approximation scheme [107, 109] which, however, is not used in practice due to its high running time.

Some notation. Assume that we are given a set of n taxa S. A set of quartet topologies is *complete* if it contains exactly one topology for every quartet of S. A complete set of quartet topologies for the quartets over S is denoted by Q_S. A set of quartet topologies Q is *tree-consistent* [11] if there exists a tree T such that, for the set Q_T of quartet topologies induced by T, we have $Q \subseteq Q_T$. Set Q is *tree-like* [11] if there exists a tree with $Q = Q_T$. Since an evolutionary tree is uniquely characterized by the topologies for all its quartets [36], a complete set of topologies is tree-consistent if and only if it is tree-like. Intuitively, a set of topologies has a "conflict" whenever it is not tree-consistent. We will call a conflict "global," when a complete set of topologies is not tree-consistent. In contrary, we call it "local," when a size three set of topologies, which necessarily is incomplete, is not tree-consistent.

6.2 Global Conflicts are Local

The key to develop a fixed-parameter solution for MQI is as follows: It is sufficient to examine the size three sets of quartet topologies and to recursively branch on local conflicts. We use results of Bandelt and Dress [11] who introduced the *substitution property* to identify subsets of quartet topologies which cause conflicts: Given a set of taxa S and a complete set of quartet topologies $\mathsf{Q_S}$ over these taxa, a set $\mathsf{S}_5 \subseteq \mathsf{S}$ of five taxa satisfies the *substitution property* if, for every choice of distinct $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e} \in \mathsf{S}_5$, $[\mathsf{ab}|\mathsf{cd}] \in \mathsf{Q_S}$ implies $[\mathsf{ab}|\mathsf{ce}] \in \mathsf{Q_S}$ or $[\mathsf{ae}|\mathsf{cd}] \in \mathsf{Q_S}$.

Proposition 6.2.1. *(Proposition 6 in [11]) Given a set of taxa* S *and a complete set of quartet topologies* $\mathsf{Q_S}$ *and some taxon* $\mathsf{f} \in \mathsf{S}$*, then* $\mathsf{Q_S}$ *is tree-like iff every size five set of taxa that contains* f *satisfies the substitution property.* □

The proof for Proposition 6.2.1 (given in [11]) relies on the “denseness” given in a complete set of quartet topologies (for short, we sometimes only write topologies).

In the following, we show that in Proposition 6.2.1, we can replace the substitution property with the more common term of tree-consistency.

Lemma 6.2.2. *Three topologies involving more than five taxa are tree-consistent.*

Proof. Assume we have topologies $\mathsf{t}_1, \mathsf{t}_2$, and t_3 involving more than five taxa. We distinguish two cases, in which Case (1) will apply if one of $\mathsf{t}_1, \mathsf{t}_2$, t_3 involves a taxon not occurring in the other two topologies. Case (2) will apply if, for each pair of topologies from $\mathsf{t}_1, \mathsf{t}_2$, t_3, there are exactly two taxa occurring in both topologies. Counting arguments make sure that either Case (1) or Case (2) applies. Assume that Case (1) does not apply: Then, we have three quartets, each of them containing four from the at least six given taxa, and every taxon has to occur in at least two of the three quartets. This is only possible for exactly six taxa—here Case (2) applies. With more than six taxa one would necessarily have a taxon occurring in only one of the topologies; this is handled in Case (1).

Case (1) A topology $\mathsf{t} \in \{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\}$ contains a taxon occurring in none of the other topologies. Assume, w.l.o.g., that $\mathsf{t} = \mathsf{t}_1 = [\mathsf{ab}|\mathsf{cd}]$ and that a is the taxon occurring only in t_1 and not in t_2 or t_3. We can certainly find a tree T inducing t_2, and t_3, since the topologies for only two different quartets are always tree-consistent.

In the case that b does occur in t_2 or t_3, we replace in T the leaf b by an inner node having two leaves as its children, one labeled with a and the other with b. In the case that b does not occur in t_2 and t_3, we also create a new inner node with children a and b, and insert it at some arbitrary edge of T.

The modified tree induces t_1, t_2, t_3, hence they are tree-consistent.

Case (2) For each pair of topologies from t_1, t_2, t_3, there are exactly two taxa occurring in both topologies. W.l.o.g., we can assume that topology t_1 is given for quartet $\{a, b, c, d\}$, topology t_2 is given for quartet $\{a, b, e, f\}$, and topology t_3 is given for quartet $\{c, d, e, f\}$. Checking all possible combinations of topologies for t_1, t_2, t_3 (we omit the details here), we find that we always can find a tree inducing t_1, t_2, t_3. □

When searching for local conflicts, Lemma 6.2.2 makes it possible to focus on the case of three topologies involving only five taxa. If the substitution property is not satisfied for taxa $a, b, c, d, e \in S$, since $[ab|cd] \in Q_S$ but $[ab|ce] \notin Q_S$ and $[ae|cd] \notin Q_S$, then we say that the topologies for the quartets $\{a, b, c, d\}$, $\{a, b, c, e\}$, and $\{a, c, d, e\}$ *contradict* the substitution property.

Lemma 6.2.3. *For a given a set of taxa* S*, three topologies consisting of taxa from* S *are tree-consistent iff they do not contradict the substitution property.*

Proof. First, we note that with three topologies involving more than five taxa, on the one hand, we can, according to Lemma 6.2.2, build a tree inducing these taxa and, on the other hand, these taxa cannot contradict the substitution property (the substitution property is formulated over five taxa only). Therefore, we can in the following focus on the case of three topologies involving only five taxa.

($\Rightarrow$) As the three topologies are tree-consistent, we can find a tree inducing the topologies. The set of induced topologies is tree-like. With Proposition 6.2.1 the substitution property holds.

($\Leftarrow$) We are given three topologies which do not contradict the substitution property and which involve five taxa $\{a, b, c, d, e\}$.

First, we want to reduce the number of cases we have to consider. For three topologies over five taxa which do not contradict the substitution property, we show that it is, w.l.o.g., possible to assume that two of them are $[ab|cd]$ and $[ab|ce]$. This means that two of the topologies have to be equal on one side. Assuming that this is not true leads to a contradiction. To see this, we take two topologies $t_1 = [ab|cd]$ and $t_2 = [ac|de]$, and show that there is no topology t_3 with the properties (1) that t_1, t_2, t_3 do not contradict the substitution property and (2) that no side of t_3 equals a side of t_1 or t_2. Topology t_3 cannot be a topology for quartets $\{a, b, c, e\}$ or $\{a, b, d, e\}$. The reason is that, given topology $t_1 = [ab|cd]$, the substitution property would require either topology $[ae|cd]$ (and $[be|cd]$) or topology $[ab|ce]$ (and $[ab|de]$). Since $[ae|cd]$ would contradict t_2, we necessarily would have that the topology is $[ab|ce]$ or $[ab|de]$. These, however, would contradict property (2) because they equal t_1 in the "ab side." Analogously, t_3 cannot be a topology for quartet $\{b, c, d, e\}$—the substitution property would require that the topology is $[bc|de]$, which would

Topology	Topology	Topology	Contradict	Completion
t_1	t_2	t_3	the subst. prop.	to tree-like set
$[ab \mid cd]$	$[ab \mid ce]$	$[ab \mid de]$	no	$[ae \mid cd]$, $[be \mid cd]$
		$[ad \mid be]$	yes	
		$[ae \mid bd]$	yes	
		$[ac \mid de]$	no	$[ab \mid de]$, $[bc \mid de]$
		$[ad \mid ce]$	no	$[ab \mid de]$, $[bd \mid ce]$
		$[ae \mid cd]$	no	$[ab \mid de]$, $[be \mid cd]$
		$[bc \mid de]$	no	$[ab \mid de]$, $[ac \mid de]$
		$[bd \mid ce]$	no	$[ab \mid de]$, $[ad \mid ce]$
		$[be \mid cd]$	no	$[ab \mid de]$, $[ae \mid cd]$

Table 6.1: The quartet topologies considered in the proof of Lemma 6.2.3.

contradict property (2) since it equals t_2 in the "de side." Since there are no quartets over $\{a, b, c, d, e\}$ remaining, there are no choices left for t_3. Therefore, our assumption was wrong. Thus, for three topologies over five taxa, where the topologies do not contradict the substitution property, this justifies that two of the topologies have to be equal on one side.

With the preceding considerations, we can, w.l.o.g., assume that two of the given topologies are $t_1 = [ab|cd]$ and $t_2 = [ab|ce]$. We are given a third topology t_3. There remain three quartets over $\{a, b, c, d, e\}$ whose topology can take this place. These quartets are $\{a, b, d, e\}$, $\{a, c, d, e\}$, and $\{b, c, d, e\}$.

In Table 6.1, we list the three quartets and, for each of these three quartets, the three possible topologies it can take. In case the resulting triple of topologies does not contradict the substitution property, we complete them to a set of tree-like topologies, as shown in the last column of Table 6.1. For these choices of t_3 we, thereby, show that t_1, t_2, t_3 are tree-consistent. In two of the listed cases, we cannot complete the three topologies to a tree-like set. We find, however, that those triples of topologies contradict the substitution property. With the choice of $t_3 = [ad|be]$, the substitution property requires that we have either topology $[ad|bc]$ or topology $[ac|be]$, in contradiction to topologies t_1 and t_2. Analogously, the topologies contradict the substitution property with the choice of $t_3 = [ae|bd]$. □

Note that Lemma 6.2.3 involving a necessarily incomplete set of three topologies does not generalize from size three to an incomplete set of arbitrary size, as exhibited in the following example. For taxa $\{a, b, c, d, e, f\}$, consider the incomplete set of topologies $[ab|cd]$, $[bc|de]$, $[cd|ef]$, and $[af|de]$. Without going into the details, we only state here that these topologies are not tree-consistent, although there are no three topologies which contradict the substitution property.

With Lemma 6.2.3 we can now give another interpretation of Proposition 6.2.1.

Theorem 6.2.4. *Given a set of taxa* S*, a complete set of quartet topologies* Q_S

over S*, and a taxon* $\mathsf{f} \in \mathsf{S}$*,* $\mathsf{Q_S}$ *is tree-like (and, thus, tree-consistent) iff every set of three topologies from* $\mathsf{Q_S}$ *which involves* f *is tree-consistent.*

Proof. By Lemma 6.2.3 we replace the substitution property in Proposition 6.2.1 with tree-consistency. □

Given a complete set of topologies $\mathsf{Q_S}$ for a set of taxa S, we do not necessarily know whether the set is tree-like or not. If it is not, we can, according to Theorem 6.2.4, choose an arbitrary taxon $\mathsf{f} \in \mathsf{S}$ and track down a subset of three topologies that involves f and that is not tree-consistent. Our goal will be to detect all these local conflicts involving f. This will be the preprocessing stage of the algorithm that will be described in Section 6.4; the subsequent stage of the algorithm will (try to) "repair" these conflicts. We can find all local conflicts involving f in $\mathsf{O}(\mathsf{n}^4)$ time as follows. Since, following Lemma 6.2.2, only three topologies involving five taxa can form a local conflict, it suffices to consider all size four sets of taxa $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\} \subseteq \mathsf{S}$ together with the chosen $\mathsf{f} \in \mathsf{S}$. There are five quartets over this size five set of taxa, namely, $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$, $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{f}\}$, $\{\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{f}\}$, $\{\mathsf{a},\mathsf{c},\mathsf{d},\mathsf{f}\}$, and $\{\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{f}\}$. For the topologies of these quartets, we can test, in constant time, whether there are three among them that are not tree-consistent. Doing so for every choice of $\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\} \subseteq \mathsf{S}$, we will find *all* local conflicts involving f. We summarize these considerations in the following lemma.

Lemma 6.2.5. *If we are given a set* S *of taxa, some arbitrary taxon* $\mathsf{f} \in \mathsf{S}$*, and a complete set* $\mathsf{Q_S}$ *of quartet topologies that is not tree-consistent, then* $\mathsf{Q_S}$ *has at least one local conflict involving* f *and all local conflicts involving* f *can be found in* $\mathsf{O}(\mathsf{n}^4)$ *time.* □

6.3 Combinatorics of Local Conflicts

Given three topologies, we need to decide whether they are tree-consistent or not. Directly using the definition of tree-consistency turns out to be a rather technical, troublesome task since we have to reason whether or not a tree topology exists that induces the topologies. Similarly, it can be difficult to test, for the topologies, whether or not they contradict the substitution property. To make things less technical and easier to grasp, we subsequently give a combinatorial characterization of local conflicts. Note that in the following definition we distinguish two possible *orientations* of a quartet topology $[\mathsf{ab}|\mathsf{cd}]$, namely $[\mathsf{ab}|\mathsf{cd}]$ and $[\mathsf{cd}|\mathsf{ab}]$.

Definition 6.3.1. *Given a set of topologies where each of the topologies is assigned an orientation, let* l *be the number of different taxa occurring in the left-hand sides of the topologies and let* r *be the number of different taxa occurring in the right-hand sides of the topologies.*

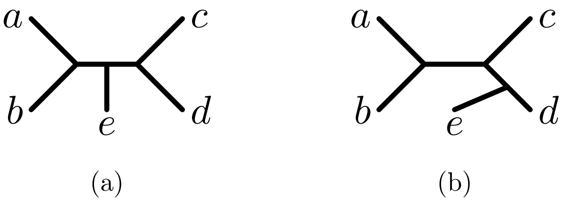


Figure 6.3: Trees inducing non-conflicting topologies in the proof of Theorem 6.3.2.

The signature *of the set of topologies, then, is the pair* (l, r) *that, over all possible orientations for these topologies, minimizes* l.

Theorem 6.3.2. *Three quartet topologies are not tree-consistent iff they involve five taxa and their signature is* $(3, 4)$ *or* $(4, 4)$.

Proof. ($\Rightarrow$) We show that, given three topologies t_1, t_2, t_3 which are not tree-consistent, they involve five taxa and have signature $(3, 4)$ or $(4, 4)$. We know, by Lemma 6.2.3, that three topologies are not tree-consistent iff they contradict the substitution property. Three topologies contradict the substitution property if for one of these topologies, w.l.o.g., $t_1 = [ab|cd]$, neither the topology t_2 for quartet $\{a, b, c, e\}$ is $[ab|ce]$ nor the topology t_3 for quartet $\{a, c, d, e\}$ is $[ae|cd]$. Therefore, the topology t_2 is either $[ac|be]$ or $[ae|bc]$, and the topology t_3 is either $[ac|de]$ or $[ad|ce]$. By exhaustively checking the possible combinations, we can find that the topologies involve five taxa and their signature is $(3, 4)$ (e.g., for $t_2 = [ac|be]$ and $t_3 = [ac|de]$) or $(4, 4)$ (e.g., for $t_2 = [ac|be]$ and $t_3 = [ad|ce]$).

($\Leftarrow$) We are given three topologies t_1, t_2, t_3 involving five taxa and having signature $(3, 4)$ or $(4, 4)$. Assume that they are tree-consistent. Showing that this would imply signature $(2, 3)$ or $(3, 3)$, we prove that they cannot be tree-consistent. For tree-consistent t_1, t_2, t_3, we can find a tree inducing them. Given, w.l.o.g., taxa $\{a, b, c, d, e\}$ and $t_1 = [ab|cd]$, we essentially have two possibilities: we can attach the leaf e on the middle edge of topology t_1 as shown in Fig. 6.3(a), or we can attach e on one of the four side branches of t_1 as exemplarily shown in Fig. 6.3(b). Considering the set of quartet topologies induced by these trees, we find in each case that the set has signature $(3, 3)$ or $(2, 3)$. For instance, the topologies induced by the tree in Fig. 6.3(a) are, besides t_1, $[ab|ce]$, $[ab|de]$, $[ae|cd]$, and $[be|cd]$. Three topologies selected from these, have signature $(3, 3)$ (e.g., $[ab|cd]$, $[ab|ce]$, and $[ae|cd]$) or $(2, 3)$ (e.g., $[ab|cd]$, $[ab|ce]$, and $[ab|de]$). □

Using Theorem 6.3.2, we can determine whether three topologies are conflicting by simply counting the involved taxa and computing their signature.

6.4 A Fixed-Parameter Algorithm for MQI

We describe a recursive algorithm for MQI in its four main parts: building the conflict list, the search tree, branching, and updating the conflict list.

Building the conflict list. We initially build the *conflict list* C of local conflicts, i.e., a list of size-three sets (or *three-sets* for short) of quartets whose topologies are not tree-consistent. By Lemma 6.2.5, it is sufficient to build a list of local conflicts containing some designated taxon, which can be chosen arbitrarily.

The search tree. The recursive procedure is outlined in Fig. 6.4. The procedure selects a local conflict from the conflict list and tries to resolve it by changing one of its topologies. After such a change it updates the conflict list (described below), and calls the recursive procedure with argument $k-1$ on the thereby created subcase. In the next paragraph, we will explain how to select and change the topologies in this branching and we will find that it is sufficient to branch into four subcases. The recursion stops if no conflicts are left in the conflict list (we have found a solution), or if $k = 0$ (with a non-empty conflict list we did not find a solution in this branch of the search tree). When a solution is found, the algorithm outputs the current, complete and tree-like set of topologies. From this, it is possible to derive the evolutionary tree in $O(n^4)$ time [24]. Scanning the whole search tree, we can find *all* solutions that can be obtained by altering k topologies when k is optimal.

Branching. By a careful selection of subcases to branch into, we can explore all ways to resolve an arbitrarily selected local conflict with only four recursive calls. Let t_1, t_2, t_3 be the three topologies which are not tree-consistent and which form the local conflict. By Lemma 6.2.3, t_1, t_2, t_3 contradict the substitution property. Recall that, given $[ab|cd]$, the substitution property (Proposition 6.2.1) requires topology $[ab|ce]$ or topology $[ae|cd]$. Therefore, we can assume the following setting for the three quartets contradicting the substitution property: Topology $t_1 = [ab|cd]$, topology t_2 is the topology for quartet $\{a, b, c, e\}$ different from $[ab|ce]$, and topology t_3 is a topology for quartet $\{a, c, d, e\}$ different from $[ae|cd]$. In order to change the three topologies such that they satisfy the substitution property, we have the following possibilities. We can change t_1; either (1) we change t_1 to $[ac|bd]$, or (2) we change t_1 to $[ad|bc]$. The cases in which t_1 is not changed remain to be covered. With unchanged t_1, we have to (3) change t_2 to $[ab|ce]$ or (4) change t_3 to $[ae|cd]$ because these are the only remaining possibilities to satisfy the substitution property. These four cases are, for an example of a local conflict, depicted in Fig. 6.5.

Updating the conflict list. The main task in every node of the search tree is to update the conflict list after changing a topology. In Fig. 6.2, this is done by the instruction `update(C, t')`, called with a conflict list C and a changed topology t' as arguments. We search, when changing t', the "neighborhood"

```
Recursive procedure resolve($Q$, $k$, $C$):
Input: Complete set $Q$ of quartet topologies, a non-negative
        integer $k$, and a list $C$ of all three-sets of conflicting quartets in $Q$.
Output: A complete tree-like set $Q'$, if existent, of quartet topologies such
        that $Q'$ differs from $Q$ in at most $k$ quartet topologies.
Method:
(Case 0) if $C$ is empty then                    /* we have found a solution */
            output the current set of quartet topologies and stop;
(Case 1) if $(k = 0)$ then return;               /* more than k recursive calls */
(Case 2) Choose $c \in C$, with $c = \{t_1, t_2, t_3\}$ such that
            the substitution property states "$t_1 \in Q \Rightarrow (t'_2 \in Q$ or $t'_3 \in Q)$"
            where $t'_2$ and $t'_3$ are alternative topologies for $t_2$ and $t_3$, resp.
         for the two alternative topologies $t'_1$ of $t_1$ do
            $C_{new}$ = Update($C$, $t'_1$);
            resolve(($Q$-$\{t_1\}$)$\cup\{t'_1\}$, $k-1$, $C_{new}$);
         end do
         $C_{new}$ = Update($C$, $t'_2$);
         resolve(($Q$-$\{t_2\}$)$\cup\{t'_2\}$, $k-1$, $C_{new}$);
         $C_{new}$ = Update($C$, $t'_3$);
         resolve(($Q$-$\{t_3\}$)$\cup\{t'_3\}$, $k-1$, $C_{new}$);
         return; /* no success in current branch
                  → step one level up in recursion */
```

Figure 6.4: Outline in pseudocode of a recursive procedure for eliminating conflicts by changing at most k quartet topologies (if possible). Further explanation is given in Section 6.4.

of t', and update the conflict list: We (1) remove the three-sets of quartets in the list whose topologies are now tree-consistent, and (2) add the three-sets of quartets not in the list whose topologies now form a local conflict.

Correctness. To obtain a non-conflicting set of quartet topologies, we have to, following Theorem 6.2.4, resolve all local conflicts. Such a local conflict can be removed by altering (at least) one of the three involved quartet topologies. The recursive procedure has to try every possibility to resolve the local conflict in order to find every possible solution. If there is a solution, we will find it by the described branching strategy. If for none of the three topologies we can find a solution while altering the topology, the conflict cannot be removed.

Running time. Initially building the conflict list takes, by Lemma 6.2.5, $O(n^4)$ time. Using the conflict list, we can, in constant time, access a local conflict and determine the cases to branch into. Since it is sufficient to branch into four subcases for a local conflict and since, in every subcase, we decrease the parameter k by one, we obtain a search tree size at most 4^k.

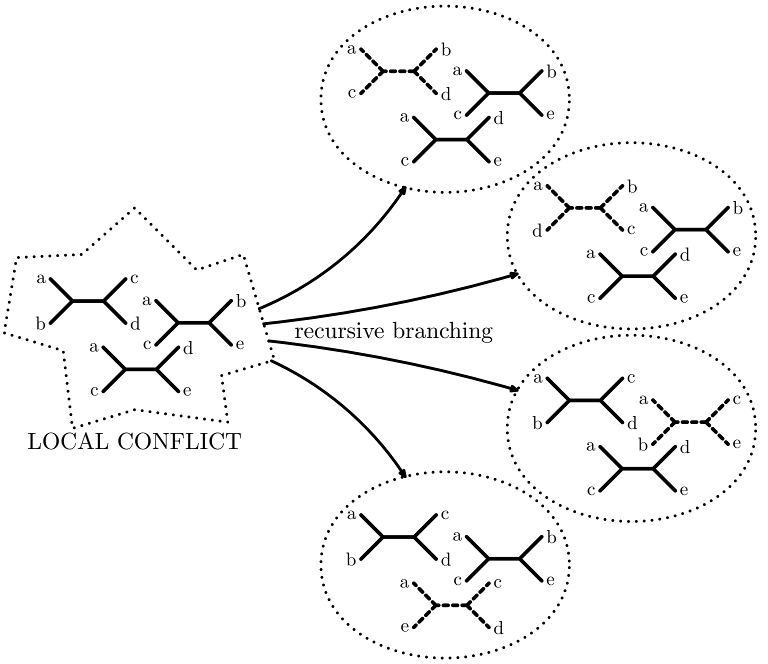


Figure 6.5: Example of a local conflict (left) and the four possible ways (right) which are considered by the algorithm to resolve the local conflict by changing one of the three topologies. The changed quartet topologies are indicated by dashed lines.

With n species, updating the list of conflicting size three sets can be done in $O(n)$ time: Following Lemma 6.2.2, local conflicts can only occur among three topologies consisting of no more than five taxa. Therefore, having changed the topology of one quartet $\{a, b, c, d\}$, we only have to examine the "neighborhood" of the quartet, i.e., those sets of five taxa containing all of a, b, c, d. For every such set of five taxa it can be examined in constant time whether for three topologies over the five taxa a new conflict emerged or whether an existing conflict has been resolved. Given taxa a, b, c, d, we have $n - 4$ choices for a fifth taxon. Thus, $O(n)$ is an upper bound for the update procedure.[3]

Altogether, we obtain:

Theorem 6.4.1. MQI *can be solved in* $O(4^k \cdot n + n^4)$ *time.* □

Our algorithm has only $O(n^4)$, i.e., up to constant factors optimal memory requirement since the input size is already $O(n^4)$: Firstly, we have to store the topology for every of the $O(n^4)$ quartets. Secondly, we maintain the conflict

[3] In fact, as explained in Section 6.2, we only consider sets of five species containing a designated taxon f. Therefore, if we change the topology of a quartet $\{a, b, c, d\}$ which does not contain the designated taxon f, then we only have to consider *one* set of five topologies, namely $\{a, b, c, d, f\}$. In this special case, the update procedure can be done in $O(1)$ time.

list; as observed in Section 6.2, the size of the conflict list is $O(n^4)$. Finally, we have to keep track of the topologies changed on the path from the root of the search tree to the current search tree node; this are at most $k < n^4$ topologies. For a generalization of our algorithm to weighted quartets refer to [89].

6.5 Improving the Running Time in Practice

In Section 6.5.1, we collect some ideas for improvements still maintaining the algorithm's optimality. In Section 6.5.2, sacrificing guaranteed optimality, we propose to combine the algorithm with existing methods that infer strongly supported parts of a tree.

6.5.1 Enhancements Maintaining Optimality

Fixing topologies. Once a topology has been altered, we will, in subsequent stages of recursion, never alter it again. We call this *fixing* the topology. This will avoid redundant branchings in the search tree.

Forcing topologies to change. In contrary to the fixing of topologies, it might be possible to identify topologies which necessarily have to be altered in order to find a solution. We call this *forcing* a topology to change. The ideas described here are similar to those used in the reduction to a problem kernel for the 3-Hitting Set problem given in [153]. Here, however, we will not obtain a problem kernel for our MQI. Nevertheless, the following ideas are likely to result in a better performance of the algorithm since they allow recognizing situations in which we cannot find a solution and they also allow for a better branching.

Lemma 6.5.1. *Consider an instance of* MQI *in which quartet* q *has topology* t*. If there are more than* $3k$ *distinct local conflicts which contain* t*, then in a solution for this instance the topology for* q *is different from* t*.*

Proof. We have shown in Section 6.2 that three topologies only can form a local conflict, if there are not more than five taxa occurring in them (Lemma 6.2.2). For five taxa, there are five quartets consisting of these taxa. Therefore, when we are given two quartet topologies t_1 and t_2 and if there are more than five taxa occurring in t_1 and t_2, they cannot form a conflict with a third topology. If there are exactly five taxa occurring in t_1 and t_2, then there are five quartets consisting of these five taxa, two of which are the quartets for t_1 and t_2. The remaining three topologies are the only possibilities for a topology t_3 that could form a conflict with t_1 and t_2.

Now, consider the situation in which, for a quartet topology t, we have more than $3k$ distinct local conflicts which contain t. We show by contradiction that we have to alter topology t to find a solution. Assume that we can find a solution while not altering t. By changing a topology t', we can cover at most three conflicts containing t since there are at most three local conflicts containing both t and t'. Therefore, by changing k topologies, we can resolve at most $3k$ local conflicts. This contradicts our assumption and shows that we have to alter t to find a solution. □

We call the topologies obtained from Lemma 6.5.1 "forced to change," and mark them appropriately in order to take them into consideration in the next branching situation.

Recognizing hopeless situations. In this paragraph, we describe situations in which, at some level in the search tree where we are allowed to alter at most k topologies, we cannot find a solution. This will allow us to avoid branching into further (useless) subcases and, thus, to cut off parts of the search tree. Having a local conflict consisting only of fixed topologies, we obviously cannot resolve this conflict while not changing one of the fixed topologies.

If, after identifying the topologies forced to change, there are more than k of them, it is obvious that a solution is not possible—already by changing these topologies we would change more topologies than we are allowed to.

The following two lemmas contain more involved observations. If a local conflict does not contain a topology which is forced to change, then we call it an *unforced* local conflict.

Lemma 6.5.2. *Let us have an instance of* MQI *in which we have identified* p *conflicts which are forced to change. If the number of unforced local conflicts is greater than* $3(k-p)k$*, then the instance has no solution.*

Proof. We have to change the p topologies that are forced to change. We, therefore, decrease the parameter by p and have the possibility to resolve all local conflicts containing such a topology. The conflicts which certainly remain to be resolved are the unforced conflicts. From the preceding paragraph we know that, by changing a topology, we can resolve at most $3k$ distinct local conflicts. Therefore, by altering $(k-p)$ topologies, we can resolve at most $3(k-p)k$ distinct local conflicts. □

Lemma 6.5.3. *An instance of* MQI *in which the number of local conflicts is greater than* $6(n-4)k$ *has no solution.*

Proof. By Lemma 6.2.2, local conflicts can only arise between three topologies that do not involve more than five taxa. Thus, given a quartet $q = \{a, b, c, d\}$ with topology t, a local conflict can arise with other quartets involving taxa from $\{a, b, c, d, e\}$ for some e. Since e has to be different from a, b, c, and d, there are $n-4$ choices for this taxon e. There are five quartets over $\{a, b, c, d, e\}$ and four of them excluding the given q. We have $\binom{4}{2} = 6$ ways to choose two

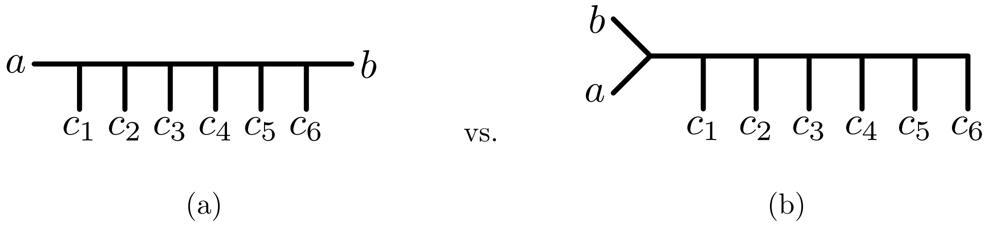

a
b
c_1 c_2 c_3 c_4 c_5 c_6
vs.
b
a
c_1 c_2 c_3 c_4 c_5 c_6
(a)
(b)

Figure 6.6: Example where the result from the Q* method leads to a suboptimal solution for MQI.

from these four quartets in order to form size three sets containing q. Therefore, by altering one topology, we can resolve at most $6(n-4)$ distinct local conflicts, and by altering k topologies at most $6(n-4)k$ distinct local conflicts. □

6.5.2 Fixing Strongly Supported Edges in Advance

To improve the performance of exact fixed-parameter algorithms in practice, it is reasonable to combine them with known heuristics. For MQI, we propose to preprocess the quartet topologies using methods that infer strongly supported edges of the tree. Examples of those methods include the Q*-method [24], quartet cleaning [25, 194], or hypercleaning [23]. The advantage of the methods is that they are fast (e.g., hypercleaning runs in $O(n^5)$). Their disadvantage is that they resolve only the strongly supported part of the tree's edges; on unfavorable input they resolve no edges at all. We can take advantage of their output, however, by fixing the topologies induced by the inferred edges. On this modified input, we run our MQI algorithm in order to completely resolve the partly resolved tree. Even a small set of fixed edges can significantly prune our search space.

Preprocessing by the Q*-method. Here, we describe the use of the Q*-method by Berry and Gascuel [24] as a preprocessing for our algorithm. The Q*-method produces the maximum subset of the given quartet topologies that is tree-like. In the combined use with our algorithm, we fix these quartet topologies from the beginning. The tree we obtain will be a refinement of the tree reported by the Q*-method.

We cannot guarantee that the reported tree is the optimal solution for MQI as illustrated by the following example. Consider the tree in Fig. 6.6(a) on taxa $\{a, b, c_1, c_2, \ldots, c_6\}$. Suppose we are given the complete set of 70 quartet topologies, where the 55 quartets containing at least three taxa taken from $\{c_1, c_2, \ldots, c_6\}$ have the topology induced by the tree, but the remaining 15 quartets have the topology $[ab|c'c'']$ where $c', c'' \in \{c_1, c_2, \ldots, c_6\}$ and $c' \neq c''$. On this input, the Q*-method infers the bipartition $ab|c'c''$ for all $c', c'' \in \{c_1, c_2, \ldots, c_6\}$; then applying the MQI algorithm leads to a solution for $k = 20$, e.g., the tree shown in Fig. 6.6(b) whose induced topologies differ from the given topologies for quartets $\{b, c, c', c''\}$ with $c, c', c'' \in \{c_1, c_2, \ldots, c_6\}$ (there are $\binom{6}{3} = 20$ possibilities to choose c, c', c''). The optimal solution of MQI, however, would be the tree depicted in Fig. 6.6(a) for $k = 15$.

The examples where the Q*-method yields misleading results are quite artificial. On real data, these mistakes are unlikely: before reporting misleading edges, i.e., edges not belonging to an optimal solution, the Q*-method will rather report no edge at all. Our experiments described in Section 6.6 support our conjecture that with the preprocessing by the Q*-method we find every solution that the MQI algorithm would find. Moreover, the experiments show that this enhancement allows us to process much larger instances than we could without using it.

6.6 Experimental Evaluation

To investigate the usefulness and practical relevance of our algorithm, we performed experiments on synthetic as well as on real data from fungi. The implementation of the algorithm was done using the programming language C. The algorithm contains the enhancements described in Section 6.5. The combined use with the Q*-method is, however, only applied when processing the fungi data, not when processing the synthetic data. The reported tests were done on a LINUX PC with a Pentium III 750 MHz processor and 192 MB main memory.

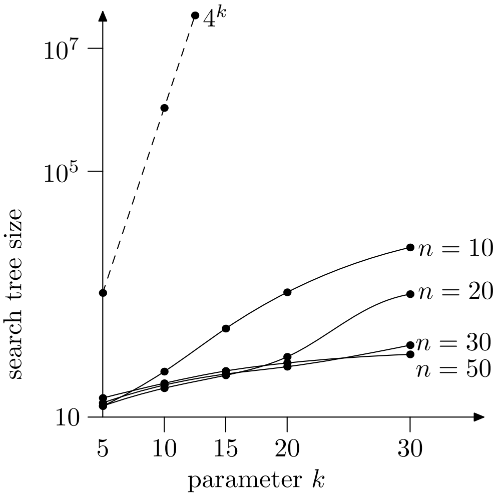
6.6.1 Synthetic Data

We performed experiments on artificially generated data in order to get some idea about the practical contexts in which our algorithm might be useful. For n given taxa and parameter k, we produced a data file as follows. We generated an evolutionary tree by recursively joining randomly selected subtrees. The subtrees were selected from a set which, initially, contained only the one-node subtrees corresponding to the taxa. When two subtrees were joined, we replaced them in the set by the newly generated subtree. This procedure, finally, yielded a tree for n taxa and we derived the quartet topologies from that tree. From these quartet topologies, we changed k distinct, arbitrarily selected topologies in a randomly chosen way. For different pairs of values for n and k, ten different data sets were created for each pair. The reported results denote the average for test runs on ten data sets.

We experimented with different values of n and k. As a measure of performance, we used two values: We measured the processing time and, since processing time is heavily influenced by system conditions, e.g., memory access time in case of cache faults, also the search tree size. The search tree size is the number of the search trees nodes.

			search tree
n	k	time	size
10	10	0.02 sec	58
	20	0.29 sec	1070
	50	26 sec	104297
20	10	0.13 sec	30
	20	0.16 sec	94
	50	25 sec	34693
	100	50 min	4460508
30	20	0.72 sec	66
	50	9 sec	7007
	100	31 min	1519700
50	50	7 sec	186
	100	8 min	45514
	150	<5 h	6119560

(a)



(b)

Figure 6.7: Table (a) displays the results of our algorithm on MQI instances for different values of n and k. We give processing time and the size of the scanned search tree. Fig. (b) displays, on a logarithmic scale, the difference of the theoretical 4^k bound (dashed) and the real search tree size (solid lines). Each solid line shows, for a fixed number of taxa n, how the search tree size increases for increasing values of k.

Fig. 6.7(a) gives a table of results for different values of n and k. We could process large instances of the problem, e.g., $n = 50$ and $k = 100$ in eight minutes. Regarding the search tree size, we compare in Fig. 6.7(b), on a logarithmic scale, the theoretical upper bound of 4^k to the real size of the search tree. The search trees are, by far, smaller than the 4^k bound. This is mainly due to the practical improvements of the algorithm (see Section 6.5). We also note that, for equal value of k, a higher number of taxa n results in a smaller search tree, if the value of k exceeds some turning point; this is due to the fact that the number of "correct" quartet topologies becomes, with growing n, higher compared to the at most k "incorrect" topologies and, therefore, the practical improvements presented in Section 6.5 apply more often.

6.6.2 Real Data

Using our algorithm, we analyzed the evolutionary relationships of three sets of fungi species.

In our first example, we considered species from the mushroom genus *Amanita*, a group that includes well-known species like the Fly Agaric and the Death Cap. The underlying data were an alignment of nuclear DNA sequences coding for the D1/D2 region of the ribosomal large subunit (alignment length 576) from *Amanita* species and one outgroup taxon, as used by Weiß *et al.* [199,

	running time in sec.	
n	no Q*	with Q*
8	0.46	0.36 (21%)
9	3.41	0.85 (32%)
10	35.96	2.68 (38%)
11	617.56	4.11 (41%)
12	7039.82	5.44 (43%)

Figure 6.8: Speed-up when using Q* preprocessing on species taken from the Amanita dataset described in Subsection 6.6.2. The value given in brackets is the percentage of quartet topologies fixed by Q*.

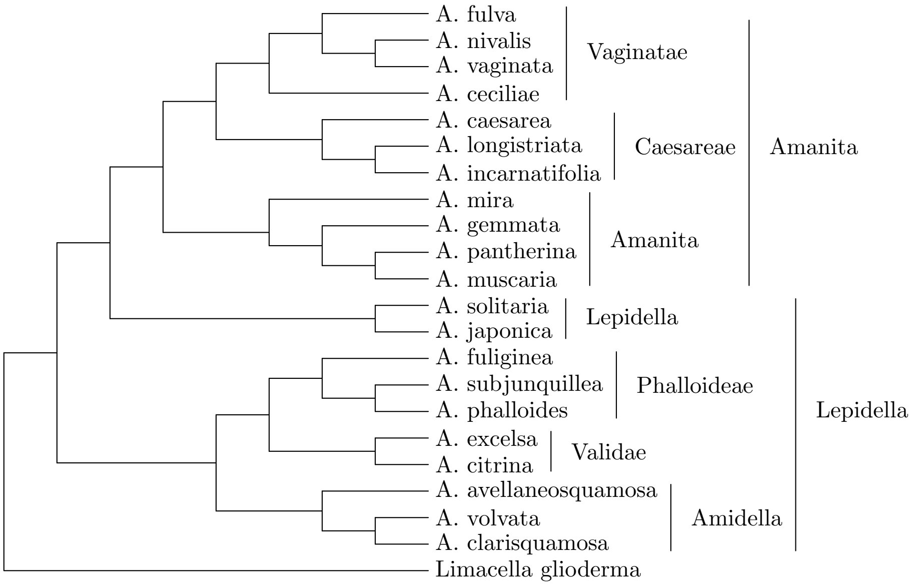


Figure 6.9: Optimal tree found for a set of 21 *Amanita* species and one outgroup taxon; indicated is the grouping of *Amanita* species into 7 sections and 2 subgenera.

200]. We inferred the quartet topologies by (1) using `dnadist` from the Phylip package [71] to compute pairwise distances with the maximum likelihood metric, and (2) using `distquart` from the Phyloquart package [24] to infer quartet topologies based on the distances.

The construction of the evolutionary tree was done by a preprocessing of the data using the Q*-method, which was also taken from the Phyloquart package. Experiments on small instances, e.g., 10 taxa, showed that all solutions we found without using the Q*-method are also found when using it. Using the Q*-method, however, resulted in a significant speed-up of the processing. Fig. 6.8 shows this impact for small numbers of *Amanita* species. Note, however, that the speed-up heavily depended on the data. In our results shown in Fig. 6.8

and in the following, we neglect the time needed for the preprocessing by the Q*-method, which was, e.g., 0.11 seconds for $n = 12$.

Our recursive algorithm processed a set of $n = 22$ taxa in 23 minutes (which includes 0.2 seconds needed by Q*). The resulting tree was rooted using the outgroup taxon *Limacella glioderma* and is displayed in Fig. 6.9. The first k-value for the given 7315 quartet topologies for which we found a solution is $k = 978$. The Q*-method had fixed 41 percent of the quartet topologies in advance. Considering the tree, the grouping of taxa is consistent with the grouping into seven sections supported by Weiß *et al.* [200], who used the distance method neighbor joining, heuristic parsimony methods, and maximum likelihood estimations. Particularly, our grouping is nearly identical to the topology revealed by Weiß *et al.* using maximum likelihood estimation. This topology is well compatible with classification concepts based on morphological characters, e.g., the sister group relationship of sections *Vaginatae* and *Caesareae*, and the monophyly of subgenus *Amanita* which were not supported in the neighbor joining and heuristic parsimony analysis.

To contrast our results with another heuristic method, we ran quartet puzzling with 1000 puzzling steps on the same data set. The resulting tree was computed in 1.3 minutes and also reflects the mentioned grouping of taxa into the seven subsections. However, it could not resolve the grouping of subsections and did, in particular, not indicate the division into two sections.

To assess our method on further datasets, we shortly describe two further analyses made on fungi sequence data; these sequences were taken from the TreeBase database [137] which also provides the corresponding published phylogenetic analyses.

Taking the data from Oda *et al.* [155], we considered a set of 26 sequences, namely length-875 nucleotide sequences of the internal transcribed spacer region of nuclear ribosomal DNA. In [155], these sequences were analyzed by a neighbor joining analysis using PHYLIP [71]. Within 17 minutes, we found a solution with $k = 762$ depicted in Fig. 6.10. The tree agrees in most parts with the tree published in [155], except of an exchange of the taxa *Amanita ceciliae* and *Amanita vaginata*. This example shows that we can tolerate larger numbers of taxa provided that the quality of quartets is "good," i.e., only a limited number of quartet topologies has to be changed in order to obtain a tree.

We analyzed the data presented in Hughey *et al.* [102], and considered one example set of sequences more closely here. This set consisted of length-469 sequences of mitochondrial small rDNA, for 19 representatives of fungi genera including several representatives of the familiy *Boletales*. The objective in [102] was to study the relationship of the species *Calostoma* in comparison with other Basidiomycetes. There, the dataset was analyzed with heuristic maximum parsimony techniques using PAUP. With our method, we found a solution with $k = 289$ within 15 minutes, and the corresponding tree is depicted in Fig. 6.11.

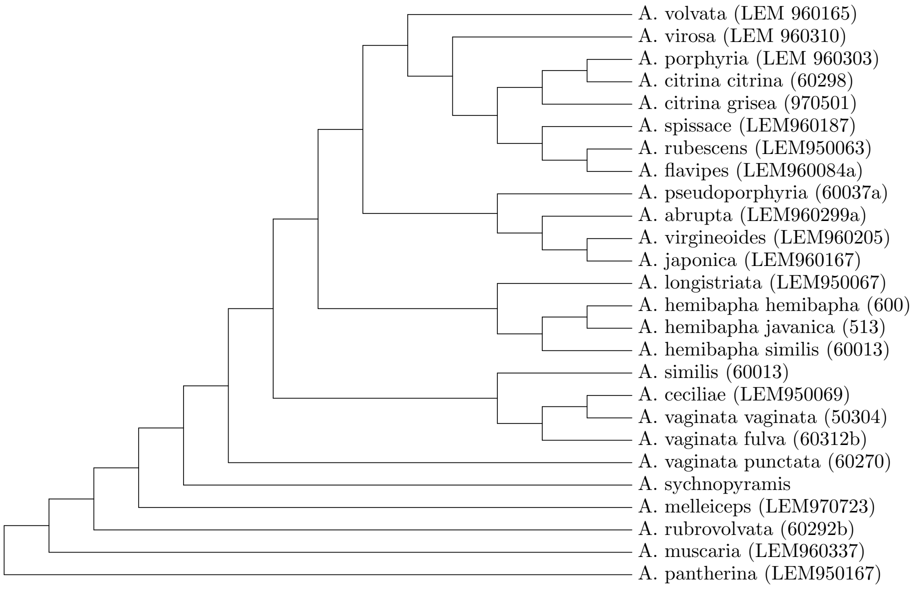

A. volvata (LEM 960165)
A. virosa (LEM 960310)
A. porphyria (LEM 960303)
A. citrina citrina (60298)
A. citrina grisea (970501)
A. spissace (LEM960187)
A. rubescens (LEM950063)
A. flavipes (LEM960084a)
A. pseudoporphyria (60037a)
A. abrupta (LEM960299a)
A. virgineoides (LEM960205)
A. japonica (LEM960167)
A. longistriata (LEM950067)
A. hemibapha hemibapha (600)
A. hemibapha javanica (513)
A. hemibapha similis (60013)
A. similis (60013)
A. ceciliae (LEM950069)
A. vaginata vaginata (50304)
A. vaginata fulva (60312b)
A. vaginata punctata (60270)
A. sychnopyramis
A. melleiceps (LEM970723)
A. rubrovolvata (60292b)
A. muscaria (LEM960337)
A. pantherina (LEM950167)

Figure 6.10: Optimal tree found for a set of 26 *Amanita* species taken from [155] (numbers refer to sequence identification as used in [155]).

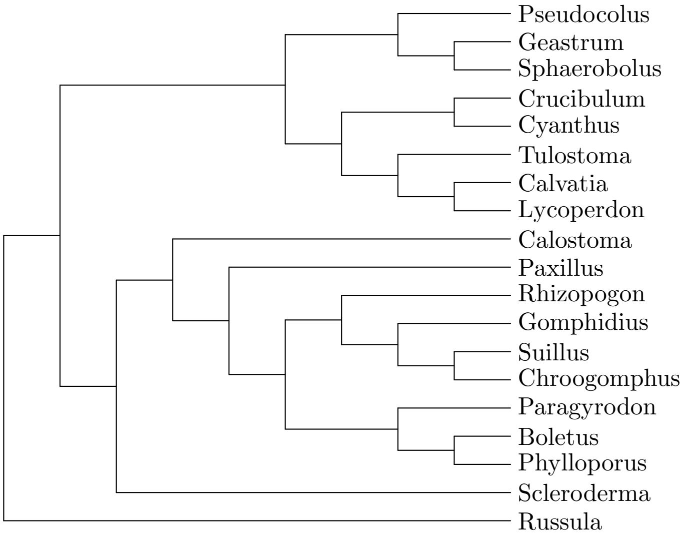


Figure 6.11: Optimal tree found for representatives of 19 Basidiomycetes genera, original data were taken from [102].

The tree agrees in most parts with the tree published in [155]; an exchange of genera *Scleroderma* and *Calostoma* even better supports the hypothesis of Hughey *et al.* [102] that *Calostoma* is closely grouped to the members of the familiy *Boletales* (which do not include *Scleroderma*) in spite of an “extreme morphological divergence” [102]. Moreover, it seems reasonable that in our tree, in contrast to the tree in [102], the *Paxillus* genus (belonging to *Boletales*) is also grouped with the other *Boletales* genera. The other datasets presented in [102] contained even less taxa and, thus, could also be analyzed with our method within minutes.

In summary, our experiments showed that our method allows to derive interesting results for real data with a limited number of taxa. The performance of our approach was highly dependent on the quality of the quartets. In our examples, we analyzed up to 26 taxa and datasets and it seems reasonable to solve up to 30 taxa for quartets of good quality. However, we also encountered datasets with only 15 taxa which turned out to be unsolvable for our approach within reasonable time, i.e. over night, due to the poor quality of the inferred quartets.

6.7 Conclusion and Open Questions

We showed that Minimum Quartet Inconsistency can be solved in worst-case $O(4^k n + n^4)$ time, meaning that the problem is fixed-parameter tractable for parameter k denoting the number of faulty quartet topologies. Several ideas for tuning the algorithm showed that its practical performance is much better than the theoretical bound given above (in particular, concerning the size of the search tree, 4^k). This is clearly expressed by our experimental results in Section 6.6. The combination of heuristic methods, inferring the strongly supported edges of a tree, and exact algorithms seems to result in trees of high quality and, therefore, deserves further investigation. This combination is a good trade-off between the heuristics, which are fast but result in only partially resolved trees, and the exact algorithms, which result in completely resolved trees but have exponential running times. We conclude with questions that remain open:

1. Extend the experimental investigation of the fixed-parameter MQI algorithm to weighted quartet topologies (to which our algorithm can be generalized to, see [88, 89] for more details).
2. Are there better heuristic strategies reducing the size of the search tree, e.g, lower bounds on the number (or the total weight) of quartets that we have to alter in order to find a solution. Such bounds could rapidly improve the running time of our algorithm.
3. Is there a practically relevant reduction to a problem kernel for MQI?
4. Are there other parameters than the number of erroneous quartet topologies that also allow fixed-parameter algorithms? Can we, in this way, find fixed-parameter algorithms for the Sparse MQI problem?
5. For every set of m quartet topologies, there is a binary tree inducing at least $m/3$ of the given quartet topologies. Therefore, is it fixed-parameter

tractable with respect to parameter k to determine whether we can satisfy $(m/3) + k$ quartets for a given complete set of m quartet topologies (this question was posed by Chor [46])?

6. Since MQI can be solved in polynomial time for $k < (n-3)/2$ [25], we can ask—in the "spirit of parameterizing above guaranteed values" [135]: Is it fixed-parameter tractable to find a tree that violates at most $(n-3)/2+k$, $k \geq 0$, quartet topologies?

Chapter 7

Consenus of Gene Orderings

With breakpoint distance, the genome rearrangement field delivered one of the currently most popular measures for comparing genomes of related species on the level of their gene orders. Here, genomes of species are represented as signed orderings of elements which represent the genes. Differences in the gene order are represented as breakpoints, i.e., pairs of elements which are adjacent in one ordering but not in the other. Then, BREAKPOINT MEDIAN is a core problem:

> BREAKPOINT MEDIAN
> **Input:** Signed orderings $\pi_1, \pi_2, \ldots, \pi_k$ on n elements and a non-negative integer d.
> **Question:** Is there a signed ordering π with $\sum_{i=1}^{k} d_{bp}(\pi_i, \pi) \leq d$?

Herein, $d_{bp}(\pi_i, \pi)$ denotes the *breakpoint distance* between orderings π_i and π, see Section 7.1 for definitions. The main application of BREAKPOINT MEDIAN lies in the recently intensively studied field of breakpoint phylogenies [51, 52, 138, 140, 141, 142] where the goal is to compute phylogenetic trees according to the maximum parsimony criterion for a set of species based on their gene order information. Notably, in this context, Moret *et al.* [141] emphasize that "exact solutions are strongly preferred over approximate solutions," since suboptimal solutions can lead to very different evolutionary reconstructions.

What is currently known about BREAKPOINT MEDIAN is summarized as follows.

1. BREAKPOINT MEDIAN is NP-complete and remains so even in the case of only three input orderings [33, 161].

2. In the case of three input orderings, BREAKPOINT MEDIAN has a polynomial-time algorithm with approximation ratio 7/6 [165].

3. Sankoff and Blanchette [174] presented a mathematically unanalyzed greedy heuristic for Breakpoint Median using a search tree employing a branch-and-bound technique. This algorithm was re-used and improved with respect to its implementation by Moret *et al.* [142]. Both implementations focus on the case of three input orderings.

By way of contrast, we also employ a search tree method which deviates from Sankoff and Blanchette's approach in that it employs a significantly different branching strategy. Thus, we can present an algorithm solving Breakpoint Median in $O(2.15^d \cdot kn)$ time, which is practical (as demonstrated by our experiments) when d is not too large, a reasonable assumption in applications. Notably, with increasing k, the base of our exponential base becomes smaller and smaller. Observe that, because Breakpoint Median is already NP-complete for $k = 3$, in some way the parameterization with d is "enforced"—the problem is *fixed-parameter intractable* with respect to parameter k unless P = NP. Besides experimental investigations for our Breakpoint Median algorithm itself, we also use it to propose a new approach to compute breakpoint phylogenies, applying it to chloroplast gene order data in Campanulaceae [51]. This complements celebrated "fixed-parameter heuristics" [51, 52, 68] for computing breakpoint phylogenies. For our new approach, it is important that we can find optimal breakpoint medians also for $k > 3$.

7.1 Preliminaries

We start with introducing orderings as they are used to model genome rearrangements, e.g., in [33, 141, 165, 174]. By default, we restrict ourselves to genomes containing the same set of genes and not containing duplicated genes; to meet these restrictions, it is a common technique to preprocess input data by deleting genes which occur only in a subset of the genomes or which occur in more than one copy [51]. These restrictions will be addressed again at the end of this chapter, in Section 7.5. Further, we restrict our explanation to linear genomes but it is straightforward to extend our approach to circular genomes.

Given a set $G = \{1, \ldots, n\}$, an *ordering* π on G is a $1:1$ function $\pi : G \to G$. We require that every ordering is extended by two special elements s, marking the start, and t, marking the end, and write ordering π as $\langle\, s\ \pi(1)\ \pi(2)\ \ldots\ \pi(n)\ t\,\rangle$. We write G_s for $G \cup \{s\}$ (G_t and $G_{s,t}$, analogously). An ordering π is *signed* iff every $\pi(x)$, $x \in G$, is equipped with a sign $\{+, -\}$, denoting the "orientation" of the element: In this way, $\pi(x)$, $x \in G$, can be, for $y \in G$

- a "positive" element $+y$ (or, for sake of brevity, only y), having left-to-right orientation, or
- a "negative" element $-y$, having right-to-left orientation.

Note that a signed ordering contains either y or $-y$, but not both at the same time. The special elements s and t are always unsigned. We write $G^{\pm}$ for the set $\{-1, 1, -2, 2, \ldots, -n, n\}$ and $G^{\pm}_{s}$ for $G^{\pm} \cup \{s\}$ ($G^{\pm}_{t}$ and $G^{\pm}_{s,t}$ analogously).

Example. The signed ordering

$$\pi = < s \; +1 \; -3 \; -2 \; +4 \; t >$$

is the ordering where $\pi(1) = 1$, $\pi(2) = -3$, $\pi(3) = -2$, and $\pi(4) = 4$. □

We use $\mathit{succ}_{\pi}(x)$, for a signed ordering π and $x \in G_s$, to denote the *successor* of element x in π, which is defined with respect to x's direction: For an element $x \in G$ occurring positively in π, the successor is the element following x. An $x \in G$ occurring negatively, however, has "reverse" orientation; hence, from x's point of view, its successor is the "reverse version" of the element preceding x. For instance, in ordering π as given above, the successor of element 1 is –3. Element 2, however, occurs negatively in π and the element following 2 with respect to this orientation is 3. Formally, we define a successor as follows:

Definition 7.1.1. *For $x \in G^{\pm}$, the* **successor** *$\mathit{succ}_{\pi}(x)$ is given by $\mathit{succ}_{\pi}(x) = y$, $y \in G^{\pm}$, iff we can find $l \in G$ such that one of the following two conditions applies:*

1. *$\pi(l) = x$ and $\pi(l+1) = y$, or*
2. *$\pi(l) = -x$ and $\pi(l-1) = -y$.*

For the special cases that $x = s$ or that $y \in \{s, t\}$, we define $\mathit{succ}_{\pi}(s) := y$ if $\pi(1) = y$; for $x \in G^{\pm}$, $\mathit{succ}_{\pi}(x) := t$ if $\pi(n) = x$, and $\mathit{succ}_{\pi}(x) := s$ if $\pi(1) = -x$. The predecessor $\mathit{pred}_{\pi}(y)$ for $y \in G^{\pm}_{t}$ is defined analogously. Notably, for $x \in G^{\pm}$, this definition satisfies

$$\mathit{succ}_{\pi}(x) = -\mathit{pred}_{\pi}(-x). \tag{7.1}$$

Example (continued). For π as defined before, we give the tables of successors and predecessors (the predecessors corresponding, due to equation (7.1), the successors of the negative elements). In the last column, we indicate which of the two possibilities in the above definition of succ_{π} applies (since $\mathit{pred}_{\pi}(x) = -\mathit{succ}_{\pi}(-x)$, we can also give the corresponding cases for the predecessor table).

element x	$\mathit{succ}_{\pi}(x)$	
s	1	–
1	–3	(1)
2	3	(2)
3	–1	(2)
4	t	–

element x	$\mathit{pred}_{\pi}(x)$	
1	s	–
2	–4	(1)
3	2	(1)
4	–2	(2)
t	4	–

For instance, $succ(2) = 3$ since (case (2) applies) we find $l = 3$ with $\pi(l) = -2$ and $\pi(l-1) = -3$; $pred(2)$ is -4 since $pred(2) = -succ(-2)$, and (for $succ(-2)$ case (1) applies) we find $l = 3$ with $\pi(l) = -2$ and $\pi(l+1) = 4$. □

We use $succ^*_\pi(x)$ for $x \in G^\pm$ to denote the set of elements "reachable" in a signed ordering π by a series of subsequent "successor steps":

Definition 7.1.2. *For $x \in G^\pm_{s,t}$ and a signed ordering π, the set of* **reachable elements** *for x is given by*

$$succ^*_\pi(x) = \{\, y \in G^\pm_{s,t} \mid \text{there are } x_1, \ldots, x_r \in G^\pm \text{ for which the subsequent conditions 1, 2, 3 apply} \,\}$$

1. $x = x_1$,
2. *for* $l = 1, \ldots, r-1$*:* $succ_\pi(x_l) = x_{l+1}$*, and*
3. $y = x_r$.

Analogously, we define $pred^*_\pi(x)$.

Example (continued). Considering $\pi\ = <\ s\ +1\ -3\ -2\ +4\ \ t\ >$, we obtain:

x	$succ^*_\pi(x)$	x	$pred^*_\pi(x)$
s	$\{1,-3,-2,4,t\}$	1	$\{s\}$
1	$\{-3,-2,4,t\}$	2	$\{-4,t\}$
2	$\{3,-1,s\}$	3	$\{2,-4,t\}$
3	$\{-1,s\}$	4	$\{-2,-3,1,s\}$
4	$\{t\}$	t	$\{4,-2,-3,1,s\}$

□

Using the successor relation, we can also formally define breakpoints:

Definition 7.1.3. *Given two signed orderings π_1 and π_2, both over G, we call a pair (x, y), $x \in G^\pm_s$ and $y \in G^\pm_t$, a* **breakpoint** *of π_1 with respect to π_2, iff*

1. $x = s$ *or* $\pi_1(l) = x$ *for some* $l \in G$*, and*
2. $succ_{\pi_1}(x) = y$ *and* $succ_{\pi_2}(x) \neq y$.

Using the notion of breakpoints, we define the *breakpoint distance* d_{bp} between two signed orderings as follows:

Definition 7.1.4. *For two signed orderings π_1 and π_2, both over G, the* **breakpoint distance** *between π_1 and π_2 is given by*

$$d_{bp}(\pi_1, \pi_2) = \left| \{\, (x, y) \mid x, y \in G^\pm_{s,t},\ (x, y) \text{ is breakpoint of } \pi_1 \text{ w.r.t. } \pi_2 \} \right| .$$

Due to symmetry, $d_{bp}(\pi_1, \pi_2) = d_{bp}(\pi_2, \pi_1)$.

Example. Given orderings

$$\pi_1 \ = < \ s \ +1 \ -3 \ -2 \ +4 \ \ t \ >$$

and

$$\pi_2 \ = < \ s \ +1 \ +2 \ +3 \ +4 \ \ t \ >,$$

the breakpoint distance is $d_{bp}(\pi_1, \pi_2) = 2$: Considering the breakpoints of π_1 with respect to π_2, one breakpoint is $(1, -3)$, since $\pi_1(1) = 1$, $succ_{\pi_1}(1) = -3$, and $succ_{\pi_2}(1) = 2 \neq -3$; the other breakpoint is $(-2, 4)$, since $\pi_1(3) = -2$, $succ_{\pi_1}(-2) = 4$, and $succ_{\pi_2}(-2) = -1 \neq 4$. Aside from these two, there are no other breakpoints since $succ_{\pi_1}(s) = succ_{\pi_2}(s) = 1$, $succ_{\pi_1}(-3) = succ_{\pi_2}(-3) = -2$, and $succ_{\pi_1}(4) = succ_{\pi_2}(4) = t$. □

Now, we have all definitions necessary to define Breakpoint Median, the central problem of this paper. As input, it takes signed orderings $\pi_1, \pi_2, \ldots, \pi_k$. The input orderings do not necessarily contain the additional symbols s and t which are necessary for a correct definition of breakpoint distance; if not, these symbols are added implicitly by the algorithm. Then, Breakpoint Median is the question whether there a signed ordering π with $\sum_{i=1}^{k} d_{bp}(\pi_i, \pi) \leq d$. The term $\sum_{i=1}^{k} d_{bp}(\pi_i, \pi)$ will, subsequently, also be referred to as the *breakpoint score* of π.

Preprocessing the input instance. The following intuitive lemma from [33] gives us a way to simplify a given input instance by preprocessing:

Lemma 7.1.5. **[33]** *Given signed orderings* π_1, $\pi_2, \ldots, \pi_k$*, all on a set* G *of* n *elements, and elements* $x, y \in G^{\pm}_{s,t}$*, which are adjacent in* π_1, $\pi_2, \ldots, \pi_k$*, i.e.,* $succ_{\pi_r}(x) = y$ *for all* $r = 1, \ldots, k$*. Then* x *and* y *are also adjacent in an optimal breakpoint median* π*, i.e.,* $succ_{\pi}(x) = y$*.* □

Using Lemma 7.1.5, we can preprocess the instance by "contracting" elements adjacent in all input sequences. From a parameterized complexity point of view, this preprocessing can be interpreted as a kind of reduction to a problem kernel, where the original instance consisting of k orderings of n elements each is reduced to a new instance consisting of k orderings of at most $d+1$ elements each when also counting the elements s and t (still, of course, all orderings have the same number of elements). The reason is that, for the resulting input instance with solution ordering π, we have one breakpoint (x, y) with respect to at least one of the input orderings for every $x \in G^{\pm}$ and also for $x = s$. Therefore, we can assume that in the given set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$, for every element x, there are at least two orderings in which x has different successors and there are at least two orderings in which x has different predecessors.

Surprisingly, an optimal breakpoint median can have adjacencies that are not present in any of the input orderings [33].

Lemma 7.1.6. [33] *Given signed orderings* π_1, $\pi_2, \ldots, \pi_k$, *all on a set* G *of* n *elements, and an optimal breakpoint median* π. *Then there can be elements* $x, y \in G^{\pm}_{s,t}$ *with* $succ_{\pi}(x) = y$ *and* $succ_{\pi_r}(x) \neq y$ *for all* $r = 1, \ldots, k$.

Proof. We consider the following example taken from [33].

Given three orderings

$$\begin{aligned}\pi_1 &=< s \ \ +5 \ \ +6 \ \ +7 \ \ +4 \ \ +1 \ \ +2 \ \ +3 \ \ t >,\\ \pi_2 &=< s \ \ +1 \ \ +5 \ \ +6 \ \ +4 \ \ +2 \ \ +3 \ \ +7 \ \ t >,\\ \pi_3 &=< s \ \ +1 \ \ +2 \ \ +5 \ \ +4 \ \ +3 \ \ +6 \ \ +7 \ \ t >,\end{aligned}$$

then

$$\pi =< s \ \ +1 \ \ +2 \ \ +3 \ \ +4 \ \ +5 \ \ +6 \ \ +7 \ \ t >$$

is an optimal breakpoint median, since it induces 12 breakpoints and, using, e.g., exhaustive search, we can show that there is no breakpoint median inducing less than 12 breakpoints. However, in π neither the adjacency $succ_{\pi}(3) = 4$ nor the adjacency $succ_{\pi}(4) = 5$ is present in any of the three given orderings. The reason is that π realizes all adjacencies that are present in at least two input orderings. This outweighs the disadvantage of setting $succ_{\pi}(3) = 4$ and $succ_{\pi}(4) = 5$ since 4 has different successors in π_1, π_2, and π_3 and the same is true for the predecessors of 4. □

Notably, in the example shown in the proof of Lemma 7.1.6, there exists also a breakpoint median $\pi = \pi_1$ inducing 12 breakpoints while containing only adjacencies which are present in at least one of the input orderings. In general, however, there are also examples where the optimal breakpoint necessarily contains adjacencies which occur in none of the input orderings (this can be verified by testing all possible input orderings). Since these examples are more complex, we omit to give one here.

Lemma 7.1.6 and the discussion following Lemma 7.1.6 imply that, when searching an optimal breakpoint median, it is not sufficient to only consider adjacencies present in the input orderings.

7.2 A Fixed-Parameter Algorithm for Breakpoint Median

In this section, we present a bounded search tree algorithm that solves Breakpoint Median in $O(2.15^d \cdot kn)$ time.

In Subsection 7.2.1, we introduce some further notation used specifically in the description of the algorithm. The recursive search procedure which constitutes the core of the algorithm, is presented in Subsection 7.2.2; it computes an intermediate representation of the solution. How to translate this intermediate representation into one (or more) solutions for the given Breakpoint Median instance is described in Subsection 7.2.3. The correctness of the algorithm is discussed in Subsection 7.2.4. The running time analysis is treated in two steps: Subsection 7.2.5 gives a running time analysis for the case of three input orderings and, based on this, Subsection 7.2.6 generalizes this analysis to more than three input orderings.

7.2.1 Notation

The algorithm starts its search for a median ordering π with the set of *unconnected* elements G, i.e., no element is assigned a successor or a predecessor. For an element $x \in G$ for which we have not yet chosen a successor (or predecessor), we write $\mathit{succ}_\pi(x) = \emptyset$ (or $\mathit{pred}_\pi(x) = \emptyset$). We introduce a *link* between elements x and y, $x \in G^{\pm}_{s}$, $y \in G^{\pm}_{t}$, with $\mathit{succ}_\pi(x) = \emptyset$ and $\mathit{pred}_\pi(y) = \emptyset$, when we set $\mathit{succ}_\pi(x) := y$; implicitly, this also sets $\mathit{pred}_\pi(y) := x$. The algorithm searches a median by introducing link by link into π. As long as there are elements $x \in G^{\pm}_{s,t}$ with $\mathit{succ}_\pi(x) = \emptyset$ or $\mathit{pred}_\pi(x) = \emptyset$, we call the ordering π *partial*. Otherwise, we call it *complete*. A predicate complete(π) is used to test whether π is complete. A (partial) ordering obtained from a partial ordering π by setting $\mathit{succ}_\pi(x) := y$ is referred to by $\pi[\mathit{succ}(x) = y]$ (which also implies that $\mathit{pred}_\pi(y) = x$). Analogously, we use $\pi[\mathit{pred}(x) = y]$.

In its search for the median, the algorithm considers to set links that are also present in at least one of the input orderings. However, due to Lemma 7.1.6, it also has to consider to set links that are not present in the input orderings. In the latter case, e.g., if the algorithm decides to link $x \in G$ with an element which is not successor of x in any of the input orderings, the algorithm may defer the determination of the concrete successor element; this is indicated by setting $\mathit{succ}_\pi(x) :=\perp$ (analogously for $\mathit{pred}_\pi(x) :=\perp$). If $\mathit{succ}_\pi(x) =\perp$ or $\mathit{pred}_\pi(x) =\perp$, then x is referred to as an *isolated* element. Note that we call the ordering complete if we have chosen successor and predecessor values—including $\perp$—for all elements.

In this way, the table containing for all $x \in G$ the successor and predecessor entries is, in our algorithm, an intermediate representation of the solution ordering, which is constructed in a step-by-step fashion. Every ordering has a corresponding representation as a successor/predecessor table which does not contain $\emptyset$ and $\perp$ entries. But not every successor/predecessor table is *valid* meaning that it can be extended, by replacing $\emptyset$ and $\perp$ entries, to a table corresponding to an ordering. Therefore, we introduce, for sake of brevity, the predicate invalid(π) that tests whether the successor/predecessor table for π can be extended to a valid table. More precisely, invalid(π) tests π for two conditions which are, if at least one of them applies, the reason that the successor/predecessor table of π is not valid. Predicate invalid(π) evaluates to true if one of the following two conditions holds:

- there is some $x \in G$ with $x \in \mathit{succ}^*_\pi(x)$, i.e., there is a "cycle" in the successor relation, or
- there is $t \in \mathit{succ}^*_\pi(s)$, but $|\mathit{succ}^*_\pi(s)| < |G| + 2$, i.e., there is a "bypass" in the successor relation between the start and the end element.

Given a set of signed orderings Π, all on set G, and $x \in G^{\pm}_{s,t}$, we define $\mathit{succ}(\Pi, x)$ as the set of elements y for which $\mathit{succ}_\pi(x) = y$ for $\pi \in \Pi$. Analogously, we define $\mathit{pred}(\Pi, x)$. Further, we write $\#(\Pi, \mathit{succ}(x) = y)$ to denote the number of orderings $\pi \in \Pi$ in which $\mathit{succ}_\pi(x) = y$; $\#(\Pi, \mathit{pred}(x) = y)$ is defined analogously.

7.2.2 The Recursive Procedure

The recursive procedure of Algorithm BM is presented in Figure 7.1. It takes as input signed orderings $\pi_1, \ldots, \pi_k$ and a positive integer d, and reports, if existent, a median π for which $\sum_{i=1}^{k} d_{bp}(\pi_i, \pi) \leq d$. In the following, firstly, we specify how to compute a successor and predecessor table for such a π (which can contain $\perp$ entries); it is deferred to Subsection 7.2.3 to describe how to obtain a concrete ordering π (by setting links also for the isolated elements) from this table.

Description of the procedure. The recursive algorithm builds a search tree to construct π from initially unconnected elements; in one node of the search tree, it selects an element $x \in G^{\pm}_{s,t}$ with $\mathtt{succ}_\pi(x) = \emptyset$ (or $\mathtt{pred}_\pi(x) = \emptyset$). It decides on a set of possible successor (or predecessor) values and recursively considers these values by branching into one subcase for each successor (or predecessor) value in the set. In this search, we keep track of the number of induced breakpoints: The algorithm is started with a parameter d denoting the allowed number of breakpoints. It maintains a breakpoint counter Δd denoting the difference between d and the number of breakpoints that are already induced by the (partial) ordering π as it is constructed up to this point. When branching into one subcase while introducing a new link, Δd is decreased by the number of breakpoints caused by this new link. The recursion stops as soon as we introduced more breakpoints than were allowed, i.e., if $\Delta d < 0$. A solution is found when we complete the ordering with a non-negative Δd parameter.

Branching cases. As possible successors for an element x, we consider the successors of x in the input orderings. Having chosen an arbitrary $x \in G_s$ with $\mathit{succ}_\pi(x) = \emptyset$, we recursively consider the case of setting $\mathit{succ}_\pi(x) := y$ for every $y \in G^{\pm}_{s,t}$ with $\mathit{succ}_{\pi_i}(x) = y$ for some $i \in \{1, \ldots, k\}$. Considering these branching cases, however, is not sufficient. Due to Lemma 7.1.6, we also have to allow successors which do not occur in any of the input orderings; such a link causes k breakpoints. We handle this by one additional "isolation" case which sets $\mathit{succ}_\pi(x) :=\perp$ and, thus, defers the choice of a concrete successor to

```
Recursive procedure BM($\pi, \Delta d$)
Global: Set $\Pi$ of $k$ signed orderings $\pi_1, \dots, \pi_k$ over $G_{s,t}$
        and a non-negative real $d$.
Input: A partial ordering $\pi$ and an integer $\Delta d$.
Output: A complete ordering, if existent, that can be obtained by
        completing $\pi$ in a way such that at most $\Delta d$ new breakpoints
        are introduced.
Method:
(Case 0) Recursion ends.
    if ($\Delta d < 0$) then return;   /* $\pi$ causes more breakpoints than allowed. */
    if invalid($\pi$) then return;                     /* $\pi$ is not a valid ordering */
    if complete($\pi$) then report $\pi$;                          /* solution found */
(Case 1) Recursion continues.
    choose $x \in \{1, \dots, n\}$ with $succ_\pi(x) = \emptyset$ or $pred_\pi(x) = \emptyset$;
    if ($succ_\pi(x) = \emptyset$) then
        /* Try to link x with every successor in the input orderings: */
        for every $y \in succ(\Pi, x)$ do
            if ($pred_\pi(y) = \emptyset$) then
                newd := $\Delta d - (k - \#(\Pi, succ(x) = y))$;
                call BM($\pi[succ(x) = y]$, newd);
            end if
        end for
        /* Try to isolate x: */
        call BM($\pi[succ(x) =\perp], \Delta d - k/2$);
    else if ($pred_\pi(x) = \emptyset$) then
        /* Try to link x with every predecessor in the input orderings: */
        for every $y \in pred(\Pi, x)$ do
            if ($succ_\pi(y) = \emptyset$) then
                newd := $\Delta d - (k - \#(\Pi, pred(x) = y))$;
                call BM($\pi[pred(x) = y]$, newd);
            end if
        end for
        /* Try to isolate x: */
        call BM($\pi[pred(x) =\perp], \Delta d - k/2$);
    end if
```

Figure 7.1: **Recursive procedure of Algorithm** BM. Inputs are a Breakpoint Median instance consisting of a set of signed orderings $\Pi = \{\pi_1, \pi_2, \dots, \pi_k\}$, each over n elements, and a non-negative integer d. First, we perform a preprocessing according to Lemma 7.1.5: If the resulting instance has more than d many elements, we can reject the instance. Otherwise, we invoke the recursion with BM(π, d) for a partial permutation π containing only unconnected elements, i.e., $succ_\pi(x) = \emptyset$ and $pred_\pi(x) = \emptyset$ for all elements x.

the stage in which we translate the intermediate representation of π from the successor/predecessor table to a concrete ordering.

Adjusting the breakpoint counter. When choosing a successor which occurs in one of the input orderings, it is straightforward to adjust the breakpoint counter: By setting of $\mathit{succ}_\pi(x) := y$, we introduce $\#(\Pi, \mathit{succ}(x) = y)$ breakpoints and, therefore, decrease Δd by this number. In the remaining isolation case, we adjust the breakpoint counter as follows: By setting of $\mathit{succ}_\pi(x) :=\perp$, we assume that we will, when π is complete, finally link x with an element y with $\mathit{pred}_\pi(y) =\perp$, by setting $\mathit{succ}_\pi(x) := y$ where $\mathit{succ}_{\pi_i}(x) \neq y$ for every $i \in \{1, \ldots, k\}$. Therefore, this link will cause k breakpoints. However, since both elements are marked as isolated, this introduced link corresponds to two isolation subcases, one setting $\mathit{succ}_\pi(x) :=\perp$ and one setting $\mathit{pred}_\pi(y) :=\perp$. Therefore, to take these k breakpoints into account, we will decrease Δd by $k/2$ in each isolation subcase.

The recursive procedure is initially called with BM(π, d) for the trivial partial ordering π containing no successor or predecessor entry for any element and the breakpoint parameter d from the Breakpoint Median formulation. If a value for d is not known, it could be determined, e.g., using incremental search.

7.2.3 Interpreting the Successor and Predecessor Tables

The recursive procedure given in Subsection 7.2.2 specifies how to compute a successor and predecessor table, in the following referred to as "s/p-table," of a proposed solution π; this s/p-table may contain $\perp$ entries to indicate isolated elements. In the following, we show that we can easily output an actual ordering using the information stored in the table. This is accomplished in two successive steps to be presented in this section: Step (a) tests whether all information stored in the s/p-table is "consistent" and, if it is, Step (b) outputs the ordering. Before stating these steps explicitly, we give some further insight into the structure of the information stored in the s/p-table.

Lemma 7.2.1. *Given a* Breakpoint Median *instance and the s/p-table for a complete ordering generated by procedure* BM*, i.e., an s/p-table containing no $\emptyset$ entries, the number of $\perp$ entries is even.*

Proof. The reason is that the table has an even number of entries (since an element $x \in G$ has a successor and a predecessor entry, s has only a successor entry, and t has only a predecessor entry) and every new link between two elements adds exactly two entries. □

The idea of representing an ordering π as a s/p-table which also contains $\perp$ entries is to, in this way, identify blocks which can be rearranged without affecting the breakpoint score of π. We define these blocks as follows:

Definition 7.2.2. *Given a set of signed orderings* Π *and a signed ordering* π*, all over* $G^{\pm}_{s,t}$*, a sequence of elements* $x_1, \dots, x_r \in G^{\pm}_{s,t}$ *is called* **isolated block** b *of* π *if*

1. $succ_{\pi}(x_l) = x_{l+1}$ *for* $l = 1, \dots, r-1$,
2. $pred_{\pi}(x_1) = \perp$ *or* $x_1 = s$*, and*
3. $succ_{\pi}(x_r) = \perp$ *or* $x_r = t$*.*

Elements x_1 *and* x_r *are called endpoints of the isolated block and they are referred to as* $e_l(b)$ *and* $e_r(b)$*. An isolated block with* $x_1 = s$ *is referred to as* b_s*; an isolated block with* $x_r = t$ *is referred to as* b_t*.*

In the recursive procedure BM as given in Subsection 7.2.2, it may happen that we create two isolated blocks b_i and b_j which have endpoints $e_r(b_i)$ and $e_l(b_j)$, respectively, which are adjacent in one of the input orderings. In this case, however, it would be preferable to link these blocks by setting $succ_{\pi}(e_r(b_i)) := e_l(b_j)$. This solution, however, would be found in another branch of the search tree. Therefore, we can, in this case, waive the proposed solution π. We will see in the next subsection that, thereby, we do not loose any solution.

> **Step (a):** Given an ordering π, represented by its s/p-tables, with isolated blocks b_s, b_1, $b_2, \dots, b_q$, b_t; if there are blocks b_i and b_j among them (if $q \geq 1$, it is not allowed that both $b_i = b_s$ and $b_j = b_t$) such that
>
> - $succ_{\pi'}(e_r(b_i)) = e_l(b_j)$ for any $\pi' \in \Pi$
>
> then we return without output.

After Step (a), the isolated blocks have the intended property which is described in the following. To this end, we introduce rearrangements of these blocks. Given an isolated block b consisting of the sequence $x_1, \dots, x_r \in G^{\pm}_{s,t}$, we use $-b$ to denote the "reversal" of b, i.e., the sequence $-x_r, -x_{r-1} \dots, -x_1$, which is also an isolated block. If π consists of isolated blocks b_s, b_1, $b_2, \dots, b_q$, b_t, we use $(b_s, b_1, b_2, \dots, b_q, b_t)$ to denote the ordering π' that is obtained from π by setting $succ_{\pi'}(e_r(b_s)) := e_l(b_1)$, $succ_{\pi'}(e_r(b_1)) := e_l(b_2)$ etc. In the following lemma, we spell out that isolated blocks can be permuted and reversed without affecting the induced number of breakpoints.

Lemma 7.2.3. *Given a* Breakpoint Median *instance and an ordering* $\pi = (b_s, b_1, b_2, \dots, b_q, b_t)$ *with isolated blocks* b_s*,* b_1*,* $b_2, \dots, b_q$*,* b_t *such that the condition of Step (a) does not apply. Then,* π *causes exactly as many breakpoints as*

$$(b_s, b_1, b_2, \dots, b_{i-1}, b_{i+1}, b_i, b_{i+2}, \dots, b_q, b_t) \text{ for } i = 1, ..., q-1$$

and

$$(b_s, b_1, b_2, \ldots, b_{i-1}, -b_i, b_{i+1}, \ldots, b_q, b_t) \text{ for } i = 1, ..., q.$$

Proof. Every link between two isolated blocks induces exactly k breakpoints, no matter in which order or orientation the isolated blocks are arranged (except of the fact that b_s is required to be the first and b_t the last of the isolated blocks). □

Lemma 7.2.3 shows how to obtain a concrete solution from the s/p-table supplied by Procedure BM:

> **Step (b):** Report the median ordering, by outputting the isolated blocks, beginning with b_s, linking the remaining isolated blocks arbitrarily, while ending with b_t.

In this, way, the s/p-table is a representation for possibly many equally good solutions, among which the algorithm, without additional knowledge, cannot decide which one to prefer. Step (b) outputs an arbitrary solution from them. However, it would as well be possible to output all possible solutions (then, however, sacrificing the time bounds which will be shown in Subsections 7.2.5 and 7.2.6) or to compute compute a preferred solution due to additional restrictions.

The recursive procedure BM together with the postprocessing described in this subsection is subsequently referred to as Algorithm BM.

7.2.4 Correctness of the Algorithm

The two directions of the correctness proof are given by the following two lemmas:

Lemma 7.2.4. *For a solution π which is computed by Algorithm* BM *it holds that* $\sum_{i=1}^{k} d_H(\pi, \pi_i) \leq d$.

Proof. Algorithm BM maintains a breakpoint score counter Δd that is initialized by d, and it is decreased when two elements are linked or an element is made isolated. In the following, we explain that this "bookkeeping" of the value of Δd is correct. In a branching which links two elements, Δd is decreased exactly by the number of breakpoints that are caused by this new link. More subtle is a branching in which we turn an element x into an isolated element by setting, e.g., $succ_\pi(x) = \perp$. When reporting the solution, we will link x with another isolated element y with $pred_\pi(y) = \perp$ (which exists since the number of isolated elements in π must be even). On the one hand, the algorithm accounts

$k/2$ breakpoints when setting $\mathit{succ}_\pi(x) :=\perp$ and another $k/2$ breakpoints when setting $\mathit{pred}_\pi(y) =\perp$. On the other hand, with k input orderings this link causes exactly k breakpoints. Since a solution is reported only if $\Delta d \geq 0$, this shows that a solution supplied by the algorithm causes at most d breakpoints. □

Lemma 7.2.5. *If there is a solution* π *with* $\sum_{i=1}^{k} d_H(\pi, \pi_i) \leq d$ *then Algorithm* BM *finds one.*

Proof. Consider a Breakpoint Median instance and a solution ordering π. We will show that the algorithm finds either π or a solution that induces at most as many breakpoints as π. Consider elements $x, y \in G^{\pm}_{s,t}$ which are linked in π, e.g., $\mathit{succ}_\pi(x) = y$, and which are linked in the same way in at least one of the input orderings, i.e., $\mathit{succ}_{\pi_i}(x) = y$ for $i \in \{1, \dots, k\}$. Then, our algorithm explores linking x and y either when considering element x (trying all successors occurring in one of the input orderings) or element y (trying all predecessors occurring in one of the input orderings). Next, consider elements $x, y \in G^{\pm}_{s,t}$ which are linked in π, e.g., $\mathit{succ}_\pi(x) = y$, and which are not linked in the same way in any of the input orderings, i.e., $\mathit{succ}_{\pi_i}(x) \neq y$ for all $i = 1, \dots, k$. Algorithm BM covers this situation when setting $\mathit{succ}_\pi(x) :=\perp$ and $\mathit{pred}_\pi(y) :=\perp$. Although it is possible that the algorithm may, when reporting the solution, arrange the isolated blocks in a way such that the successor of x is an isolated element different from y, the obtained solution induces, by Lemma 7.2.3, exactly as many breakpoints as π. Summarizing, for *every* solution π of the given Breakpoint Median instance, our algorithm either finds π or a solution that can be obtained from π by a rearrangement of its isolated blocks. □

7.2.5 Running Time for $k = 3$ Orderings

The following estimation of Algorithm BM's running time is based on the analysis of its search tree size. By Lemma 7.1.5 and the subsequent comment given there, we conclude that every recursive call made in Algorithm BM reduces breakpoint counter Δd by at least one. Given an element x with $\mathit{succ}_\pi(x) = \emptyset$ (or $\mathit{pred}_\pi(x) = \emptyset$, resp.), in the case of three input orderings there are, in fact, two possible situations:

(1) Either x has the same successor y_1 in two of the input orderings, and successor $y_2 \neq y_1$ in the third input ordering,

(2) or x has pairwisely different successors y_1, y_2, and y_3 in the three input orderings.

In (1), we decrease Δd by 1 when we set $\mathit{succ}_\pi(x) := y_1$, we decrease Δd by 2 when we set $\mathit{succ}_\pi(x) := y_2$, and we decrease Δd by 3/2 when we set $\mathit{succ}_\pi(x) :=\perp$. This yields branching vector $(1, 2, 3/2)$ and the branching corresponds to branching number 2.15 (for background on branching vectors and branching numbers, refer to Section 4.5). In (2), we decrease Δd by 2 when we set $succ_\pi(x) := y_i$ for all $i = 1, 2, 3$, and we decrease Δd by 3/2 when we set $succ_\pi(x) := \perp$. This yields branching vector $(2, 2, 2, 3/2)$ and the branching corresponds to branching number 2.12. This gives an upper bound of $O(2.15^{\Delta d})$ on the search tree size. In every search tree node, we can, in linear time, test whether the ordering is invalid or completed, and select the branching subcases. This gives the following result.

Proposition 7.2.6. Breakpoint Median *for* $k = 3$*, i.e., three signed input orderings, can be solved in* $O(2.15^d \cdot n)$ *time.*

Note that the branch-and-bound technique based on lower bounds as introduced by Sankoff and Blanchette [174] can also be used in the framework of our algorithm to further improve its performance in practice. The key distinguishing point between the algorithm of Sankoff and Blanchette and the above one seems to be that above there is a special treatment of the case of isolated elements and isolated blocks, which is not considered as such by Sankoff and Blanchette. This is advantageous in some cases (namely, when the lower bound used in the branch-and-bound algorithm has little effect).

7.2.6 Running Time for More Than Three Orderings

Studying Breakpoint Median with $k > 3$, we observe, with growing k, an increasing number of possible branching situations in Algorithm BM. For instance, consider $k = 4$. If we choose an element x for branching which has successor y_1 in three of the four input orderings and which has successor y_2 in the remaining input ordering, then we decrease Δd by 1 when we set $succ_\pi(x) := y_1$, we decrease Δd by 3 when we set $succ_\pi(x) := y_2$, and we decrease Δd by 4/2 when we set $succ_\pi(x) := \perp$; this branching corresponds to branching vector $(1, 3, 2)$. The other branching possibilities of Algorithm BM for $k = 4$ are characterized by branching vectors $(3, 3, 3, 3, 2)$, $(3, 3, 2, 2, 2)$, and $(2, 2, 2)$. The branching vectors correspond to characteristic polynomials $p(z)$, e.g., the branching vector $(3, 1, 2)$ corresponds to $p(z) := 1 - z^3 - z - z^2$. The following lemma shows how to characterize the branchings of Algorithm BM by their polynomials. The branchings of BM that are possible for fixed k are referred to as k-*branchings.*

Lemma 7.2.7. *A* k*-branching corresponds to a characteristic polynomial of the form*

$$p(z) := 1 - \sum_{i=1}^{k-1} a_i z^{k-i} - z^{k/2},$$

where $a_1, \ldots, a_{k-1}$ *are non-negative integers with* $\sum_{i=1}^{k-1} a_i \cdot i = k$.

Proof. Consider a k-branching that selects an element x and branches on the possible successors of x. It creates a subcase for every distinct successor of x

that occurs in one of the k given orderings; additionally it creates the "isolation" subcase by setting $\mathit{succ}(x) :=\perp$. In general, we can have, for integers $1 \leq i \leq k-1$ and $0 \leq a_i \leq k$, a_i many pairwisely distinct successors y such that $\mathit{succ}(x) = y$ in exactly i input orderings. Obviously, with k input orderings, we have $\sum_{i=1}^{k-1} a_i \cdot i = k$. The branch in which we choose to set $\mathrm{succ}_\pi(x) := y$ in the solution π causes $k-i$ many breakpoints when $\mathit{succ}(x) = y$ in i input orderings. Therefore, we decrease Δd by $k-i$ in this branch. We have a_i branches of this kind. Summarizing over all branches, the k-branching is characterized by the characteristic polynomial $p(z) := 1 - \sum_{i=1}^{k-1} a_i z^{k-i} - z^{k/2}$ with $\sum_{i=1}^{k-1} a_i \cdot i = k$. □

In the remainder of this section, we show that the worst-case branching number of Algorithm BM becomes better with increasing k. To this end, we, firstly, show that the branching $(k-1, 1, k/2)$, characterized by the polynomial

$$p_k(z) := 1 - z^{k-1} - z - z^{k/2},$$

is the worst-case branching among all k-branchings (this yields Proposition 7.2.10). We refer to the branching number of $(k-1, 1, k/2)$ by c_k. Secondly, we prove that c_k decreases with increasing k (this yields Proposition 7.2.11).

In order to show Proposition 7.2.10, which is the harder part when compared to Proposition 7.2.11, we need the following two technical lemmas. Lemma 7.2.8 easily follows by function analysis using a computer algebra system such as Maple or Mathematica.

Lemma 7.2.8. *For all integers* $k \geq 3$ *and* $p_k(z) = 1 - z^{k-1} - z - z^{k/2}$*, it holds that*

$$p_k\left(\sqrt[k-2]{\frac{1}{k-1}}\right) < 0.$$

□

Lemma 7.2.9 describes the worst-case k-branching among those k-branchings having the same number of subcases.

Lemma 7.2.9. *A* k*-branching with* i *subcases,* $i \in \{2, \ldots, k-1\}$ *has the same or a better branching number than the particular* k*-branching with* i *subcases characterized by the polynomial* $1 - (i-1)z^{k-1} - z^{i-1} - z^{k/2}$ *corresponding to the branching vector*

$$(\underbrace{k-1, \ldots, k-1}_{i-2 \text{ times}}, i-1, k/2).$$

Proof. We can transform the branching vector $(k-1, \ldots, k-1, i-1, k/2)$ for the branching characterized by polynomial $1 - (i-1)z^{k-1} - z^{i-1} - z^{k/2}$ into the branching vector for an arbitrary k-branching with i subcases by a succession of steps of the following form: Each step takes two entries from the branching vector, which have values v_1 and v_2, resp., with $v_1 > v_2$, and decreases v_1 by one and increases v_2 by one. Lemma 8.5 in [120] shows that each of these steps improves the corresponding branching number or leaves it unchanged. □

For example, in the case of $k = 3$, we can conclude by Lemma 7.2.9 that the branching number for the branching vector $(2, 2, 2)$ has no higher branching number as the branching number for the branching vector $(3, 1, 2)$; the reason is that we can transform $(3, 1, 2)$ to $(2, 2, 2)$ by decreasing the value of the first entry by one while increasing the value of the second entry by one.

Proposition 7.2.10. *For every integer* $k \geq 3$*, the branching number of any* k*-branching of Algorithm* BM *is upperbounded by* c_k*.*

Proof. By Lemma 7.2.7, k-branchings are characterized by polynomials

$$p(z) := 1 - \sum_{i=1}^{k-1} a_i z^{k-i} - z^{k/2},$$

where $a_1, \ldots, a_{k-1}$ are non-negative integers with $\sum_{i=1}^{k-1} a_i \cdot i = k$. Note that the polynomials described by p have a single positive real root which is between 0 and 1 (see Section 4.5).

One particular k-branching is the one characterized by the polynomial

$$p_k(z) := 1 - z^{k-1} - z - z^{k/2}.$$

The branching number c_k is $1/\beta$ for the positive real root β of polynomial p_k; let q denote another polynomial of those polynomials described by p and let $c = 1/\gamma$ for the positive real root γ of q.

We show that $\beta \leq \gamma$ which implies that $c_k \geq c$. We assume that $\beta > \gamma$ and derive a contradiction. Setting $p_k(\beta) = q(\gamma)$, we can conclude that

$$\beta^{k-1} + \beta + \beta^{k/2} = \sum_{i=1}^{k-1} a_i \gamma^{k-i} + \gamma^{k/2} < \sum_{i=1}^{k-1} a_i \beta^{k-i} + \beta^{k/2}.$$

This can be simplified to

$$\beta^{k-1} + \beta < \sum_{i=1}^{k-1} a_i \beta^{k-i}. \tag{7.2}$$

We distinguish two cases.

Case 1 applies if

$$\sum_{i=1}^{k-1} a_i \beta^{k-i} \leq k\beta^{k-1}.$$

Together with inequality (7.2), this implies that $\beta^{k-1} + \beta \leq k\beta^{k-1}$ which can be simplified to $\beta \leq (k-1)\beta^{k-1}$ and further to $1/(k-1) \leq \beta^{k-2}$. On the one hand, this means that

$$\beta \geq \sqrt[k-2]{\frac{1}{k-1}}.$$

On the other hand, we derive by Lemma 7.2.8 that

$$\beta < \sqrt[k-2]{\frac{1}{k-1}},$$

a contradiction.

Case 2 applies if

$$\sum_{i=1}^{k-1} a_i \beta^{k-i} > k\beta^{k-1}. \tag{7.3}$$

By Lemma 7.2.9, the branching number of the branching characterized by polynomial q is upperbounded by the branching number of the branching characterized by polynomial $p_j(z) = 1 - jz^{k-1} - z^j - z^{k/2}$, for an appropriately chosen $j \in \{1, \dots, k-2\}$; more precisely, j is chosen such that $j+1$ denotes the number of subcases in the branching characterized by q. We conclude (by setting $\sum_{i=1}^{k-1} a_i \gamma^{k-i} = p_j(\delta)$ for the positive real root δ of $p_j(z)$ and, then, substituting δ by γ)

$$\sum_{i=1}^{k-1} a_i \gamma^{k-i} + \gamma^{k/2} \leq j\gamma^{k-1} + \gamma^j + \gamma^{k/2}. \tag{7.4}$$

Firstly, using inequality (7.4) and setting $p_k(\beta) = q(\gamma)$ yields

$$\beta^{k-1} + \beta + \beta^{k/2} \leq j\gamma^{k-1} + \gamma^j + \gamma^{k/2}$$

and, since we assume that $\beta > \gamma > 0$, also

$$\beta^{k-1} + \beta + \beta^{k/2} \leq j\beta^{k-1} + \beta^j + \beta^{k/2}.$$

This simplifies to $\beta^{k-1} + \beta < j\beta^{k-1} + \beta^j$ and further to

$$1 < (j-1)\beta^{k-2} + \beta^{j-1}. \tag{7.5}$$

Secondly, by inequality (7.4), we derive that $j\beta^{k-1} + \beta^j \geq \sum_{i=1}^{k-1} a_i \beta^{k-i}$. Together with inequality (7.3), this yields

$$j\beta^{k-1} + \beta^j > k\beta^{k-1},$$

which simplifies to $\beta^j > (k-j)\beta^{k-1}$ and further to $1/(k-j) > \beta^{k-j-1}$. We conclude that

$$\beta < \sqrt[k-j-1]{\frac{1}{k-j}}.$$

Substituting this into (7.5), we obtain $1 < f(k,j)$ for

$$f(k,j) := (j-1)\left(\frac{1}{k-j}\right)^{\frac{k-2}{k-j-1}} + \left(\frac{1}{k-j}\right)^{\frac{j-1}{k-j-1}}$$

and $j = 1, \dots, k-2$. Function analysis (employing a computer algebra system such as Maple or Mathematica), however, yields (since we find that $f(k,j) = 1$ only for $j = 1$, $f(k,j)$ is continuous, and, for fixed k, the derivation by j of f for $j = 1$ is negative) $f(k,j) \leq 1$ for $j \geq 1$, a contradiction. □

k	3	4	5	20	40	60	80	100
c_k	2.15	1.84	1.68	1.22	1.13	1.09	1.08	1.06

Table 7.1: Concrete values of c_k in Theorem 7.2.12 for increasing values of k.

Proposition 7.2.11. *For every integer* $k \geq 3$, $c_k > c_{k+1}$.

Proof. Let β_k be the smallest positive root of polynomial

$$p_k(z) = 1 - z^{k-1} - z - z^{k/2}.$$

Then, $c_k = 1/\beta_k$. Analogously, let β_{k+1} be the smallest positive zero of polynomial

$$p_{k+1}(z) = 1 - z^k - z - z^{(k+1)/2},$$

such that $c_{k+1} = 1/\beta_{k+1}$. Since $p_k(\beta_k) = 0$ and $p_{k+1}(\beta_{k+1}) = 0$, it follows that

$$(\beta_k)^{k-1} + \beta_k + (\beta_k)^{k/2} = (\beta_{k+1})^k + (\beta_{k+1}) + (\beta_{k+1})^{(k+1)/2}.$$

Assuming $\beta_k > \beta_{k+1}$, we derive a contradiction. With $0 < \beta_{k+1} < \beta_k < 1$, we have also

$$(\beta_{k+1})^k < (\beta_k)^{k-1} \text{ and } (\beta_{k+1})^{(k+1)/2} < (\beta_k)^{k/2}.$$

This yields

$$(\beta_{k+1})^k + \beta_{k+1} + (\beta_{k+1})^{(k+1)/2} < (\beta_k)^{k-1} + \beta_k + (\beta_k)^{k/2},$$

a contradiction. □

Theorem 7.2.12. Breakpoint Median *for* $k \geq 3$ *can be solved in* $O((c_k)^d \cdot kn)$ *time for* $c_k \leq 2.15$.

Proof. By Proposition 7.2.10, c_k is the worst-case branching number among all k-branchings of Algorithm BM. By Proposition 7.2.11, $c_3 > c_k$ for $k > 3$. By Proposition 7.2.6, $c_3 = 2.15$, which concludes the proof. □

In Theorem 7.2.12, we only give an upper bound for c_k. In fact, the value of c_k is decreasing as k increases. Table 7.1 list some c_k values for increasing k.

7.3 Experimental Evaluation on Synthetic Data

In this section, we report about experiments on synthetic Breakpoint Median instances. The implementation was done using GNU C++ version 3.0.4, and the running time was measured on a Sun Blade 100 machine with Sparc 2e processor (500 MHz) and 512 MB main memory under Solaris 5.8.

The experiments were done on synthetic Breakpoint Median instances. For given values of n, k, and d, we produced a dataset containing k orderings over n elements such that there exists a median having a breakpoint score of at most d as follows. We started with k pairwisely equal orderings, each containing n elements. In one step, we randomly decided between a “reversal,” i.e., the inversion of a contiguous segment, and a “transposition,” i.e., the movement of a contiguous segment to another position. Then, we performed this operation on a randomly selected segment in a randomly selected ordering. We maintained a counter for the potential breakpoints: for a reversal we counted two, for a transposition we counted three breakpoints, since this, in both cases, is exactly the breakpoint distance between the ordering before the operation and the ordering after the operation. In this way, we continued to rearrange the data, step by step, as long as the breakpoint counter was at most d. In general, especially with large numbers of generated breakpoints, we had less breakpoints than we counted, since not every operation really introduces *new* breakpoints; for “reasonable” values of n, k, and d, e.g., $d \leq (1/4)kn$, however, the number of breakpoints was close to d. For each set of values for n, k, and d, we measured the average performance on 25 different input instances.

Performance Tests

Table 7.2 shows the running time and the size of the generated search trees on datasets with k orderings, each having n elements and having a breakpoint median requiring approximately breakpoint score d, for different combinations of values for n, k, and d. For search tree algorithms, the search tree size provides a machine-independent performance measure since constant running time factors might be improved due to implementation issues or due to improved computer technology. We generated instances for growing “rearrangement factors,” i.e., growing ratios of breakpoint score d to the instance size nk. On the one hand, with growing rearrangement factor the accuracy and the uniqueness of the computed breakpoint medians decreases. On the other hand, Breakpoint Median instances with high rearrangement factors are possible to occur in applications for computing breakpoint phylogenies [142, 174] on highly rearranged input genomes, e.g., from bacteria data.

By the results shown in Table 7.2, we can make the following observations: For fixed values of n and k, growing values for d mean, as we expect from the running time bound, an growth of running time and search tree size for Algorithm BM. This growth exhibits an exponential nature for very small values of k, i.e., $k \leq 5$. For larger values of k, the search tree size remains comparatively small in practice; this effect of small search trees for larger values of k is currently not fully understood and the question for an explanation remains open here. Note, however, that $k = 3$ is the case in which the problem has been studied and the case arising in known applications for computing breakpoint phylogenies [142, 174]. Further, we note that for $n = 100$ instances with rel-

n	k	d	time	size	n	k	d	time	size
100	3	60	0.01	65	150	3	90	0.37	159
		90	0.61	350			135	0.28	1 293
		120	1.26	5 385			180	0.84	3 656
		150	2.44	19 531			225	3:29.54	938 850
		180	9.21	35 307			270	14:52.54	4 488 091
		240	7:53.36	1 776 547		4	120	0.04	91
	4	80	0.02	61			180	0.53	140
		120	0.04	92			240	0.11	290
		160	0.03	108			300	0.72	2 714
		200	0.09	255			360	24:30.93	5 245 368
		240	5.55	35 503		5	150	0.04	101
		320	3:10.33	1 147 504			225	0.06	117
	5	100	0.02	62			300	0.12	240
		150	0.33	85			375	0.18	364
		200	0.06	131			450	9.29	28 611
		250	0.84	223		10	300	0.20	129
		300	0.37	953			450	0.26	144
		400	26.72	129 179			600	0.37	147
	10	200	0.04	86			750	0.42	150
		300	0.12	97			900	0.53	150
		400	0.17	99	200	3	120	0.75	210
		500	0.17	100			180	5.26	21 558
		600	0.22	117			240	14.17	35 158
		800	0.34	129			300	141:47.27	25 734 624

Table 7.2: Performance of Algorithm BM on synthetically generated datasets containing k signed orderings on n elements, for several "rearrangement factors," i.e., several ratios of breakpoint score d to the instance size nk, namely for $d = d_f \cdot nk$ for $d_f = 0.2, 0.3, 0.4, 0.5, 0.6$ (for $n = 100$ also $d_f = 0.8$, for $n = 200$ only up to $d_f = 0.5$). By "time" we refer to the running time in "minutes:seconds" format (or only seconds if the time is less than one minute), by "size" we refer to the size of the generated search trees, i.e., their number of nodes. Each shown value is the average of measurements on 25 datasets.

atively small rearrangement factor pose no computational problems whereas, with growing value of n, the exponential growth of search tree size and running time can already be observed for small rearrangement factors. While values of n around 100 are encountered in current applications, e.g., [51, 139], it is to be expected that in the near future instances with larger values of n have to be solved and, then, also instances with small rearrangement factor can pose computational challenges.

For fixed values of k and d, increasing values of n actually mean a decrease in the running time since the value of d becomes smaller compared to the total size of the instance and, thus, the instances become "easier" since the lower bound applies more often. For example consider $k = 3$ and $d = 100$: the search tree size (running time) is 5700 (0.75 sec) for $n = 50$, 1915 (0.44 sec) for $n = 100$, and 434 (0.14 sec) for $n = 150$. An analogous observation applies also for fixed values of n and d, and growing values for k. Here we, in addition to the just mentioned reason (i.e., the lower bound applies more often if d is small compared to the instance size), we also expect smaller search trees from the results in Subsection 7.2.6, i.e., from the bound $(c_k)^d$ on the search tree size where the value of c_k decreases with growing values of k; for example, consider $n = 100$ and $d = 200$: the search tree size (running time) is 154817 (38.13 sec) for $k = 3$, 126 (0.05 sec) for $k = 5$, and 87 (0.04 sec) for $k = 10$.

Notably, the results shown here include the resources needed to determine the optimal parameter value by testing for increasing values of d, starting with $d = 0$, whether d allows for a solution. This search for the optimal parameter value will be discussed in more detail below.

Comparison with Branch-and-Bound Heuristic

Instances which have small values of d compared to the total input size nk, e.g., $d \leq 0.2 \cdot nk$, are easy to solve. This accounts to the fact that Algorithm BM additionally employs the bounds as they were proposed in the branch-and-bound heuristic by Sankoff and Blanchette [174] and as they are also used in the GRAPPA software [142]. This branch-and-bound heuristic works particularly well in the case of few rearrangements. To compare the search tree generated by Algorithm BM to the search trees generated by the branch-and-bound heuristic, we reimplemented the branch-and-bound heuristic as it was proposed by [174]; the reimplementation was necessary since the heuristic is only used as a subprocedure and, thus, not directly accessible in the available software packages [142, 174]. Table 7.3 lists, for both approaches, the running times and the search tree sizes for various combinations of values for n and d, while mainly focusing on three input orderings. Notably, the case of three input orderings is the case for which the branch-and-bound heuristic was implemented [142, 174] and used as a subprocedure in the computation of breakpoint phylogenies. Therefore, most research on Breakpoint Median, so far, concentrated on $k = 3$, e.g., [165]. The results show that, for small values of d, the branch-and-bound heuristic works better since it quickly finds the optimal solution and the computed bounds work well. With growing value of d, however, the bounds apply less and less often. In these cases, the search trees generated by Algorithm BM are significantly smaller. For larger values of k, the employed lower bound on the breakpoint score works surprisingly well such that, since it is used in both approaches, the search trees are in both cases almost identical in size; as mentioned in the previous paragraph, the reason for this effect is currently not fully understood. When comparing the running times, we note that the time spent in one search tree node is higher in the branch-and-bound heuristic than for Algorithm BM. This may result from a straightforward and non-optimized implementation, e.g., when determining in the branch-and-bound heuristic the

			Algorithm BM		branch-and-bound	
n	k	d	time	size	time	size
50	3	30	0.40	25	0.90	22
		60	0.24	195	0.75	73
		90	0.91	650	0.83	997
		120	0.38	2 035	106.23	86 468
	10	100	0.02	45	0.07	43
		400	0.08	57	0.12	51
100	3	60	0.12	65	0.18	48
		120	1.26	5 385	12.12	3788
		180	9.21	35 307	1042.607	783 892
		240	7:53.36	1 776 547	not run	
	10	200	0.46	92	0.63	86
		800	0.32	101	0.972	100
150	3	90	0.37	159	1.27	167
		180	0.838	3 656	230.30	77 950
		270	892.54	4 488 091	not run	
	10	300	0.09	128	1.09	128
		1200	0.48	236	1.64	152

Table 7.3: Comparison between Algorithm BM and the branch-and-bound heuristic as proposed by Sankoff and Blanchette [174] on synthetically generated datasets containing k signed orderings on n elements, having a breakpoint median for breakpoint score d. “Time” refers to the running time in minutes:seconds (or only seconds if it is less than one minute), “size” to the size of the generated search trees. Each shown value is the average of measurements on 25 datasets. The entry “not run” means that at least one of the datasets was not finished within 12 h computing time.

pair of elements to branch on. For this reason, we propose to use the search tree size as the fair measure of comparison.

Summarizing, our results underline that Algorithm BM combines the advantages of the lower bound from the branch-and-bound heuristic with performance guarantees which are not given by the heuristic. Thus, Algorithm BM works comparatively well on instances which are “easy” for the heuristic but exhibits an better performance on “worst cases” on which the heuristic breaks down.

Determining the Optimal Distance Parameter

Besides the branching strategy, a significant difference between Algorithm BM and the branch-and-bound heuristic as proposed by Sankoff and Blanchette [174] lies in the way how the optimal distance parameter value is determined. The branch-and-bound heuristic searches, in a “greedy” way, an initial solution and tries to decrease the breakpoint score corresponding to this solution subsequently by exploring the search space of all possible ways to link the elements, pruning the search when the computed lower bound applies. In contrast, Algorithm BM starts the Algorithm for every $d = 0, 1, 2, \ldots$ until it finds a d value which allows a solution. This additional overhead is comparatively small: In average over 25 datasets with three input orderings, each having 100 elements, such that a breakpoint median with breakpoint score 100 exists, we measured a search tree size 14 for $d = 98$, size 474 for $d = 99$, and size 789 for $d = 100$. For all $d < 97$ the search tree is not invoked since it is smaller than the computed lower bound on the minimum number of breakpoints. This observation indicates that, in practice, the search for an optimal parameter value does not imply an additional d factor, but means only a constant running time factor.

Normalization of the Distance Parameter

As indicated in Subsection 7.2.6, the exponential base c_k corresponding to the branching vector $(k-1, 1, k/2)$ in the running time bound of Algorithm BM (Theorem 7.2.12) becomes better for increasing k. For instance, we have $c_3 = 2.15$, $c_4 = 1.84$, $c_5 = 1.68$, $c_{20} = 1.21$, $c_{50} = 1.11$, and $c_{100} = 1.06$, which tends to 1 when k goes to infinity. On the other hand, since Breakpoint Median sums up the distances over all k input orderings, the distance parameter d should necessarily also increase with increasing k. Hence, it would be natural to consider some kind of "normalized" parameter d', which does not increase with increasing k.

The first approach is to pose the question whether Breakpoint Median is fixed-parameter tractable with respect to parameter $d' := d/k$. This question would be appropriate when we assume that with every additional input ordering the total sum of breakpoints to be expected is increased by d for a given integer d. This assumption, however, may be inappropriate when processing real data since input orderings added to an input set are likely to share breakpoints with orderings which are already in the set. For this reason, a "lighter" normalized parameterization such as $d' := d/\sqrt{k}$ or $d' := d/\log(k)$ might make sense.

We pose it as an open question to determine whether Breakpoint Median is fixed-parameter tractable with respect to one of the mentioned parameters d'. Using the example given by Algorithm BM, we can, however, give some experimental evidence: In Table 7.4, we calculate, based on the running time bound proved in Theorem 7.2.12, the value of c' in the running time bound $O((c')^{d'} \cdot kn)$ when the parameter is chosen as d' (for $d' = d/k$, $d' = d/\log(k)$, and $d' = d/\sqrt{k}$) and k is increasing. If c' decreases with growing k, this may be taken as evidence that Breakpoint Median is fixed-parameter tractable with respect to d'. Our results give no evidence that Algorithm BM is fixed-parameter with respect to $d' = d/k$. However, with respect to $d' := d/\log(k)$ and $d' := d/\sqrt{k}$, the algorithm clearly exhibits fixed-parameter behavior.

k	$c' = c_k$	$c' = c_k^k$	$c' = c_k^{\log(k)}$	$c' = c_k^{\sqrt{k}}$
	$(d' = d)$	$(d' = d/k)$	$(d' = d/\log(k))$	$(d'=d/\sqrt(k))$
3	2.148	9.909	2.316	3.759
4	1.839	11.445	2.327	3.383
5	1.674	13.165	2.293	3.167
6	1.570	14.985	2.244	3.020
7	1.497	16.876	2.194	2.910
8	1.443	18.827	2.145	2.823
9	1.401	20.831	2.099	2.752
20	1.211	45.850	1.774	2.352
30	1.154	72.707	1.626	2.187
40	1.123	103.034	1.533	2.081
50	1.103	136.556	1.469	2.004
60	1.090	173.069	1.421	1.945
70	1.080	212.422	1.384	1.897
80	1.072	254.482	1.354	1.858
90	1.065	299.147	1.330	1.824
100	1.060	346.319	1.309	1.795

Table 7.4: Value of c' such that it can be shown that Algorithm BM has running time $O((c')^{d'} \cdot kn)$ when the parameter is chosen as d'. Here, c_k denotes constant base in the exponential upper bound on the search tree size for k input orderings, determined by $c_k = 1/\beta$ for the positive real root β of polynomial $p_k(z) = 1 - z^{k-1} - z - z^{k/2}$ (see Subsection 7.2.6).

One thing to additionally take into account here is that our estimates for the search tree sizes always are worst-case; in practice, our algorithm turned out to be much faster than could be expected from the theoretical (worst-case) running time analysis.

7.4 Application to Phylogeny Reconstruction

An application of Breakpoint Median is given in the reconstruction of breakpoint phylogenies, i.e., the problem of finding the most parsimonious phylogenetic tree with respect to breakpoint distance. In this section, we outline a new heuristic strategy computing the breakpoint phylogeny for a set of gene order data which uses the Breakpoint Median algorithm as a central subprocedure. In comparison with previous heuristics [138, 142, 174], our approach does not exhaustively explore the whole search space consisting of all binary trees with k leaves, but resolves the grouping of taxa, level by level, from a (hypothetical) root down to the leaves of the phylogenetic tree.

7.4.1 A Heuristic Computing Breakpoint Phylogenies

Given gene orderings $\Pi = \{\pi_1, \ldots, \pi_k\}$ for a set of k taxa, the algorithm starts by computing a root node, called *virtual root* of the tree (only necessary for the construction) and, then, the algorithm recursively divides the set of taxa into two subsets, associating new nodes with these subsets; the new nodes become child nodes of the virtual root and roots for the subtrees corresponding to the subsets. The recursion ends when the subsets have size one.

To label the virtual root node, our heuristic computes the breakpoint median π_r for the given set of gene orderings. To obtain a bipartition of the set of taxa, we consider all $2^{k-1} - 1$ distinct bipartitions of Π into non-empty sets Π_1 and Π_2. We compute the optimal breakpoint medians π_1 for $\Pi_1 \cup \{\pi_r\}$, inducing a score of d_1 breakpoints, and π_2 for $\Pi_2 \cup \{\pi_r\}$, inducing a score of d_2 breakpoints. Among all these bipartitions, we choose the ones with a minimum total number of induced breakpoints, i.e., the ones for which $d_1 + d_2$ is minimum. The breakpoint medians π_1 and π_2 corresponding to such an optimal bipartition are chosen to label the two children of the node labeled π_r.[1] We choose π_1 in this way (π_2 is analogous) such that π_1 is not only a good median with respect to the orderings in Π_1 but also takes into account the information on the orderings in Π_2 which is reflected in π_r. Now, if Π_1 (Π_2 is completely analogous) consists of two elements only, we create two children of the π_1 node, each child labeled with one element from Π_1. If Π_1 contains more than two elements, we process this set recursively, taking the π_1 node as the virtual root and Π_1 as the set of gene orderings, again considering all bipartitions of Π_1.

7.4.2 The Campanulaceae Dataset

Using our heuristic, we analyzed the dataset introduced by [51, 52], which contains signed gene order information for 13 chloroplast genomes of the plant family Campanulaceae. These data are referred to in a considerable number of papers (e.g., [39, 141, 142]) and, therefore, seem to be an appropriate challenge dataset.

In these data, every gene occurs exactly once in all orderings. Within 1 min 45 sec, we processed the dataset and the best tree we found caused 89 breakpoints, i.e., we found no tree causing less breakpoints. The topology of this tree is given in Figure 7.2; the displayed tree is not binary, since we contracted inner branches of score zero, i.e., whose endpoints are labeled by the same orderings. Due to the contracted inner branches the shown tree corresponds to 216 different binary topologies for each of which we can give a labeling of the inner nodes that yields the breakpoint score 89. These tree topologies are exactly the 216

[1] We optionally allow to investigate all optimal breakpoint medians that are found for $\Pi_1 \cup \{\pi_r\}$ and for $\Pi_2 \cup \{\pi_r\}$ and to run the described recursion for each combination of optimal medians separately. In the Campanulaceae dataset, however, these medians turned out to be unique in most cases.

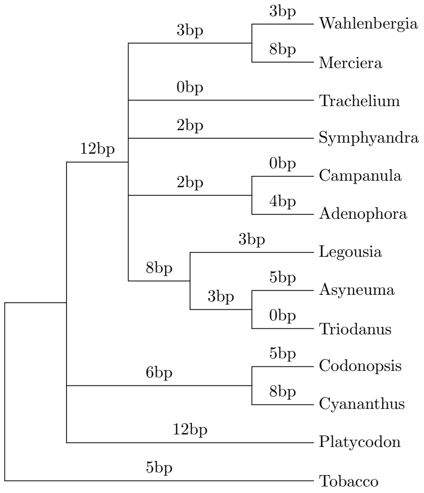


Figure 7.2: Tree with breakpoint score 89 that is found by our heuristic method for the Campanulaceae dataset (outgroup taxon tobacco).

different tree topologies that are also found by [141]. They used a (heuristically determined) constraint tree topology in order to explore the space of all binary tree topologies that are a refinement of this constraint tree. This way, they selected the 216 topologies from 10,395 considered binary tree topologies.

This result is preferable to a tree causing 96 breakpoints that is, according to [51], found by the "BPAnalysis" program [174] within a day, and it is competitive with the tree causing 89 breakpoints reported by [51, 52] which was found using the "GRAPPA" software.[2] To find the optimal trees by searching the whole space of all phylogenetic trees over 13 taxa, even when using the highly optimized "GRAPPA" software which includes additional bounding techniques, [140] still need "a few hours on a single workstation." This running time is drastically reduced when using a so-called "constraint tree," i.e., a not necessarily binary tree topology having its leaves in one-to-one correspondence with the taxa, as an additional problem input. Given a constraint tree, the "GRAPPA" software inspects only trees whose topology is a binary refinement of the constraint tree topology. Using a constraint tree (here the consensus of trees computed by fast heuristic methods like, e.g., neighbor joining) [141] reduced the size of the search space significantly (no exact time is given).

7.5 Conclusion and Open Questions

We conclude with some open questions and directions for further research:

[2] In [141], it is also mentioned that trees causing 84 breakpoints have been found; this, however, is according to one of the authors of [141] a typo. Using the "GRAPPA" software [142] we could only find the 216 mentioned topologies causing 89 breakpoints.

1. It would be desirable to extend Algorithm BM to the case where not all input orderings are over the same set of elements or when elements occur more than once within one ordering. These cases apply when genomes have a different set of genes or contain duplicated genes.

2. Reversal distance is, besides breakpoint distance, another popular distance measure on gene orderings. For a signed orderings π, a reversal replaces a segment e_i, $e_{i+1}, \ldots, e_{j-1}$, e_j in π by its inversion $-e_j$, $-e_{j-1}, \ldots, -e_{i+1}$, $-e_i$. Given two signed orderings π_1 and π_2, the reversal distance between them is the minimum number of reversals that are necessary to transform π_1 into π_2. A reversal median is defined in analogy to a breakpoint median, only replacing the breakpoint distance by reversal distance. Research on reversal distance concentrated, so far, on computing the reversal distance between two signed orderings [8], and on heuristics to compute reversal medians [39, 183]. In some cases, reversals might be preferable to breakpoint medians; e.g., Moret *et al.* [139] show that it has advantages to employ reversal medians instead of breakpoint medians in their "GRAPPA" software. However, there are no exact algorithms with non-trivial provable time bounds for computing reversal medians. In particular, it is an open question to develop a fixed-parameter algorithm with respect to the distance parameter.

3. Breakpoint Center is the problem in which, by way of contrast to Breakpoint Median, not the sum of distances is to be minimized, but the maximum distance of the solution ordering to each of the input distances. It is open whether Breakpoint Center is fixed-parameter tractable with respect to the distance parameter. With respect to the number of input orderings, we conjecture that, like Breakpoint Median, Breakpoint Center is already NP-hard for $k = 3$.

Chapter 8

Consensus of RNA Secondary Structures

Structure comparison of RNA and of protein sequences has become a central computational problem, bearing many challenging computer science questions. A sound and meaningful mathematical formalization of secondary structures is the one of *arc-annotated sequences*: For a sequence S an *arc annotation* A of S is a set of unordered pairs of positions in S. To compute similarities or for searching patterns in RNA structures, the notion of *arc-preserving subsequences* recently received considerable attention [64, 65, 67, 110, 130]. For two arc-annotated sequences S_1 and S_2, S_2 is an arc-preserving subsequence (aps) of S_1 iff one can delete letters (also called *bases*) from S_1—when deleting a letter at position i, then *all* arcs with endpoint i are also deleted—such that S_1 and S_2 are the same and also their arc annotations coincide. The advantage of this model is that it takes into account both sequence as well as structure information. In this way, we obtain a similarity value for two given arc-annotated input sequences S_1 and S_2 by computing the length of a *longest arc-preserving common subsequence (lapcs).* In the following, we focus on RNA structures; for related studies concerning algorithmic aspects of protein structure comparison using "contact maps," refer to [78, 123] (for more details on contact maps also see Section 8.4). The central problems of this chapter are given as follows:

Arc-Preserving Subsequence (APS)
Input: Arc-annotated sequences S_1 and S_2.
Question: Is S_2 an arc-preserving subsequence of S_1?

Longest Arc-Preserving Common Subsequence (LAPCS)
Input: Arc-annotated sequences S_1 and S_2, and non-negative integers k_1 and k_2.
Question: Is there a common arc-preserving subsequence of S_1 and S_2 that can be obtained by deleting at most k_1 bases from S_1 and

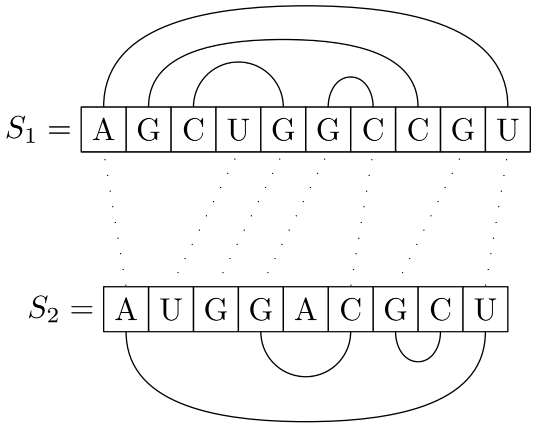


Figure 8.1: Example of a longest arc-preserving common subsequence for two arc-annotated input sequences S_1 and S_2. The common subsequence is obtained by deleting three bases in S_1 and two bases in S_2, a dotted line indicates that two bases are matched.

at most k_2 bases from S_2?

An example showing the longest arc-preserving common subsequence for two arc-annotated sequences is given in Fig. 8.1. Notably, the definition of LAPCS already introduces a particular parameterization of the problem, where it would also be possible to parameterize by the length l of the (longest) common subsequence; this parameterization was, e.g., used for longest common subsequence problems without arc annotations [28, 29]. In fact, the parameters l and the number of deletions k_1 and k_2 are *dual* parameters since $|S_1| = l + k_1$ and $|S_2| = l + k_2$, i.e., a larger value of l implies smaller values of k_1 and k_2. In applications, parameter l is small for unsimilar sequences—in this case the comparison of the sequences suffers from additional difficulties, e.g., since already two random arc-annotated sequences are likely to have a small common arc-preserving subsequence. For similar RNA structures, however, the presented way of choosing parameters is more meaningful.

Instead of allowing that the input sequences have an arbitrary arc structure, we, usually, can assume that certain restrictions apply. E.g., in a *nested* arc structure, one requires that no two arcs share an endpoint and that no two arcs cross each other; for example, both sequences shown in Fig. 8.1 have nested arc structure. According to Lin *et al.* [130] nested arc annotations are "generally thought of as the most important variant of the LAPCS problem," and nested arc-structures are also the central case in this chapter. Further, the term *plain* refers to sequences without arcs and *crossing* denotes arc structures where no two arcs share an endpoint. Finally, *unlimited* refers to a completely unrestricted arc structure. Using these terms, we can define various versions of LAPCS where LAPCS(TYPE1, TYPE2) refers to the case in which input sequence S_1 has an arc structure of TYPE1 and S_2 has an arc structure of TYPE2.

LAPCS(.,.)	UNLIM.	CROSS.	NESTED	CHAIN	PLAIN
UNLIMITED	NP-complete [64, 65]				
CROSSING		NP-complete [64, 65]			
NESTED			NP-compl. [130]	$O(nm^3)$ [110]	
PLAIN					$O(nm)$ [94]

Table 8.1: Survey of the computational complexity for different versions of LAPCS.

Analogously, we define APS(TYPE1, TYPE2).

What is currently known about the complexity of LAPCS and, in particular, of LAPCS(NESTED,NESTED) is summarized in the following. An overview of the computational complexity for different versions of LAPCS is given in Table 8.1.

1. LONGEST COMMON SUBSEQUENCE (LCS) for two sequences without arc annotations is solvable in quadratic time by dynamic programming [94]. For constant alphabet size, this can be improved to $O(n^2 \log\log n / \log n)$ (which is better than quadratic) [100] when both sequences have length n; for a survey of LCS results see, e.g., [94, 100]. LCS becomes NP-complete when allowing for an arbitrary number of input sequences
2. LAPCS(CROSSING,CROSSING) for two sequences is NP-complete [64, 65].
3. LAPCS(NESTED,NESTED) is NP-complete [130].
4. Special cases of LAPCS(NESTED,NESTED), which are also NP-complete, allow polynomial-time approximation schemes (PTAS's), e.g., when matches between two given input sequences are allowed only in a "local area" (of constant size) with respect to matching position numbers [130]; these PTAS's, however, suffer from high constant running time factors depending on the approximation ratio.
5. LAPCS(NESTED,NESTED) has a factor-2 approximation which needs quadratic time [110].

The APS problem as defined here has not been considered before. Table 8.2 gives an overview on the computational complexity for different versions of APS. Closely related problems can be found in the literature:

1. In the context of structured text databases, Kilpeläinen and Mannila already presented quadratic-time algorithms for the so-called ordered tree inclusion problem [117, 118] which is a strict special case of APS(NESTED,NESTED); for their special model the quadratic-time algorithm was later slightly improved in [44, 168].

APS(.,.)	UNLIM.	CROSSING	NESTED	CHAIN	PLAIN
UNLIMITED	NP-compl. [64, 65]				
CROSSING		NP-complete [64, 65]	NP-complete [93]		?
NESTED			$O(nm)^{(*)}$		

Table 8.2: Survey of computational complexity for different versions of APS. The result denoted by $(*)$ will be described in this chapter. The complexity of APS(CROSSING,PLAIN) remains open.

2. In the context of pattern matching for RNA structures, Vialette [195] stated a problem called PATTERN MATCHING OVER 2-INTERVAL SET which is related to an APS problem in which S_1 has unlimited arc structure; for a detailed comparison of the problems refer to [82]. Among others, Vialette shows NP-completeness for his problem when the arc structure of S_2 is crossing, he gives polynomial-time algorithms for cases where the arc structure of S_2 is a strict special case of nested, and he explicitly asks for the complexity of the problem when the arc-structure of S_2 is nested.

In this chapter, we discuss an exact, fixed-parameter search tree algorithm that solves the general LAPCS(NESTED,NESTED) problem in $O(3.31^{k_1+k_2} \cdot n)$ time where n is the maximum input sequence length. This gives an efficient algorithm in case of reasonably small values for k_1 and k_2 (the numbers of deletions allowed in S_1 and S_2, respectively), providing an *optimal* solution. The "Diplomarbeit" (masters thesis) of Guo [93], which investigates the parameterized complexity of different variants of LAPCS problems, also contains this algorithm for LAPCS(NESTED,NESTED) together with a running time analysis. Here, we focus on two special issues regarding the algorithm, while referring for many details to [93]. Firstly, we point out how the bounded search tree paradigm of parameterized complexity is applied in a novel way. The running time analysis of the search tree algorithm for LAPCS(NESTED,NESTED) required a new and non-standard technique (which constitutes our own contribution to this algorithm) in order to deal with a particular bottleneck case. Secondly, as a contribution not contained in [93], we complement the search tree algorithm by showing that special cases of the problem appearing in its traversal are solvable in quadratic time. More precisely, we show that APS(NESTED,NESTED) is solvable in $O(nm)$ time by dynamic programming. Notably developing such a dynamic programming algorithm over a non-trivial domain has "intrinsic difficulties" [77]. This dynamic programming algorithm implies a significant heuristic speed-up for our search tree algorithm. Complementing fixed-parameter algorithms with additional polynomial-time routines for preprocessing or handling special instances appearing in the search tree is one of the current research foci in parameterized complexity [68], and our combination of algorithms provides an excellent example. Moreover, the result for APS(NESTED,NESTED) was used to show that LAPCS(NESTED,NESTED) is fixed-parameter tractable with respect to the length l of the common subsequence (see the long version of [2]): Enumerate all length-l sequences with all possible nested arc annotations. For each of the enumerated sequences S, invoke the dynamic programming algorithm for APS(NESTED,NESTED) in order to check whether S is arc-preserving subsequence of both given sequences S_1 and S_2. Further, the dynamic programming technique allowed to show that PATTERN MATCHING OVER 2-INTERVAL SET restricted to $\{<, \sqsubset\}$ patterns [195] is solvable in quadratic time (for details refer to [82]), thus answering an open question by Vialette [195].

8.1 Preliminaries

The classical, NP-complete LONGEST COMMON SUBSEQUENCE problem is of central importance in computer science. In computational biology, it has been proposed as a technique to compute multiple alignments for a set of at least two input sequences [94] In this context, it has also been considered from the viewpoint of parameterized complexity; we refer to [28, 29, 58] for any details. Evans [64, 65] initiated classical and parameterized complexity studies for the more general case that the input sequences additionally carry an *arc structure* each, which is motivated by structure comparison problems in computational molecular biology. For a sequence S of length $|S| = n$, an *arc annotation* (or *arc set*) A of S is a set of unordered pairs of numbers from $\{1, 2, \dots, n\}$. Each pair (i, j) connects the two *bases* $S[i]$ and $S[j]$ at positions i and j in S by an arc. Since LAPCS is NP-complete even for two input sequences [64, 65], here and in the literature attention is focused on this case. Let S_1 and S_2 be two sequences with arc sets A_1 and A_2, respectively, and let i_1, i_2, j_1, j_2 be positive integers. If $S_1[i_1] = S_2[j_1]$, we refer to this as *base match* and if $S_1[i_1] = S_2[j_1]$, $S_1[i_2] = S_2[j_2]$, $(i_1, i_2) \in A_1$, and $(j_1, j_2) \in A_2$, we refer to this as *arc match*. If S_2 is a subsequence of S_1, this induces a one-to-one mapping $M = \{\langle i_r, j_r\rangle \mid 1 \leq r \leq |S_2|\}$ from a subset of $\{1, 2, \dots, |S_1|\}$ to $\{1, 2, \dots, |S_2|\}$ in which the matches are "order-preserving," i.e., for $\langle i, j\rangle$, $\langle i', j'\rangle \in M$ we have $i < i' \Leftrightarrow j < j'$. We say that S_2 is an *arc-preserving subsequence (aps)* of S_1 if the arcs induced by M are preserved, i.e., for all $\langle i_{r_1}, j_{r_1}\rangle, \langle i_{r_2}, j_{r_2}\rangle \in M$:

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

We say that S_1 and S_2 have an arc-preserving *common* subsequence if there is an arc-annotated sequence T which is arc-annotated subsequence of S_1 and S_2; if T is of maximum length with this property, it is a *longest arc-preserving common subsequence (lapcs)*.

We distinguish several types of arc structures for an arc-annotated sequence S, where *nested* arc structures are the most central case considered in this work. Formally, an arc set has *nested* arc structure if no two arcs share an endpoint and no two arcs cross each other, i.e., for all (i_l^1, i_r^1), $(i_l^2, i_r^2) \in A$ it holds that

$i_l^2 < i_l^1 < i_r^2$ iff $i_l^2 < i_r^1 < i_r^2$ (other types of arc structures have been introduced on page 138).

8.2 Fixed-Parameter Algorithm for LAPCS(nested,nested)

In this section, we describe Algorithm LAPCS which solves LAPCS(NESTED,NESTED) in $O(3.31^{k_1+k_2} \cdot n)$ time, where n is the maximum length of the input sequences. This algorithm is also contained in [93] to which we will also refer for the detailed running time analysis. Here, we stress only the integral parts of this analysis which give an example for a novel non-standard technique in the analysis of search trees. Thereby, we present with this algorithm a new way to realize the search tree paradigm in computational biology, which has not been seen in the preceding chapters.

Algorithm LAPCS is presented in 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, here, 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 $S_1 \approx_1 S_2$ that is satisfied when $S_1 = S_2$ after deleting at most one base in S_1 or at most one base in S_2. Note that we can check whether $S_1 \approx_1 S_2$ in linear time. The subsequence obtained from an arc-annotated sequence S by deleting $S[i]$ is denoted by $S - S[i]$. When branching into the case of a simplified sequence $S - S[i]$, the input for the recursive call is $S_{new} := S - S[i]$—hence, $|S_{new}| = |S| - 1$—and, therefore, $S_{new}[i] = S[i+1]$. We use $S[i,+]$ to denote the subsequence starting at $S[i]$ and extending to the end of S. 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. If base $S[i]$ is endpoint of an arc then we write, equivalently, that $S[i]$ is arc endpoint or that there is an arc from $S[i]$. We abbreviate $n_1 := |S_1|$ and $n_2 := |S_2|$.

Algorithm Description

Inputs are a LAPCS instance consisting of two arc-annotated sequences S_1 and S_2, and two integers k_1 and k_2. We process the sequences from left to right. Based on a case distinction depending on the bases at the currently first positions in S_1 and S_2, we decide how to continue recursively. For sake

```
Recursive procedure LAPCS($S_1, S_2, k_1, k_2$):
Input: Arc-annotated sequences $S_1$ and $S_2$ (with annotations $A_1$ and $A_2$,
       respectively), integers $k_1$ and $k_2$.
Output: Integer denoting the length of an lapcs of $S_1$ and $S_2$, if existent
        and if $k_1, k_2 \geq 0$, which can be obtained by deleting at most $k_1$
        bases in $S_1$ and at most $k_2$ bases in $S_2$. Return value $-1$ otherwise.
Method:
(Case 0) /* Recursion ends. */
    if $k_1 < 0$ or $k_2 < 0$ then return $-1$                    /* No solution found. */
    else if $|S_1| = 0$ and $|S_2| = 0$ then return $0$          /* Success! Solution found.*/
    else if $|S_1| = 0$ and $|S_2| > 0$ then                   /* One sequence done but not... */
        if $k_2 \geq |S_2|$, then return $0$, else return $-1$ end if;      /* ...the other. */
    else if $|S_1| > 0$ and $|S_2| = 0$ then                                    /* ditto */
        if $k_1 \geq |S_1|$, then return $0$, else return $-1$ end if
(Case 1) /* Non-matching bases $S_1[1]$ and $S_2[1]$. */
    else 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 $S_1[1]$ and $S_2[1]$. */
    else if $S_1[1] = S_2[1]$ then

        ... (for details refer to [2, 93])

    (2.5.3)
        if $(1, i) \in A_1$, $(1, j) \in A_2$, $S_1[i] = S_2[j]$, and
            neither $S_1[2, i-1] \approx_1 S_2[2, j-1]$ nor $S_1[i+1, n_1] \approx_1 S_2[j+1, n_2]$
        then return the maximum of the following four 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 $S_2[1]$. */
        { 1+LAPCS($(S_1 - S_1[i])[2, +], (S_2 - S_2[j])[2, +], k_1 - 1, k_2 - 1$)
        {     /* match $S_1[1]$ and $S_2[1]$, but do not match arcs $(1, i)$ and $(1, j)$;
        {     this implies the deletion of $S_1[i]$, $S_2[j]$, and the incident arcs. */
        { LAPCS_Case2.5.3.4($S_1, S_2, k_1, k_2$) (defined in Fig. 8.3) /* match arcs. */
        end if
    end if
```

Figure 8.2: Algorithm solving LAPCS(NESTED,NESTED). We provide, here, only an overview on its case distinction, omitting most details of Case (2) and only focusing onto the "bottleneck case," Case (2.5.3) (for better comparability, we keep the numbering of cases as it is used in [2, 93]).

of clarity, we, firstly, give in Figure 8.2 an overview in pseudocode of the algorithm which omits, for the sake of a better overview, many details which can be found in [2, 93]. The focus of this presentation will, in particular, be on the "bottleneck case" of the algorithm, namely Subcase (2.5.3). Although the algorithm as given reports only the length of an lapcs, it can easily be extended to compute the lapcs itself within the same running time. In our pseudocode, the **return** statement implies that the current procedure is immediately left with the specified value as output.

Case (0) deals with those cases in which no further recursive call is invoked: For example, if both parameters k_1 and k_2 are negative and, thus "used up," we did not find a solution in this branch of the search tree. Because of processing S_1 and S_2 from left to right, we make a case distinction depending on $S_1[1]$, $S_2[1]$. Case (1) covers the situation in which $S_1[1]$ and $S_2[1]$ do not match and Case (2) covers the situation in which $S_1[1]$ and $S_2[1]$ do match. Within Case (2), we further distinguish depending on possibly outgoing arcs from $S_1[1]$ and $S_2[1]$. We omit here and in Fig. 8.2, for the sake of a better overview, most of the case distinction which is made within Case (2); the omitted single cases are treated in an analogous way as Case (1) (for details see [2, 93]). Here, we right away point to the bottleneck case of our algorithm which is described as follows (we keep, for better comparability, the numbering of cases as used in [2, 93]):

In Case (2.5.3), we have the situation that there is an arc connecting $S_1[1]$ with $S_1[i]$, there is an arc connecting $S_2[1]$ with $S_1[j]$, and $S_1[1] = S_1[i]$ as well as $S_2[1] = S_2[j]$, i.e., there is the possibility to match the two arcs. Moreover, in (2.5.3), it is already clear that neither $S_1[2, i-1] \approx_1 S_2[2, j-1]$ nor $S_1[i+1, n_1] \approx_1 S_2[j+1, n_2]$ (both can be checked in linear time); in these cases, it is the best choice to match the arcs. In contrary, in (2.5.3), it is not yet clear that matching the arcs would lead to an optimal solution. Therefore, we recursively consider following possibilities: Not matching the arcs by breaking at least one of them (handled by the first three recursive calls in (2.5.3)) or matching the arcs (handled by the fourth recursive call in (2.5.3)); The fourth recursive call is described in Fig. 8.3. Here, we introduce the value l which 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 $S_1[2, i-1] \approx_1 S_2[2, j-1]$ and $S_1[i+1, n_1] \approx_1 S_2[j+1, n_2]$. Therefore, the deletion parameters k_1 and k_2 will be decreased by at least two in both recursive calls computing l' and l''; knowing this will be essential for the running time analysis. We invoke two calls for l' and l'' as displayed in Fig. 8.3.

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).

Recursive procedure LAPCS_**Case2.5.3.4**(S_1, S_2, k_1, k_2)**:**

Input: Arc-annotated sequences S_1 and S_2 such that Case (2.5.3) applies and positive integers k_1 and k_2.

Output: 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 while matching arc $(1, i)$ with arc $(1, j)$. The return value is -1 if no such subsequence exists.

Method:

```
    $l' := 0$;
    if $n_1 - i > n_2 - j$ then $l' :=$ LAPCS($S_1[2, i-1], S_2[2, j-1], k_1 - 2, k_2$)
    else if $n_1 - i < n_2 - j$ then $l' :=$ LAPCS($S_1[2, i-1], S_2[2, j-1], k_1, k_2 - 2$)
    else $l' :=$ LAPCS($S_1[2, i-1], S_2[2, j-1], k_1 - 1, k_2 - 1$)
    end if;
    $k'_{1,1} := i - 2 - l'$;                 /* number of deletions spent in $S_1[1, i]$ */
    $k'_{2,1} := j - 2 - l'$;                 /* number of deletions spent in $S_2[1, j]$ */
    $l'' :=$ LAPCS($S_1[i+1, n_1], S_2[j+1, n_2], k_1 - k'_{1,1}, k_2 - k'_{2,1}$);
    return $l' + l''$;
```

Figure 8.3: **Algorithm** LAPCS**, matching the arcs in (Case 2.5.3)**. Inputs are an LAPCS instance consisting of two arc-annotated sequences S_1 and S_2 with an arc $(1, i)$ in S_1 and with an arc $(1, j)$ in $S_2[1]$ such that these arcs match. This part of Algorithm LAPCS investigates the case that the arcs are matched.

Correctness of the Algorithm

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 and for those cases not discussed in this section, and, instead, focus on the most involved situation, Case (2.5.3).

In Case (2.5.3), we have three possibilities: Firstly, we can decide to not match $S_1[1]$ with $S_2[1]$; this is considered by the first two recursive calls of Case (2.5.3). Secondly, we can match $S_1[1]$ with $S_2[1]$ without matching arc $(1, i)$ with arc $(1, j)$; this is considered by the third recursive call of Case (2.5.3). In the latter case, we have to delete both $S_1[i]$ and $S_2[j]$ since, otherwise, we cannot maintain the arc-preserving property. Thirdly, we can match the endpoints of arc $(1, i)$ with the endpoints of arc $(1, j)$.

When matching the arcs in this way, 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 start the algorithm recursively on part 1 (to compute l') and, also 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 it will also require at least two deletions in part 2. Thus, when starting the algorithm on part 1, we can "spare" two of the $k_1 + k_2$ deletions for part 2, depending on part 2 (as outlined above). Having, thus, found an optimal solution of length l' for part 1, the number of allowed deletions remaining for part 2 is determined: we have, in part 1, already spent $k'_{1,1} := i - 2 - l'$ deletions in $S_1[2, i-1]$ and $k'_{2,1} := j - 2 - l'$ deletions in $S_2[2, j-1]$. Thus, there remain, for part 2, $k_1 - k'_{1,1}$ deletions for $S_1[i+1, n_1]$ and $k_2 - k'_{2,1}$ deletions for $S_2[j+1, n_2]$.

This discussion showed that, in Case (2.5.3), our case distinction covers all subcases in which we can find an optimal solution and, hence, Case (2.5.3) is correct.

Running Time Analysis

Here, we only point to the main ideas of the running time analysis of Algorithm LAPCS, a detailed proof can be found in [2, 93]. The analysis is mainly based on estimating the search tree size. This is done by considering the branching vectors corresponding to each case in the algorithm's case distinction. In a straightforward way (as shown in Subsection 4.5) we can show that all cases of the algorithm aside from Case (2.5.3) correspond to a branching number smaller than 3.31. Therefore, we focus now on Case (2.5.3), which branches into four subcases. Notably, the last of these branching subcases invokes not only one but two recursive calls, one to compute the value of l', the other to compute the value of l''. The main idea to deal with this case is to upperbound the sum of the sizes of these two search trees, as summarized in the following lemma.

Lemma 8.2.1. *Given two arc-annotated sequences* S_1 *and* S_2 *with non-negative integers* k_1 *and* k_2 *such that Case (2.5.3) in Algorithm* LAPCS *applies, the call* LAPCS_*Case2.5.3.4*(S_1, S_2, k_1, k_2) *generates a search tree of size upperbounded by* $3.31^{k_1+k_2-1}$.

The key to prove the lemma is that the search tree is traversed in breadth-first style. If Case (2.5.3) is called with parameter values k_1 and k_2, we can, in this way, derive that the heights of the two search trees computing l' and l'', respectively, add up to at most k_1+k_2. Moreover, we know that each of the two search trees is of height at least two. Using induction on the number of occurrences of Case (2.5.3) in these search trees, then, leads to the result summarized in Lemma 8.2.1. In the inductive step, we assume that the claim is true for all further occurrences of Case (2.5.3) in search trees computing l' and l''. Therefore, the recursive calls computing l' and l'' in these occurrences of Case (2.5.3) can be considered as if only one recursive call would be made, decreasing the parameter $k_1 + k_2$ by one. Thus, (Case 2.5.3) can be seen as corresponding to a branching vector $(1, 1, 2, 1)$ and to the branching number 3.31. For details on the proof refer to [2, 93].

The technique to conduct an accumulated analysis of two search trees computing l' and l'' by processing them in breadth-first manner seems novel in the area of search tree algorithms. In contrast to well known analysis techniques as outlined in Chapter 4, the two search trees are, here, not independent from each other, but the size of one of them depends on solutions found in the other. In this way, the search trees exchange information and only this allows us to bound, firstly, the sum of their heights and, secondly, the sum of their sizes. Using Lemma 8.2.1, we can prove the following theorem that summarizes the main result of this section.

Theorem 8.2.2. LAPCS(NESTED, NESTED) *for two sequences* S_1 *and* S_2 *with* $|S_1|, |S_2| \leq n$ *can be solved in* $O(3.31^{k_1+k_2} \cdot n)$ *time where* k_1 *and* k_2 *are the number of deletions needed in* S_1 *and* S_2*, respectively.* □

8.3 Dynamic Programming for APS(nested,nested)

Recognizing, in the traversal of the search tree, easy-to-process instances as, e.g., outlined in Section 4.4, can considerably speed up the performance of the search tree algorithm. In this Section, we show how to speed up the algorithm presented in Section 8.2. Procedure LAPCS as shown in Fig. 8.2 continues recursion until either a solution is found with non-negative values of parameters k_1 and k_2, or until *both* parameter values are 0. In contrary, we show that the recursion can actually be stopped already when only *one* parameter value is 0: The resulting instance is an APS(NESTED,NESTED) instance and it can be solved in quadratic time as we will show in the following.

The intrinsic difficulty in APS(NESTED,NESTED) is that, when considering an arc (i_l, i_r) in S_1 we have the following possibilities: (1) We can either be match (i_l, i_r) to an arc in S_2. (2) We can match only $S_1[i_l]$ (or only $S_1[i_r]$) to a base in S_2; then, $S_1[i_r]$ (or $S_1[i_l]$) cannot be matched to a base in S_2 due to the arc-preserving property. (3) Of course, we can also match neither $S_1[i_l]$ nor $S_1[i_r]$ to a base in S_2. To decide between those possibilities would lead to an exponential running time if care is not taken. Here, we present a dynamic programming approach that processes the bases in S_1 by their right endpoints, locally decides which matching possibility is the best, and stores this information in a dynamic programming table T. In the following, we will call $(i_l, i_r) \in A_1$ *aps-matching* arc for $(j_l, j_r) \in A_1$ iff $S_1[i_l] = S_2[j_l]$, $S_1[i_r] = S_2[j_r]$, and $S_2[j_l, j_r]$ is an aps of $S_1[i_l, i_r]$. We call $(i_l, i_r) \in A_1$ *innermost* aps-matching arc for $(j_l, j_r) \in A_2$ iff there is no $(i'_l, i'_r) \in A_1$ with $i_l < i'_l < i'_r < i_r$ which is aps-matching arc for (j_l, j_r). Using this terminology, two main principles of our algorithm are given as follows:

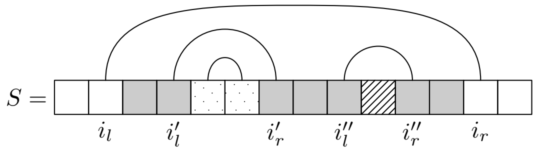


Figure 8.4: Illustrating the "I sets" corresponding to an arc-annotated sequence S: Bases in set I are white, bases in set $I^{(i_l,i_r)}$ are shaded, bases in set $I^{(i'_l,i'_r)}$ are dotted, and bases in set $I^{(i''_l,i''_r)}$ are patterned with lines.

- Bases in S_2 which are not endpoint of an arc or which are right endpoint of an arc are matched to the leftmost possible base in S_1 such that S_2 is an aps of S_1.
- Arcs in S_2 are matched to an innermost aps-matching arc in S_1.

If S_2 is an aps of S_1, then the algorithm computes *one* solution that respects both principles.

For the formulation of the algorithm, we introduce some new notation: For each arc $(i_l, i_r) \in A_1$, we define a set $I_1^{(i_l,i_r)}$ (analogously $I_2^{(j_l,j_r)}$ for $(j_l, j_r) \in A_2$) which contains the positions of the bases that are inside arc (i_l, i_r) but not inside any arcs that are inside (i_l, i_r),

$$I_1^{(i_l,i_r)} = \{\, i \mid i_l < i < i_r\} \setminus \bigcup_{\substack{(i'_l,i'_r)\in A_1 \\ \wedge\ i_l<i'_l<i'_r<i_r}} \{\, i' \mid i'_l < i' < i'_r\}.$$

If A_1 has a *nested* arc structure then the sets $I_1^{(i_l,i_r)}$ for different arcs are disjoint. We define I_1 (analogously I_2) as the set of endpoints of the outermost arcs in A_1 and of positions of all bases which are not inside any arcs in A_1. An example illustrating the set "I sets" for a sequence with nested arc-annotation is given in Fig. 8.4.

Algorithm Description

The dynamic programming table T contains entries for every arc in A_1 and every base in S_2. We refer to the table entries corresponding to $(i_l, i_r) \in A_1$ by $T(i_l, j)$, where j is an arbitrary position in S_2. Entry $T(i_l, j)$ is defined to contain the rightmost position $j' \geq j$ in S_2 such that $S_2[j, j']$ is an arc-preserving subsequence of $S_1[i_l, i_r]$ (or $j-1$ if no such j' exists). To fill table T, we process the arcs in A_1 by the order of their right endpoints and, thus, from inner to outer arcs. For every arc in A_1, we process the arcs in A_2 also by the order of their right endpoints, i.e., from inner to outer arcs. For an arc $(j_l, j_r) \in A_2$, we compute the table entries for the bases in $I_2^{(j_l,j_r)}$ before computing the table entry corresponding to j_l (no table entry corresponding to j_r is needed).

Recursive procedure maxaps($S_1[i_1, i_2], S_2[j_1, j_2]$):
Input: Arc-annotated sequences S_1 and S_2, positive integers i_1, i_2, with $1 \leq i_1, i_2 \leq |S_1|$, positive integers j_1, j_2, with $1 \leq j_1, j_2 \leq |S_2|$.
Output: The maximum j', $j_1 \leq j' \leq j_2$, such that $S_2[j_1, j']$ is an arc-preserving subsequence of $S_1[i_1, i_2]$, or $j_1 - 1$ if no such j' exists.
Method:

```
if $i_1 > i_2$ or $j_1 > j_2$ then
    return $j_1 - 1$;
else if $i_1 = i_2$ then
    if $S_1[i_1] = S_2[j_1]$ and $S_2[j_1]$ is not an arc endpoint then
        return $j_1$;
    else
        return $j_1 - 1$;
    end if
else if $i_1 < i_2$ and $j_1 = j_2$ then
    if $S_1[i_1] = S_2[j_1]$ and $S_2[j_1]$ is not an arc endpoint then
        return $j_1$;
    else if $S_1[i_1] \neq S_2[j_1]$ and $S_2[j_1]$ is not an arc endpoint then
        return maxaps($S_1[i_1 + 1, i_2], S_2[j_1, j_1]$);
    else /* if $S_2[j_1]$ is an arc endpoint */
        return $j_1 - 1$;
    end if
else if $i_1 < i_2$ and $j_1 < j_2$ and neither $S_1[i_1]$ nor $S_2[j_1]$ are arc endpoints then
    if $S_1[i_1] = S_2[j_1]$ then
        return maxaps($S_1[i_1 + 1, i_2], S_2[j_1 + 1, i_2]$);
    else
        return maxaps($S_1[i_1 + 1, i_2], S_2[j_1, i_2]$);
    end if
else if $i_1 < i_2$ and $j_1 < j_2$ and $S_2[j_1]$ is arc endpoint but $S_1[i_1]$ is not then
    return maxaps($S_1[i_1 + 1, i_2], S_2[j_1, j_2]$);
else /* $i_1 < i_2$ and $j_1 < j_2$ and $S_1[i_1]$ is the left endpoint of arc $(i_l, i_r)$ */
    return maxaps($S_1[i_r + 1, i_2], S_2[T[i_l, j_1] + 1, j_2]$);
end if
```

Figure 8.5: Recursive definition of maxaps. Note that positions i_1, i_2 and j_1, j_2 also have to be considered as arguments of maxaps and the notation maxaps($S_1[i_1, i_2], S_2[j_1, j_2]$) is used for better readability.

As an auxiliary function, the algorithm uses the recursive procedure maxaps which is called by maxaps($S_1[i_1, i_2], S_2[j_1, j_2]$) and which returns the maximum j', $j_1 \leq j' \leq j_2$, such that $S_2[j_1, j']$ is an arc-preserving subsequence of $S_1[i_1, i_2]$ (or $j_1 - 1$ if no such j' exists). Procedure maxaps is defined in Fig. 8.5. Procedure maxaps processes $S_1[i_1, i_2]$ and $S_2[j_1, j_2]$ from left to right and the basic principle is to match $S_1[i_1]$ with $S_2[j_1]$ whenever possible. To this end, an exhaustive case distinction is given depending on $S_1[i_1]$ and $S_2[j_1]$ and depending on possibly outgoing arcs from $S_1[i_1]$ and $S_2[j_1]$. The main idea of this procedure can be seen in the situation when $S_1[i_1]$ is the left endpoint of arc (i_l, i_r) (while $i_1 < i_2$ and $j_1 < j_2$). Then, we "jump" over arc (i_l, i_r) by

```
Procedure APS($S_1$, $S_2$):
Input: Arc-annotated sequences $S_1$ and $S_2$.
Output: Message stating whether $S_2$ is an aps of $S_1$.
Global Variable: array of int $T[n][m]$;
Method:
    /*********************** Phase 1 ****************************/
    for each $(i_l, i_r) \in A_1$ (ordered by their right endpoints) do
        for each $(j_l, j_r) \in A_2$ (ordered by their right endpoints) do
            for each $j \in I_2^{(j_l, j_r)}$ do
                $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps}(S_1[i_l, i_r - 1], S_2[j, j_r - 1]), \\ \text{maxaps}(S_1[i_l + 1, i_r], S_2[j, j_r - 1]) \end{array} \right\}$
            end for
            $T(i_l, j_l) := \left\{ \begin{array}{ll} j_r & \text{if } (i_l, i_r) \text{ is an innermost} \\ & \text{aps-matching arc for } (j_l, j_r), \\ \text{maxaps}(S_1[i_l + 1, i_r], S_2[j_l, m]) & \text{otherwise.} \end{array} \right.$
        end for
     end for
    /*********************** Phase 2 ****************************/
    for each $j \in I_2$ such that $S_2[j]$ is not an endpoint do
        for each $(i_l, i_r) \in A_1$ (ordered by their right endpoints) do
            $T(i_l, j) := \max \left\{ \begin{array}{l} \text{maxaps}(S_1[i_l + 1, i_r], S_2[j, m]), \\ \text{maxaps}(S_1[i_l, i_r - 1], S_2[j, m]) \end{array} \right\}$
        end for
    end for
    if ($\text{maxaps}(S_1[1, n], S_2[1, m]) = m$)
        then print '$S_2$ is an aps of $S_1$';
        else print '$S_2$ is not an aps of $S_1$';
    end if
```

Figure 8.6: Outline in pseudo-code of the algorithm that solves APS(NESTED,NESTED).

using the precomputed results stored in table T. In this way, we can advance the current position in S_1 from $i_1 = i_l$ to $i_r + 1$, i.e., to the position after the arc, while advancing the current position in S_2 from j_1 to $T[i_l, j_1] + 1$ since $T[i_l, j_1] + 1$ contains the maximum j' such that $S_2[j_1, j']$ is aps of $S_1[i_l, i_r]$. Observe that, when computing maxaps$(S_1[i_1, i_2], S_2[j_1, j_2])$, all entries in T corresponding to arcs in A_1 with both endpoints inside $S_1[i_1, i_2]$ have been computed before; this holds due to the order of processing the arcs in A_1 and A_2 (i.e., from inner to outer arcs). Therefore, the computation of the function maxaps is well-defined.

The complete algorithm solving APS(NESTED,NESTED) is outlined in Fig. 8.6. It is divided into two phases. The first phase processes S_1 and S_2 restricted to those bases which are endpoint of an arc or which are within an arc. The second phase, then, processes the bases in S_1 and S_2 which are outside every arc. In the following, we outline the first phase in more detail.

In the first phase, we process, in two nested loops, the arcs in A_1 from inner to outer arcs, in order to compute the table entries in T corresponding to these arcs. All table entries are computed using Procedure maxaps. For each arc in A_1, the arcs in A_2 are processed from inner to outer arcs. Then, for each pair $(i_l, i_r) \in A_1$ and $(j_l, j_r) \in A_2$, we, firstly, compute the table entries corresponding to bases in S_2 which are in $I^{(j_l,j_r)}$, i.e., which are in $S_2[j_l, j_r]$ but which are not within an arc having its endpoints $S_2[j_l, j_r]$. Then, secondly, we compute the table entry corresponding to $S_2[j_l]$. In this case, we decide whether we match arc (i_l, i_r) with (j_l, j_r). We decide to match the arcs only if (i_l, i_r) is the innermost aps-matching arc for (j_l, j_r). This test (which is omitted in Fig. 8.6 for the sake of simplicity) is done as follows: Arc $(i_l, i_r) \in A_1$ is an aps-matching arc for $(j_l, j_r) \in A_2$ iff maxaps$(S_1[i_l{+}1, i_r{-}1], S_2[j_l{+}1, j_r{-}1]) = j_r{-}1$, $S_1[i_l] = S_2[j_l]$, and $S_1[i_r] = S_2[j_r]$. To decide whether it is an *innermost* aps-matching arc, we recall that we process the arcs in A_1 in increasing order by their right endpoints. Therefore, we simply keep track of the so far last found arc $(i'_l, i'_r) \in A_1$ such that (j_l, j_r) is aps-matching arc for (i'_l, i'_r). If there was none so far or (i'_l, i'_r) is left of (i_l, i_r), i.e, $i'_l < i'_r < i_l < i_r$, then (i_l, i_r) is an innermost aps-matching arc for (j_l, j_r). This completes the description of Procedure APS.

Correctness of the Algorithm

For one direction of the correctness, we have to check that Algorithm APS returns with 'S_2 is an aps of S_1' only if S_2 is, in fact, an aps of S_1. To make this clear, we point out the following facts:

- A base $S_2[j]$ is only matched to a base $S_1[i]$ if $S_1[i] = S_2[j]$, and the order of bases is preserved, i.e., two bases $S_2[j]$ and $S_2[j']$ with $j < j'$ are matched to bases $S_1[i]$ and $S_1[i']$, respectively, only if $i < i'$ (see the definition of function maxaps in Fig. 8.5 and, if $S_2[j]$ or $S_2[j']$ is endpoint of an arc, the description of the test for aps-matching arcs in the preceding paragraph).
- The arc-preserving property is maintained:
  - If an arc $(i_l, i_r) \in A_1$ is not matched to an arc $(j_l, j_r) \in A_2$, i.e., we do not match both $S_1[i_l]$ with $S_2[j_l]$ and $S_1[i_r]$ with $S_2[j_r]$, then only one of $S_1[i_l]$ and $S_1[i_r]$ is matched to a base in S_2 but not the other (see the computation of table entry $T(i_l, j)$ for a position j in S_2 which is not endpoint of an arc in Procedure APS, Phase 1).
  - An arc $(j_l, j_r) \in A_2$ is matched only to an arc $(i_l, i_r) \in A_1$ (see the test for aps-matching arcs in the preceding paragraph).
- The matching of arcs respects the nested structure in S_1 and S_2, i.e., $(j_l, j_r), (j'_l, j'_r) \in A_2$ with $j_l < j'_l < j'_r < j_r$, are matched to $(i_l, i_r), (i'_l, i'_r) \in A_1$, respectively, such that $i_l < i'_l < i'_r < i_r$ (see the computation of table entries $T(i_l, j_l)$ in Procedure APS: (i_l, i_r) is matched with (j_l, j_r) only if $S_1[i_l + 1, i_r - 1]$ is an aps of $S_2[j_l + 1, j_r - 1]$).

For the reverse direction, we make sure that Algorithm APS returns with 'S_2 is an aps of S_1' if S_2 is, in fact, an aps of S_1: Observe, that a base $S_2[j]$ which is not endpoint of an arc is, by the definition of function maxaps, matched to an leftmost possible base in S_1. An arc $(j_l, j_r) \in A_2$ is matched to an innermost aps-matching arc in S_1. Therefore, if S_1 is an aps of S_2, then the algorithm finds the unique matching that respects these two principles.

Running Time Analysis

Essential for the running time analysis is to give a tight estimate of the running time of a call to Procedure maxaps as follows.

Lemma 8.3.1. *Let either* $I'_1 = I_1^{(i_l,i_r)}$ *for an arc* $(i_l, i_r) \in A_1$ *or* $I'_1 = I_1$*. In both cases, if* $i_1, i_2 \in I'_1$*, then a call of* maxaps$(S_1[i_1, i_2], S_2[j_1, j_2])$ *takes* $O(|I'_1|)$ *time.* □

Lemma 8.3.1 is based on the fact that, when processing an arc $(i_l, i_r) \in A_1$, we treat in Procedure maxaps all arcs within this arc, i.e., $(i'_l, i'_r) \in A_1$ with $i_l < i'_l < i'_r < i_r$, as "black boxes" since they have been processed before. This shows that, when processing the arcs from inner to outer arcs, we do only consider, for one arc, those parts of the sequence which have not been considered before. This is essential for showing the quadratic running time as follows. For a proof of Lemma 8.3.1, refer to [82]. The arcs in A_1 and the arcs in A_2 are processed in two nested loops and together with Lemma 8.3.1, this yields the following bound on the running time (for details refer to [82]).

Theorem 8.3.2. APS(NESTED, NESTED) *can be solved in* $O(nm)$ *time.* □

8.4 Conclusion and Open Questions

We conclude with open questions for future research:

1. It remains open to implement and evaluate the presented algorithms. An application for the algorithms is the computation of similarity values for RNA structures. An example of RNA sequences for which such structure comparisons may be of interest is given by the sequences of introns (see Chapter 2). On the one hand, we can distinguish certain types of introns, e.g., so-called group-I or group-II introns [32], where introns of one type have common structural properties. On the other hand, introns of one type exhibit large differences in their primary sequence. Therefore, algorithms are of interest which compute similarity values based both on sequence and structure. These similarity values can, then, be used to construct hypotheses on the evolutionary relationship of several introns of one type.

2. What is the parameterized complexity of LAPCS(CROSSING,NESTED) and LAPCS(CROSSING,CROSSING) when parameterized by the number of deletions? The problems are NP-complete and W[1]-hard when parameterized by the length of the common subsequence [64, 65].

3. Although nested arc structures are a usual case encountered in practice, there are also many examples of RNA with crossing but not nested arc structure. However, general crossing arc structures constitute a much more difficult case than the nested arc structures studied in this chapter. For example, APS(CROSSING, CROSSING) is NP-complete and, thus, there is no hope to extend our dynamic programming approach from Section 8.3 to this case. In the context of computing an "optimal" folding for a given RNA sequence, Rivas and Eddy [169] specify the notion of *solvable configurations* of arcs, which are a generalization of a nested arc structure but which are a proper subclass of crossing arc structures (see [169] for details). It is conceivable that these solvable configurations of arcs capture most tertiary RNA structures encountered in real world examples. Moreover, Rivas and Eddy showed that these structures are accessible to a dynamic programming algorithm; they gave a dynamic programming algorithm which computes, given a length-n RNA sequence, an optimal structure among all "solvable configuration" arc-structures under an energy minimization model. Their algorithm has $O(n^6)$ running time, subsequently improved to $O(n^5)$ [134]. Although the combinatorial question addressed by Rivas and Eddy is different from the questions studied in this chapter, a natural question is whether our algorithms in Section 8.2 and 8.3 can be extended to solvable configurations of arc structures.

4. *Contact maps* have been proposed as a model for protein structures and a *maximum contact map overlap* as a measure for the similarity of two structures [78, 123]. Contact maps are defined as arc-annotated sequences over a unary alphabet in which every base is endpoint of at least one arc. Given two arc-annotated sequences S_1 and S_2 with arc-annotations A_1 and A_2, a common subsequence T is defined by a one-to-one mapping $M = \{\langle i_r, j_r\rangle \mid 1 \leq r \leq |T|\}$ from a subset of $\{1, \ldots, |S_1|\}$ to a subset of $\{1, \ldots, |S_2|\}$ in which the matches are "order-preserving," i.e., for $\langle i, j\rangle$, $\langle i', j'\rangle \in M$ we have $i < i' \Leftrightarrow j < j'$. Then, a maximum contact map overlap is defined as a common subsequence that maximizes

$$\Big|\{(i_l, i_r) \in A_1 \mid \langle i_l, j_l\rangle, \langle i_r, j_r\rangle \in M \text{ and } (j_l, j_r) \in A_2\}\Big|.$$

CONTACT MAP OVERLAP, i.e., the question for the maximum contact map overlap of two given arc-annotated sequences, is NP-complete [78]. On the one hand, protein structures are more complicated than RNA structures since, in terms of arc-annotated sequences, one base can have an arc to more than one other base and the arc structures derived from real world examples are definitely crossing or even unlimited. On the other hand, CONTACT MAP OVERLAP is easier than the question for longest arc-preserving subsequences in the following ways: the alphabet is unary, every base is endpoint of an arc, and, therefore, partial arc matches are not possible, i.e., it is not possible to match only one endpoint of an arc in one sequence to a free base in the other sequence.

This raises questions as follows: Can a bounded search tree approach as discussed in Section 8.2 be applied to Contact Map Overlap, possibly restricting to certain structural patterns? Examples of structural patterns for which Contact Map Overlap is solvable in polynomial time are given in [78]. Further, can we solve APS problems for that model in polynomial time when the arc structure of the pattern belongs to a class of arc structures (to be determined) which captures consensus structures of protein families or protein domains, e.g., as they can be found in the PFAM database [14]?

Chapter 9

Contributions in Context

In this section, we give a selective summary of our results by putting them into the context of current research on parameterized complexity. We show commonalities and differences in the results that we achieved for different problems. Thus, we build bridges between the single chapters of this work. Moreover, we point out where our results help to find connections between the theory of parameterized complexity and other important areas in algorithm theory such as integer linear programming, approximation algorithms, and heuristics. We show how the examples studied in this work exhibit the potential but also the limitations of fixed-parameter approaches for computational biology problems. Therefore, this section is also intended to be a critical assessment, pointing to open questions which are raised by our work but remain open here.

9.1 Connections to Integer Linear Programming

Integer linear programming (as introduced in Chapter 1), is one the most central techniques in combinatorial optimization. For many combinatorial problems their formulation as an ILP has been discussed and software specialized for solving linear programs in general and ILP's in particular has been developed. For an overview on (integer) linear programming, refer to [119, 145, 176]. The connections between parameterized complexity and integer linear programming have, so far, been almost unexplored.

By example of Closest String, we introduced ILP's as a novel tool for showing a problem to be fixed-parameter tractable. Closest String is, given length-L strings $s_1, s_2, \ldots, s_k$, and a non-negative integer d, the question whether we can find a length-L string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$. In Subsection 5.1.3, we have shown that Closest String is fixed-parameter tractable with respect to the number k of input strings by formulating it as an ILP whose number of variables depends only on k. We concluded the fixed-parameter tractability of Closest String with an algorithm by H. W. Lenstra [125] which solves the ILP feasibility problem, i.e., decides whether there is any solution for the ILP. The key property allowing us to conclude fixed-parameter tractability of Closest String is that Lenstra's algorithm is fixed-parameter with respect to the number of variables. This important property was, to the best of our knowledge, not employed before in the development of fixed-parameter algorithms. Notably, we were not able to show this fixed-parameter tractability of Closest String using other means. Therefore, the ILP formulation was an essential tool to locate Closest String on the parameterized complexity map.

In the Closest String example the fixed-parameter approach suffers from two limiting factors: Firstly, the exponential growth of the number of variables is huge, more precisely the number of variables is given by $\mathsf{B}(\mathsf{k}) \cdot (\mathsf{k} - 1)$ where $\mathsf{B}(\mathsf{k})$ denotes the Bell number and is upperbounded by $\mathsf{k}!$. Secondly, Lenstra's algorithm is complex and has, to the best of our knowledge, not been implemented.

Nevertheless, it is an important first step in the design of fixed-parameter algorithms to decide whether at all the problem *is* fixed-parameter tractable or whether it turns out to be fixed-parameter intractable. There are a number of important problems which still resist such a classification. Therefore, tools are of interest which show fixed-parameter tractability even if they do not directly lead to efficient algorithms. The graph minor theory [58] and color coding [5] are two examples of well-known techniques for recognizing the fixed-parameter tractability of problems. Now, ILP's are a new and easy-to-use approach of this kind.

Moreover, the Closest String example also shows that designing an ILP in which the number of variables depends only on an input parameter can be of significance in practice despite of the mentioned disadvantages. We can solve the ILP using a linear programming package, e.g., the GNU GLPK library as we did in the Closest String example. These software tools, usually, solve ILP's in a heuristic way, e.g., using branch-and-bound techniques, and find exact solutions but do not give non-trivial guarantees on the worst-case running time; thus, these algorithms are, in particular, not fixed-parameter with respect to the number of ILP variables. However, the number of variables can make a difference for the performance of ILP solvers as we could exhibit in our experiments for Closest String instances, where we compared the performance of the GNU GLPK ILP solver on our new ILP formulation with an alternative ILP formulation of the problem (Subsection 5.1.4).

Our results point out an important connection between parameterized complexity and the theory of integer linear programming and, thus, contribute to a new and, so far, unexplored field of studies. We conjecture that there are more problems for which the ILP approach can be applied as for Closest String, and for which we can, in this way, gain new insights. However, further examples

	FPT	PTAS
VERTEX COVER	yes [43, 154] (Chapter 4)	"no," MAXSNP-hard [7]
CLOSEST STRING	yes (Section 5.1)	yes [129]
DISTINGUISHING SUBSTRING SELECTION	"no," W[1]-hard [83] (based on Sec. 5.2)	yes [56]

Table 9.1: Opposing, for some example problems, results concerning the problem complexity from parameterized complexity and from approximation theory.

have still to be found. Besides, it seems prosperous to investigate the parameterized complexity of further computationally hard problems in combinatorial optimization.

9.2 Connections to Approximation Algorithms

Approximation algorithms are a common technique to approach hard combinatorial problems. Among approximation algorithms presented for problems in computational biology, we can distinguish two lines of results of interest here: Firstly, there are polynomial-time constant-factor approximations, e.g., see [131, 162, 193] for recent examples; however, in applications, exact or almost exact solutions might be more desirable. Secondly, polynomial-time approximation schemes (PTAS's), as introduced in Chapter 1, became recently more and more popular, e.g., [56, 106, 129, 196].

In the following, we point out, based on this work, some observations on the connections between approximation schemes and parameterized complexity which are "deep and developing rapidly" [69]. First of all, there is no general rule stating that a PTAS implies fixed-parameter tractability or vice versa. To illustrate this, we give, in Table 9.1, example problems for each of which we cite a parameterized complexity and an approximation result. The parameterized complexity results are given in or based on this work. In this comparison of results, the parameter of the parameterized complexity result corresponds to the value of an optimal solution for the optimization criterion on the approximation side. Regarding VERTEX COVER, we have outlined a fixed-parameter algorithm with respect to the size of the vertex cover as parameter in Chapter 4. MAXSNP-hardness means that it is NP-hard to achieve a factor-$(1+\epsilon)$ approximation for a constant ϵ (see [7, 101] for more details), i.e., there is no PTAS unless NP = P. The MAXSNP-hardness result for VERTEX COVER refers to the approximation criterion of minimizing the size of the vertex cover. CLOSEST STRING is treated in Section 5.1, and the results mentioned here, both the fixed-parameter algorithm and the PTAS, refer to the distance parameter. Regarding the DISTINGUISHING SUBSTRING SELECTION problem, both the W[1]-hardness result and the PTAS refer to the "distance parameters" (for details see [83]). The mentioned W[1]-hardness result for DISTINGUISHING SUBSTRING SELECTION [83] has directly been developed from the results on CLOSEST SUBSTRING presented in Section 5.2.

The performance of an approximation scheme is determined by a trade-off between a good running time and the goodness of approximation: The better the desired approximation factor the more costly is the running time. Taking a closer look at the running times of proposed approximation schemes, however, suggests that they, often, are rather a tool for the structural classification of problems than a tool for designing practical algorithms. We give some examples for problems whose parameterized complexity has been discussed in this work:

- CLOSEST STRING (defined in Chapter 5, page 45): Li *et al.* [129] give a PTAS for CLOSEST STRING with the optimization criterion to minimize the distance parameter d such that there is a solution string with Hamming distance at most d to each given string. For k length-L input strings, they give a worst-case bound on the running time of their PTAS of $O((k \cdot L)^r \cdot k^{O(\log |\Sigma| \cdot r^2/\epsilon^2)})$ in order to achieve a factor-$(1+1/(2r-1)+\epsilon)$ approximation, for an integer r, $2 \leq r$, and real $\epsilon > 0$. For achieving a 25 percent error, we find the optimal trade-off between r and ϵ in this term for $r = 5$ and $\epsilon \approx 0.13$; then, the r^2/ϵ^2 term in the exponent of the running time estimation already evaluates to more than 1200. (For CLOSEST STRING, we gave a an exact fixed-parameter algorithm in Section 5.1.2.)

- MINIMUM QUARTET INCONSISTENCY (defined in Chapter 6, page 85): Jiang *et al.* [109] give a PTAS for MQI with the optimization criterion to maximize the number of quartets for which the given set Q of quartet topologies coincides with the set Q_T of quartet topologies induced by the solution tree. The two essential components of the PTAS given by Jiang *et al.* [109] are given as follows. Firstly, a number of binary trees with c leaves is enumerated where c is a constant depending only on the aimed approximation factor. Secondly, for each of these trees, a PTAS given in [6] is invoked which solves so-called t-smooth integer programs. To achieve a 25 percent approximation, the worst-case bound on the running time for this latter PTAS [6] already amounts to $n^{c \cdot 1024 \log n}$ for some constant c. Notably, in Chapter 6, we considered the "dual" version of the problem where the corresponding optimization criterion is to *minimize* the number of quartets for which Q and Q_T do *not* coincide. For this version, it is not even known whether there is an approximation with factor better than n^2 for n taxa [108]. (For MQI, we presented an exact fixed-parameter algorithm in Section 6.4.)

- LAPCS(NESTED,NESTED) (defined in Chapter 8, page 137): For restricted versions of LAPCS(NESTED,NESTED), Lin *et al.* [130] give PTAS's with the optimization criterion to maximize the length of the common subsequence. The restriction imposed by them is that a base in S_1 is allowed to match only with a base in S_2 from a certain range of constant size. For example, in c-DIAGONAL LAPCS, given a constant positive integer c, we are allowed to match a base in S_1 at position p only with a base in S_2 from positions $p-c$ to $p+c$. For c-DIAGONAL LAPCS(NESTED,NESTED), Lin *et al.* [130] provide a PTAS with $O(2^{(3ck-4)(ck-1)^2} c^9 k^9 (|S_1| + |S_2|))$ running time to achieve a factor-$(1-1/k)$ approximation. For a very small value of $c = 2$, to achieve a 25 percent error, the $2^{(3ck-4)(ck-1)^2}$ factor in the above term already evaluates to 2^{1372}. (For LAPCS(NESTED,NESTED) *without* the c-diagonal restriction, we presented a fixed-parameter algorithm in Chapter 8.)

Comparing the mentioned PTAS's with the algorithms presented in this work shows that fixed-parameter algorithms offer, in many cases, a more constructive approach to solve problems than approximation schemes do. For none of the mentioned PTAS's, it is stated along with the algorithm that they have been implemented, for most of our search tree algorithms, however, we also presented implementations and tests.

One reason that makes PTAS's infeasible already for moderate approximation ratio is the $n^{O(1/\epsilon)}$ running time, often with large constant factors hidden in the exponent. Therefore, Cesati and Trevisan [41] propose the concept of an *efficient* polynomial-time approximation scheme (EPTAS) with an $f(\epsilon) \cdot n^{O(1)}$ running time for arbitrary function f. From the PTAS's mentioned above, only the one for c-DIAGONAL LAPCS(NESTED,NESTED) [130] is an EPTAS. It is observed by Bazgan [16] and by Cesati and Trevisan [41] that a problem which has a PTAS but is W[1]-hard can not have an EPTAS unless FPT =W[1]. For example, a PTAS for the Distinguishing Substring Selection problem was presented in [56] and, for the same problem, a W[1]-hardness result, obtained by using our construction from Section 5.2, was given in [83]. Following the preceding discussion, the W[1]-hardness result implies that Distinguishing Substring Selection does not have an EPTAS unless FPT =W[1]. Therefore, our results from Section 5.2 enabled to show, for one of the first computational biology problems, a border between a known PTAS and a highly unlikely EPTAS.

Results concerning the parameterized complexity of problems can also provide structural results which are, so far, not provided by approximation theory. This is shown by the following example that we recall from Chapter 5. Both Closest String as well as Closest Substring are NP-complete and for each of them Lin *et al.* [130] present a PTAS. Thus, from the viewpoint of classical complexity (both problems are NP-complete) and from the viewpoint of approximation theory (both problems have a PTAS), both problems are "equal"; intuitively, however, Closest Substring is the harder problem since, compared to Closest String, it additionally involves the difficulty of choosing a substring in every input string. Our results presented in Chapter 5 clearly distinguish these two problems from the viewpoint of parameterized complexity. With respect to the number of input strings as parameter, we show that

Closest String is fixed-parameter tractable whereas Closest Substring remains W[1]-hard.

9.3 Connections to Heuristics

The combination of fixed-parameter algorithms with heuristics is an important issue of recent research in parameterized complexity [68, 69]. Here, we clearly distinguish between two common notions of heuristics. Firstly, heuristics denote strategies that do still allow to find the guaranteed optimal solution and improve the running time in practice but do not allow to improve the worst-case estimate on the running time. Secondly, heuristics denote strategies that allow for better estimates on the worst-case running time but do neither give any guarantee on the optimality of the solution nor guarantee a degree of approximation. In this section, we point out some examples taken from this work where we combine fixed-parameter algorithms with heuristics of both mentioned types. In many cases, heuristic strategies can drastically reduce the running time in practice and can, thus, be the key to the application of the algorithms while combining advantages of heuristics (improving the performance) and fixed-parameter algorithms (mathematical analysis and running time guarantees).

Regarding heuristics that still find guaranteed optimal solutions, consider the search tree algorithms for Closest String (Section 5.1) and for Minimum Quartet Inconsistency (MQI) (Chapter 6). In both examples, we showed that the measured running time of the algorithms remains far below the theoretical worst-case bounds. In both cases, this was achieved by heuristic improvements added to the algorithms. In the Closest String example, this includes easy observations, e.g., not to redo changes that are made in the course of the search tree traversal, and more complex strategies, e.g., our strategy to avoid multiple computations of the same solution (see Subsection 5.1.2.2 for details). In the Minimum Quartet Inconsistency example, a central idea was the detection of changes that have to be made in order to find a solution. This strategy works in a similar way as the computation of a problem kernel for 3-Hitting Set [153]; for MQI, it does, however, not yield a problem kernel but still works to speed up the algorithm.

A heuristic strategy of a different flavor was presented in the search tree algorithm for Longest Arc-Preserving Subsequence (LAPCS) that is given in Chapter 8. Here, we show a way to process special instances encountered in the traversal of the search tree in polynomial time. More precisely, we showed how to process instances where one of the two problem parameters already equals zero. These instances can be solved in quadratic time $O(|S_1| \cdot |S_2|)$, where S_1 and S_2 are the two given arc-annotated sequences, by dynamic programming. Asymptotically, this does not affect the size of the search tree. However, it does “cut off” whole branches of the search tree, and, thus, improves the performance of the algorithm drastically.

Concerning the second kind of heuristics, those for which we loose the guarantee of an optimal solution, we have shown a promising approach for MQI. We propose to use a heuristic preprocessing and to use the search tree algorithm on top of the results of the preprocessing. Specifically for MQI, the preprocessing infers edges of the tree to be constructed which are "strongly supported" by the input. A range of such preprocessing strategies has been proposed for MQI [23, 24, 25, 194, 107, 109] which differ in the way they decide whether edges are strongly supported. In this way, the output of the preprocessing is a tree that is not necessarily binary and, in the worst case, a star tree. The search tree algorithm, then, refines this tree to a binary tree as required by MQI. Roughly speaking, the more edges one of these preprocessing methods infers for a given MQI instance, the higher is the possibility that an edge inferred in this preprocessing does not belong to an optimal MQI solution. In Chapter 6, we chose one of the most conservative among the mentioned preprocessing methods, Q^* [24], for which it is quite unlikely to infer edges that lead to suboptimal results of the overall algorithm. The search tree algorithm, then, refines this tree to a binary tree. The search space to be explored is reduced drastically by the preprocessing step as we show in Section 6.6. Nevertheless, on real data, the found solutions are usually optimal. Thus, this combination can be seen as a trade-off between fast but error-prone heuristics and time-consuming exact algorithms. This approach seems prosperous also for other problems, e.g., by reducing the input size using heuristic data reduction techniques.

9.4 Choice of Parameters

The choice of parameters is a subtle point in the analysis of fixed-parameter algorithms. Often there is more than one natural parameterization. This allows several starting points for the development of fixed-parameter algorithms. Also, the parameterized analysis with respect to different parameters allows important insights into the combinatorial hardness of a problem. This section highlights several issues in the choice of appropriate problem parameters which arise when considering the problems discussed in this work.

Dual Parameters. In a parameterized problem, we call two parameters *dual* if they sum up to a value depending on the input size. For example, in the Vertex Cover problem as presented in Chapter 4, one asks, given a graph $G = (V, E)$ and a integer k with $0 \leq k \leq |V|$, for a vertex cover of size k. Instead, one could, given an integer k' with $0 \leq k \leq |V|$, also ask for a vertex cover of size $|V| - k'$; in fact, this is equivalent to asking whether the input graph has an independent set of size k'. Here, k and k' are dual parameters since they sum up to $|V|$. In general, there is no rule to infer the parameterized complexity with respect to one parameter from the parameterized complexity with respect to its dual parameterization. For example, Vertex Cover turns out to be fixed-parameter tractable with respect to parameter k and W[1]-hard with respect to parameter k'. In fact, Khot and Raman [115] pose the conjecture that "typically parametric dual problems have complementary parameterized complexity." In contrast to this conjecture, this work provides two examples of problems for which two dual parameterizations both yield fixed-parameter algorithms.

In Minimum Quartet Inconsistency (MQI) as presented in Chapter 6, we ask, given a set of n taxa, a set Q containing a quartet topology for every size-four set of species, and a non-negative integer k, whether there is a binary tree T inducing the set Q_T of quartet topologies such that $|Q - Q_T| \leq k$. Instead one could, given a non-negative integer k', also ask for a binary tree T with induced quartet topology set Q_T such that $|Q \cap Q_T| \geq k'$. Here, k and k' are dual parameters since they sum up to $|Q| = \binom{n}{4}$. We have seen in Chapter 6 that MQI is fixed-parameter tractable with respect to parameter k. It is also fixed-parameter tractable with respect to parameter k' since, for every set Q of quartet topologies, there is a binary tree inducing at least one third of the topologies in Q [114]. Therefore, we know that $k' > |Q|/3$ or, equivalently, that $|Q| \leq 3k'$ and, thus, already the exact algorithm proposed by Ben-Dor *et al.* [17] is a fixed-parameter algorithm with respect to parameter k'. For MQI, however, the parameterization by k is certainly preferable to the parameterization by k' since for realistic data we can assume that k' is significantly larger than $2|Q|/3$ [17, 89].

For Longest Arc-Preserving Common Subsequence (LAPCS), as presented in Chapter 8, however, it may depend on the data which of two possible parameterizations is more appropriate. Given two arc-annotated sequences S_1 and S_2 and a non-negative integer L, one can ask whether S_1 and S_2 have an arc-preserving common subsequence of length L. Alternatively, one can ask, as proposed in Chapter 8, given non-negative integers k_1 and k_2, whether we obtain an arc-preserving common subsequence by deleting k_1 bases in S_1 and k_2 bases in S_2. Here, we have dual parameters since $L + k_1 = |S_1|$ and $L + k_2 = |S_2|$. As described in Chapter 8, LAPCS is fixed-parameter tractable both with respect to L as well as with respect to k_1 and k_2. This offers a constructive and flexible approach to this problem. Depending on the input data, one can choose between the two approaches: For comparing similar sequences where we expect a long arc-preserving common subsequence, one might better choose the fixed-parameter algorithm with respect to k_1 and k_2. For comparing less similar sequences where we expect a small arc-preserving common subsequence, one might better choose the fixed-parameter algorithm with respect to L. With no knowledge about the input data, one could use a competitive approach where the two algorithms are started in parallel.

Several Parameters. Many problems exhibit not only one but a set of natural parameterizations. Here, it is interesting to investigate which parameterizations do allow fixed-parameter algorithms and for which they are unlikely, i.e., for which the problem becomes W[1]-hard. On the one hand, this allows better understanding of which parts of the input cause the combinatorial hardness of the problem. On the other hand, several fixed-parameter algorithms allow to propose a set of algorithms for a particular problem and to choose the appropriate one depending on the input data and on the resulting parameter values.

An example problem for such a situation is contained in this work, namely Closest String, for which we can give fixed-parameter algorithms with respect to two natural parameters. Given length-L strings s_1, $s_2, \dots, s_k$ and a non-negative integer d, we ask for a length-L string s with $d_H(s, s_i) \leq d$ for all $i = 1, \dots k$. In Subsection 5.1.2, we present a search tree algorithm which is fixed-parameter with respect to the distance parameter d and has a running time of $O(kL + kd \cdot d^d)$. In Subsection 5.1.3, we showed, by formulating the problem as an integer linear program that it is fixed-parameter tractable with respect to the number k of input strings as parameter.

In this way, Closest String offers two fixed-parameter parameter algorithms, each of them is with respect to a different natural parameter and each of them having its very own flavor. The search tree algorithm is easy-to-understand but non-trivial and the ILP approach introduces a novel technique to show fixed-parameter tractability. This makes Closest String a good example problem to illustrate the concepts of fixed-parameter algorithms, e.g., in class notes [149].

Guaranteed Values. There are parameterized problems which are, in general, NP-hard but which allow polynomial-time solutions when the parameter value is below some threshold. An example is Maximum Satisfiability, where we are given a formula F in conjunctive normal form with n variables and m clauses, and a non-negative integer k, and where we ask for an variable assignment that satisfies at least k clauses. Fixed-parameter algorithms with respect to parameter k have been proposed for this problem, the currently best result having $O(1.3695^k \cdot k^2 + |F|)$ running time [42]. For a random variable assignment, however, either the assignment itself or its inverse already satisfies half of the clauses. Therefore, the "guaranteed value" of the parameter is $k > m/2$ since for $k \leq m/2$, we can easily solve the problem in linear time. For this reason, Mahajan and Raman [135] propose a parameterization "above guaranteed values," i.e., to introduce a new parameter k' and to ask for an assignment that satisfies at least $m/2 + k'$ clauses. Regarding Maximum Satisfiability, we only shortly note that the problem also turns out to be fixed-parameter tractable with respect to k' having, however, an algorithm with worse running time than the fixed-parameter algorithm with respect to k; beyond that, even larger, less obvious guaranteed values than $m/2$ exist [135].

In this work, we encounter guaranteed parameter values in the example of Minimum Quartet Inconsistency (MQI) (Chapter 6). Given a set of quartet topologies, there is a polynomial-time heuristic algorithm [25] whose solution is guaranteed to be optimal for $k < (n-3)/2$ where n is the number of taxa and k is the parameter indicating the number of erroneous quartet topologies. Therefore, we can assume in the fixed-parameter algorithm that $k \geq (n-3)/2$. This also raises questions for the parameterization above guaranteed values in the style of Mahajan and Raman [135]. For MQI, this is to ask whether the problem is fixed-parameter tractable with respect to parameter k' when we want to find a tree that violates at most $(n-3)/2+k'$, $k' \geq 0$, quartet topologies.

One has to be aware of guaranteed parameter values and take them into account in the parameterized analysis. However, parameterizing above guaranteed values can become significantly more difficult (also see the discussion of guaranteed values in [148]).

Normalized Parameters. Parameterized approaches in many cases suffer from the fact that growing problem instances usually also require larger parameter values. An example problem to which this observation applies is Breakpoint Median as presented in Chapter 7. Given signed orderings π_1, $\pi_2, \ldots, \pi_k$ and a non-negative integer d, we ask for a median ordering π such that $\sum_{i=1}^{k} d_{bp}(\pi, \pi_i) \leq d$ where d_{bp} denotes the breakpoint distance (for details refer to Chapter 7).

Since the problem is NP-complete already for $k=3$ and, thus, there is no hope for a fixed-parameter algorithm with respect to parameter k, we concentrated in Chapter 7 on the distance parameter d. For Breakpoint Median, in contrast to, e.g., Closest String, the goal is to find an object (here an ordering π) such that the *sum* of distances (here breakpoint distances) to all input objects is bounded by d, whereas in Closest String the goal is to find an object (there a string s) such that the *maximum* of distances (there Hamming distances) to all input objects is bounded by d. Therefore, compared to Closest String, we can expect for Breakpoint Median that an increasing number of input objects implies an increasing optimal value of the distance parameter, i.e., the minimal value which allows a solution.

In Chapter 7, we outlined a fixed-parameter algorithm with respect to d with a $O(kn + d \cdot 2.15^d)$ running time. On the one hand, we expect that, with every additional input ordering, the optimal distance value increases. On the other hand, we made the observation that the base c of the exponential term c^d in the running time bound decreases towards 1 as k increases. This interesting observation leads us to propose a *normalized* distance parameter $d/f(k)$ where f is a function measuring the increase of the optimal d value for growing k. This raises two open questions: to determine an appropriate f function for practical settings and to investigate the parameterized complexity of the problem with respect to parameter $d/f(k)$. Our experimental results shown in Chapter 7 indicate that our algorithm seems fixed-parameter with respect to parameter $d/\log(k)$ but, using our algorithm, we cannot give any indication that the problem is fixed-parameter tractable with respect to parameter d/k.

The investigation of normalized parameters could also be interesting and fruitful for other computational biology problems such as multiple sequence alignment, e.g., with the sum-of-pairs scoring scheme [94]. This problem also seems intractable when parameterized by the number of input sequences. We conjecture, however, that the problem is fixed-parameter tractable with respect to the distance parameter reflecting the “cost” of an alignment, i.e., the sum of all pairwise Hamming distances in the computed alignment. Obviously, this distance parameter is likely to depend on the number of input sequences.

Chapter 10

Future Research Directions

In this chapter, I indicate new research directions in the intersection of parameterized complexity and computational biology. To this end, I, firstly, point to three subjectively selected areas of current computational biology research where the algorithmic techniques from this work could apply. Secondly, I show new directions regarding the techniques, which may be of particular interest in computational biology. Open research questions more closely related to the problems discussed in this work can be found at the end of each chapter.

10.1 Problem-Oriented

I outline three areas of computational biology which are subject of current research and seem suited for the application of the algorithmic techniques presented in this work.

10.1.1 Analysis of Microarray Data

In this section, combinatorial questions in the analysis of microarray data are discussed, a new and prominent field of bioinformatics research. As clustering problems play a central role in this context, we first discuss them in general and, then, give one concrete example of a graph-theoretic problem proposed specifically for the clustering of gene expression data. To exhibit a further aspect of microarray data analysis, we also address the detection of meaningful local patterns within the data. For background on microarray data analysis refer to [10, 175].

Clustering Problems. These are a class of problems for which the typical question is, given a set of data points (e.g., genes) and a pairwise similarity measure for the data points (e.g., the similarity of two expression patterns), to find a partition into subsets, the "clusters," such that all data points within one group are similar to each other. The concrete definition of the similarity within one group depends on the particular clustering problem. Studying general approaches for data clustering (see, e.g., [99, 105] for an overview), we encounter, when aiming for exact solutions, running times characterized by n^k or n^{dk} factors, where n is the number of given data points, d is the dimension, and k is the number of clusters. So far, little has been undertaken to restrict the exponential term in the running time to certain problem parameters, e.g., the number of clusters, the dimension, or the amount of noise in the data. These problem parameters are likely to be small in applications. Fixed-parameter algorithms working efficiently in such a scenario as well as novel hardness proofs could mean a breakthrough with impact beyond computational biology. We discuss, in the following, a specific graph-theoretic version of a clustering problem where a fixed-parameter algorithm could be successfully developed. For a survey of approaches for clustering gene expression data refer to [182].

Graph Modification Problems. With the CLICK program, Sharan and Shamir [181] provided software for clustering gene expression data. From a combinatorial point of view, the employed model is a (weighted) graph in which the (weighted) edges represent the similarity of two given data points. The question for a clustering of the data points can, then, be formulated as a Cluster Editing problem: The goal is to transform the graph by edge insertions and edge deletions of a minimum weight into a clique graph, i.e., into a graph that consists of vertex-disjoint cliques. While [181] presents a heuristic approach to solve this problem, it is the goal of [180] to inspect the complexity of Cluster Editing and some variants, showing that Cluster Editing is NP-complete and giving polynomial-time solutions for special cases.

Restricting, for simplicity, to unweighted graphs, a clique graph is a graph that does not contain a P_3, i.e., a path of three vertices, as a vertex-induced subgraph. Therefore, a straightforward algorithm is, given a graph that is not a clique graph, to search three vertices inducing a P_3 and to insert or delete an edge between these vertices in order to eliminate this "forbidden subgraph;" this algorithm derives from general results on forbidden subgraph problems by Cai [38]. Improving this algorithm in [81], it is shown that Cluster Editing has a problem kernel with $O(k^2)$ vertices and $O(k^3)$ edges and a search tree algorithm with $O(2.27^k + |V|^3)$ running time where k denotes the number of allowed edges modifications. While this algorithm was designed manually, it was, by automatizing the generation of the branching rules in the search tree algorithm, shown that Cluster Editing can be solved in $O(1.92^k + |V|^3)$ running time [80]. For a discussion of this automatically generated search tree algorithm refer to Subsection 10.2.2.

The parameterized complexity of Cluster Editing is still open when we consider the aggregate parameter containing both the number of allowed edge modifications and the number p of clusters. Additionally to the mentioned problem kernel, it seems promising to develop further heuristic reduction rules for the problem, in particular when addressing the case of weighted edges. From a graph-theoretic point of view, it seems interesting to develop refined fixed-parameter algorithms for other "forbidden subgraph" problems which are defined analogously as Cluster Editing but have another forbidden subgraph instead of a P_3. An example of such a forbidden subgraph problem motivated by applications in computational biology is Directed Perfect Phylogeny [163]. Concerning clustering problems, the approach sketched here could serve as an example of how to develop fixed-parameter algorithms for other versions of clustering problems.

Pattern Discovery Problems. A drawback of clustering techniques is that they often require that all data points can be assigned to some cluster. In the analysis of microarray data it is, however, already informative to identify a part of the data points (e.g., genes) that exhibit a similar pattern (e.g., a similar expression behavior for a subset of the conditions). A further goal is, because of the difficulty in the normalization of data, to keep the notion of "similarity" for expression patterns flexible. Ben-Dor *et al.* [18] propose a problem addressing these issues, Order Preserving Submatrix (OPSM), already addressed in Section 3.2. They show NP-hardness and develop heuristic solutions. Formally, the problem is, given an $n \times m$ matrix of real-numbered expression values, and two positive integers k and s, to determine whether there is a $k \times s$-submatrix and a permutation of columns in this submatrix such that the values within every row of this submatrix are strictly increasing. Roughly speaking, OPSM searches for genes that exhibit a similar expression pattern over a subset of conditions.

In applications, the input matrix has thousands of rows (the genes), but only a comparatively small number of columns (the tissues). Therefore, a natural direction seems to be the search for algorithms that have efficient running time when the number of columns in the matrix or the submatrix is fixed, i.e., the question for the fixed-parameter tractability of the problem with respect to these parameters. Regarding the size of the submatrix, the answer is a negative one: As discussed in Section 3.2, OPSM can be shown hard for the complexity class W[1] when the number of columns (or the number of rows) in the submatrix is the parameter. It remains to find more appropriate parameters or identify special cases of the problem for which it is possible to find exact solutions. An example for such a parameter is given by Tanay *et al.* [192], who present another pattern discovery problem for the analysis of microarray data on the same input as in OPSM. To make their problem computationally tractable, they exclude, on the same input as in OPSM, those rows (the genes) from the analysis which exhibit a high expression level (above a threshold) in more than d columns for a constant integer d, because, as they argue, these genes are unlikely to contribute important information to the analysis. Thus, they obtain, for a special case of their problem, a fixed-parameter algorithm with respect to d.

As a summary of this subsection, we recall two central challenges raised herein:

- Determine the parameterized complexity of clustering problems with respect to number of clusters, the dimension, or the amount of noise in the data.
- Determine the parameterized complexity of Cluster Editing with respect to both an upper bound p on the number of clusters and the number k of allowed edge modifications.

10.1.2 SNP Haplotyping Problems

A SNP (single nucleotide polymorphism) is a single base mutation in DNA. Thus, while most of the human DNA is identical, at particular positions (the SNP positions) human DNA can exhibit different nucleotides (the SNP states). In general, at one SNP position, one of two possible SNP states can be observed. As humans have two copies of each chromosome, the copies may carry different SNP states at the SNP positions. We distinguish *genotype* data, i.e., the common sequence information of both chromosomes, from *haplotype* data, i.e., the sequence information of only one of the chromosomes. In the following, we address two combinatorial questions from the context of analyzing these data and, then, point out two closely related graph problems which are important reference points in both questions.

SNP Haplotype Assembly Problems. In the sequencing of human DNA, a large number of short genome fragments is collected. These fragments derive from both chromosome copies, but it cannot be experimentally determined which fragment derives from which of the two chromosome copies. Therefore, it is a computational problem to assign each fragment to one of the two haplotypes and, thereby, to determine the SNP states of each haplotype. Lippert *et al.* [132] give a model for this question, as already outlined in Section 3.3.3. Describing their problem informally, its input is an alignment of fragments and the goal is to partition the fragments into two sets such that fragments exhibiting different SNP states at a particular SNP position belong to different sets. More formally, the input is given as a data matrix in which every row corresponds to a fragment and every column corresponds to a SNP position. The entries of the matrix are one character from the alphabet $\{0, 1, -\}$, where 0 and 1 indicate one of two possible SNP states and "–" indicates that the fragment does not contain any information about that SNP position, e.g., because the SNP is located outside the fragment. Given this matrix, the question is to partition the rows into two sets such that, in each of the two sets, no column contains a 0 as well as a 1 entry.

Example. Consider the following matrix:

$$\begin{pmatrix} - & 0 & 1 & 0 & 1 & - & - \\ - & - & 1 & 1 & 0 & 1 & - \\ - & - & - & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & - & - & - \end{pmatrix}$$

Here, the set of rows can be partitioned into two sets, one containing the first and third row, the other containing the second and fourth row. Then, for both sets, in rows of one set, no column contains a 0 as well as a 1 entry. □

Accounting for errors in the alignment or in the sequence, we arrive at an optimization problem that asks for the minimum number of fragments that have to be removed such that we can find a partition as specified; this optimization problem is referred to as Minimum Fragment Removal (MFR).

Example. In the matrix

$$\begin{pmatrix} - & 0 & 1 & 0 & 1 & - & - \\ - & - & 1 & 1 & 0 & 1 & - \\ - & - & - & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & - & - & - \\ - & 1 & 0 & 1 & 0 & - & - \end{pmatrix}$$

no bipartition of rows with the required property is possible. Omitting the last row, however, it is possible, since this leads to the situation described in the preceding example. Therefore, on the input of the shown matrix, deleting the last row yields a solution for MFR. □

Analogously, we define Minimum SNP Removal (MSR) asking for the minimum number of SNP positions to delete from the input data, or Minimum Entry Removal (MER) asking for a minimum number of entries in the data matrix to set to "–". MFR and MSR are solvable in polynomial time when the fragments are contiguous [170], i.e., when, within every row of the given matrix, no two entries from $\{0, 1\}$ are intermitted by one or more "–" entries. However, MER with contiguous fragments is NP-hard [159]. Rizzi *et al.* [170] address the more realistic scenario when the fragments are not contiguous which makes the mentioned problems NP-hard. Here, it would be of particular interest to design fixed-parameter algorithms when the parameter is the number of fragments (or the number of SNP positions or the number of matrix entries, respectively) to remove. In general, this seems to be difficult. But when considering the gap size as a parameter, i.e., the maximum number of "–" entries in one row which intermit entries from $\{0, 1\}$, a fixed-parameter algorithm for MFR was given [170] and a fixed-parameter algorithm for MSR is given in the long version of [170]; the question for algorithms which are fixed-parameter with respect to the gap size is still open for MER.

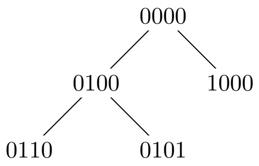
Reconstructing haplotype structure. Using microarray assays, SNP genotype data are obtained in a process called high-throughput SNP genotyping. As we, thereby, obtain genotype data only, a particular line of research is to deduce haplotype data when only given the genotype data.

This aspect is addressed by Eskin *et al.* [63]. Following the model of Gusfield [95], they describe how to compute, from genotypes for a set of individuals, the haplotypes under the assumption that the haplotypes have evolved following a *perfect phylogeny.* Here, a peferct phylogeny is, for a given set of

SNP positions, a binary tree in which all nodes are labelled with a sequence of SNP states, one for each SNP position, such that nodes exhibiting a common SNP state at a particular SNP position form a connected subtree (for a precise definition of a perfect phylogeny refer to [63, 95]). This problem is called Perfect Phylogeny Haplotype and defined more precisely as follows. Input is, similarly as in the preceding paragraph, a matrix A with entries from $\{0, 1, 2\}$ where rows correspond to genotypes and columns correspond to SNP positions, a 0 or 1 entry indicates that the respective genotype has a unique SNP state at this SNP position and a 2 entry indicates that the respective genotype has both SNP states at this SNP position. Given a length-l vector a with entries from $\{0, 1, 2\}$ and two length-l vectors b_1 and b_2 with entries from $\{0, 1\}$, we say that b_1 and b_2 are compatible with a if, for every $1 \leq i \leq l$, either $a[i] = b_1[i] = b_2[i]$, or $a[i] = 2$ and $b_1[i] \neq b_2[i]$. Then, Perfect Phylogeny Haplotype asks whether there is a matrix B with entries from $\{0, 1\}$, where for every row in A there are two compatible rows in B, and the rows in B form a perfect phylogeny. Herein, the matrix B, if existent, has the same number of columns as A, it has at least as many rows as A, and it has at most twice as many rows as A.

Example. Consider matrices A and B, and a perfect phylogeny on the rows of B as follows:

$$A = \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 1 & 2 & 2 \\ 1 & 0 & 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \text{ and}$$



Matrix B is a solution for Perfect Phylogeny Haplotype on the input of matrix A, where the first two rows of B correspond to the first row in A, the second and third row of B correspond to the second row in A, and the last rows in A and B correspond to each other. □

A polynomial-time algorithm for Perfect Phylogeny Haplotype is given in [63]. When, however, the data contain errors or there is no solution under the assumption of a perfect phylogeny, Eskin *et al.* propose to consider the following NP-hard optimization problem, called Minimum Genotype Removal: Given a matrix with entries from $\{0, 1, 2\}$ and a non-negative integer k, determine whether we can remove k rows in the matrix such that, after their removal, Perfect Phylogeny Haplotype yields a positive answer. How the algorithmic techniques of parameterized complexity can apply here, is indicated by the following.

Bipartite Graphs. When comparing the problem described in the previous two paragraphs, we make the following observations: In SNP haplotype assembly problems, the goal is to *assign* a given haplotype fragment, represented by its sequence of SNP states, to one of two possible haplotypes. In the reconstruction of haplotype structure, the goal is to *divide* the given genotype fragments, represented by their sequence of not necessarily unique SNP states, into two haplotype fragments each. Thus, on the one hand, both paragraphs describe different combinatorial problems. The commonality of both problems is that we require a bipartition of haplotype fragments into two sets such that haplotype fragments with differences in their SNP states belong to different sets. Therefore, on the other hand, the central optimization questions raised in the previous two paragraphs are both closely related to graph problems concerning bipartite graphs, described as follows.

In Edge Bipartization we ask, given a graph G and a non-negative integer k, whether we can transform the graph into a bipartite graph by deleting at most k edges. The corresponding problem considering vertex deletions instead of edge deletions is called Vertex Bipartization. Both problems are NP-complete [74] and, for both problems, it is an open question whether or not they are fixed-parameter tractable (e.g., see [115, 135] for Vertex Bipartization).

In the following, we make the connections between these graph problems and the problems from the SNP context more precise without going into the details of the reductions. Vertex Bipartization is equivalent to MFR with non-contiguous fragments; vertices of the graph (in the Vertex Bipartization instance) correspond to fragments (in the MFR instance), edges correspond to a conflict between two fragments and the number of vertices to remove corresponds to the number of fragments to remove. In an analogous way, we can “directly,” i.e., in linear time, reduce Edge Bipartization to MSR and MER. From the results shown in [63], it follows that Edge Bipartization can be reduced to Minimum Genotype Removal and vice versa.

Note that the mentioned reductions are parameter-preserving and, thus, clarifying the parameterized complexity of Edge and Vertex Bipartization with respect to the number k of edge modifications could have direct implications for the mentioned SNP problems and vice versa. If Edge and Vertex Bipartization would be W[1]-hard, then it may still be possible to achieve efficient algorithms for the SNP problems by making further assumptions about the structure of the input data.

We state the following open questions:

- Is MER fixed-parameter tractable when parameterized by the number of allowed entry removals? Is MER with non-contiguous fragments fixed-parameter tractable when parameterized by the number of allowed entry removals or when parameterized by the gap length?
- Is Minimum Genotype Removal fixed-parameter tractable when parameterized by the number of allowed genotype removals?
- Are Edge Bipartization and Vertex Bipartization fixed-parameter tractable with respect to the number of allowed edge deletions or vertex deletions, respectively?

10.1.3 Repeat and Duplication Analysis

More than ten percent of the human genome consists of tandem repeats, i.e., duplications of stretches of DNA into one or more adjacent copies, which are only approximately preserved in the course of evolution. In sequencing projects, these repeats are serious obstacles when trying to obtain reliable and complete sequence information. However, they can provide useful information as, e.g., they allow to detect micro-evolutionary signals and, thus, can be the basis to reconstruct evolutionary trees of populations within one species [22]. They are also used as laboratory tools, e.g., for DNA fingerprinting, since the number of their copies is variable among single individuals (see [20] and references therein). Moreover, tandem repeats are implicated in the causation of inherited human diseases, e.g. Huntington's disease [20].

The study of combinatorial problems in the analysis of tandem repeats exhibits also NP-hard problems. Central among them is the question of repeat history, i.e., the question to reconstruct how the observed repeat copies evolved from one common ancestor.

Repeat History. The goal of repeat history is to try to clarify evolutionary history that led to a given ordered sequence of duplicated gene copies that all originated from one common ancestor; naturally, this history can be modeled by a rooted tree. The question to reconstruct the repeat history for an observed sequence of repeat copies has been addressed in several works, e.g., by Benson and Dong [21], by Tang *et al.* [193], by Elemento *et al.* [62], and by Jaitly *et al.* [106] and, in essence, is, though there are differences in the used models, the search for a parsimonious tree with respect to duplication events. This problem has been shown NP-hard for particular formulations [106] and is conjectured to be NP-hard in general. Therefore, this problem has been addressed using heuristics [193, 61], constant-factor approximations [21, 193], PTAS's [106], exponential-time dynamic programming [193], and exhaustive search [62] results. It does not seem that the techniques of parameterized complexity have directly been applied to these problems, although some results are fixed-parameter algorithms, e.g., Tang *et al.* [193] give an exact algorithm with running time $O(4^{2k}(k + n^3))$. Herein, k is the length of repeat units and n is their number. Note, however, that this is not a satisfying result in situations where k tends to be large, e.g., $k \geq 30$, which is likely if we study gene duplications or duplications of other large repeat units. Obvious parameters to consider instead could be the number of duplication events or the "block size" which denotes the maximum number of genes duplicated in one duplication event; in search for exact solutions, the mentioned works, so far, mainly focused on the special case of block size one.

We raise the following open questions:

- Is Gene Duplication [193] fixed-parameter tractable with respect to the number of duplication events or the block size?

- Does Tandem Repeat History [106], in addition to the PTAS proposed in [106], have an EPTAS? Note that, if Tandem Repeat History is W[1]-hard with respect to the "cost" of the repeat history, then the problem does not have an EPTAS unless FPT = W[1].

10.2 Technique-Oriented

We exhibit two current research directions concerning the further development of the algorithmic search tree techniques presented in this work.

10.2.1 Data Reduction

For graph problems, preprocessing techniques have been developed that reduce the size of the input instance but do not sacrifice the optimality of possible solutions. This can be the key to solve combinatorially hard problems in practice. The Nemhauser-Trotter theorem [116, 144] provides a prominent example for the Vertex Cover problem in graph theory. New reduction rules for Dominating Set [1] give a more recent example. Typically, data reduction is achieved by giving reduction rules for a problem (see Section 4.2). Special reduction rules are those that lead to a problem kernel (see Section 4.2); in this case, we have guarantees on the size of the reduced instance.

In computational biology, examples where such techniques are applied, are rare. One example is given by showing a problem kernel for Cluster Editing which received attention in the context of the clustering of microarray data (for a definition of the problem and its applications see Subsection 10.1.1). The input of Cluster Editing consists of a graph G and a non-negative integer k. Then, it is shown in [81] that this graph can be reduced to a graph having $O(k^2)$ vertices and $O(k^3)$ edges. The efficiency of this reduction on practical data is still to be evaluated.

We raise the following open question:

- Find, implement, and analyze data reduction techniques (possibly leading to problem kernels) for problems in computational biology.

10.2.2 Automated Generation of Search Tree Algorithms

Search tree algorithms are based on a set of branching rules and their analysis as outlined in Chapter 4. In many cases the set of branching rules is complex. For example, consider search tree algorithms proposed for Vertex Cover [43, 150] or MaxSat [12, 42, 152].

The seemingly first example of an automatically generated search tree algorithm is given for Cluster Editing [80] (for a definition of the problem and its applications see Subsection 10.1.1). A manually designed search tree algorithm was given in [81]. By the automatization, the worst-case bound on the search tree size was improved from 2.27^k, where k is the problem parameter, to 1.92^k. The automatically generated set contains 108 branching rules and branching vectors which have up to 37 entries. It seems impossible to design complex algorithms like this one manually. Thus, on the one hand, the automated generation enables much more fine-grained branching rules and improved worst-case bounds. On the other hand, it remains to be analyzed to what extent more complex branching rules also imply a better performance in practice, since they, naturally, also imply an increased time factor in every node of the search tree. Moreover, it seems impossible to manually check the correctness and the analysis of such a large set of complex branching rules.

We pose the following open question:

- Provide new examples (besides [80]) for automatically generated bounded search trees.

Bibliography

[1] J. Alber, M.R. Fellows, and R. Niedermeier. Efficient data reduction for Dominating Set: a linear problem kernel for the planar case. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT)*, number 2368 in LNCS, pages 150–159. Springer, 2002. → 175

[2] J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Towards optimally solving the Longest Common Subsequence problem for sequences with nested arc annotations in linear time. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 2373 in LNCS, pages 99–114. Springer, 2002. Long version to appear in *Theoretical Computer Science.* → vi, 141, 143, 144, 146, 147, 195

[3] J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229(1–3):3–27, 2001. → 3, 195

[4] B. L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5:1–13, 2001. → 29

[5] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995. → 24, 156

[6] S. Arora, A. Frieze, and H. Kaplan. A new rounding procedure for the assignment problem with applications to dense graph arrangement problems. *Mathematical Programming*, 92(1):1–36, 2002. → 158

[7] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation—Combinatorial Optimization Problems and their Approximability Properties.* Springer-Verlag, 1999. → 2, 6, 157

[8] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distances between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001. → 28, 135

[9] R. Balasubramanian, M. R. Fellows, and V. Raman. An improved fixed parameter algorithm for Vertex Cover. *Information Processing Letters*, 65(3):163–168, 1998. → 43

[10] P. Baldi and G. W. Hatfield. *DNA Microarrays and Gene Expression: From Experiments to Data Analysis and Modeling.* Cambridge University Press, 2002. → 167

[11] H.-J. Bandelt and A. Dress. Reconstructing the shape of a tree from observed dissimilarity data. *Advances in Applied Mathematics*, 7:309–343, 1986. → vi, 87, 89, 90

[12] N. Bansal and V. Raman. Upper bounds for MaxSat: further improved. In *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC)*, number 1741 in LNCS, pages 247–258. Springer, 1999. → 175

[13] Z. Bar-Joseph, E. Demaine, David K. Gifford, A. M. Hamel, T. S. Jaakkola, and N. Srebro. K-ary clustering with optimal leaf ordering for gene expression data. *Bioinformatics*, 19(9):1070–1078, 2003. → 4

[14] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S. R. Eddy, S. Griffiths-Jones, K. L. Howe, M. Marshall, and E.L. Sonnhammer. The PFAM protein families database. *Nucleic Acids Research*, 30(1):276–280, 2000. → 154

[15] A. Baxevanis. Practical aspects of multiple sequence alignment. In A. Baxevanis and B. F. Francis Ouellette, editors, *Bioinformatics–A Practical Guide to the Analysis of Genes and Proteins*, pages 172–188. Wiley-Liss, 1998. → 18, 63

[16] C. Bazgan. Schémas d'approximation et complexité parametrée. Technical report, DEA d'Informatique 'a Orsay, 1995. → 159

[17] A. Ben-Dor, B. Chor, D. Graur, R. Ophir, and D. Pelleg. Constructing phylogenies from quartets: elucidation of eutherian superordinal relationships. *Journal of Computational Biology*, 5:377–390, 1998. → 86, 89, 162

[18] A. Ben-Dor, B. Chor, R. Karp, and Z. Yakhini. Discovering local structure in gene expression data: the Order-Preserving Submatrix problem. In *Proceedings of 6th Annual International ACM Conference on Computational Biology (RECOMB)*, pages 49–57. ACM Press, 2002. → 27, 169

[19] A. Ben-Dor, G. Lancia, J. Perone, and R. Ravi. Banishing bias from consensus sequences. In *Proceedings of the 8th Annual Symposium Combinatorial Pattern Matching (CPM)*, number 1264 in LNCS, pages 247–261. Springer, 1997. → 46, 59, 62

[20] G. Benson. Sequence alignment with tandem duplication. *Journal of Computational Biology*, 4:351–367, 1997. → 174

[21] G. Benson and L. Dong. Reconstructing the duplication history of a tandem repeat. In *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 44–53. AAAI Press, 1999. → 174

[22] S. Bèrard and É. Rivals. Comparison of minisatellites. *Journal of Computational Biology*, 10(3–4):357–372, 2003. → 174

[23] V. Berry, D. Bryant, T. Jiang, P. Kearney, M. Li, T. Wareham, and H. Zhang. A practical algorithm for recovering the best supported edges of an evolutionary tree. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 287–296. ACM Press, 2000. → 88, 100, 161

[24] V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240:271–298, 2000. Software available through `http://www.lirmm.fr/~vberry/PHYLOQUART/phyloquart.html`. → 87, 88, 95, 100, 103, 161

[25] V. Berry, T. Jiang, P. Kearney, M. Li, and T. Wareham. Quartet cleaning: improved algorithms and simulations. In *Proceedings of the 7th European Symposium on Algorithms (ESA)*, number 1643 in LNCS, pages 313–324. Springer, 1999. → 86, 88, 100, 107, 161, 163

[26] O. R. P. Bininda-Emonds, editor. *Phylogenetic Supertrees.* Kluwer Academic Press, 2003. To appear. → 5

[27] M. Blanchette, B. Schwikowski, and M. Tompa. Algorithms for phylogenetic footprinting. *Journal of Computational Biology*, 9(2):211–224, 2002. → 4, 30, 31, 45, 65, 66, 68

[28] H. L. Bodlaender, R. G. Downey, M. R. Fellows, M. T. Hallett, and H. T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11:49–57, 1995. → 27, 138, 141

[29] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and H. T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147:31–54, 1995. → 27, 138, 141

[30] P. Bonizzoni and G. Della Vedova. The complexity of multiple sequence alignment with SP-score that is a metric. *Theoretical Computer Science*, 259(1/2):63–79, 2001. → 18

[31] I. N. Bronstein and K. A. Semendjajew. *Taschenbuch der Mathematik.* Teubner, 1991. → 42

[32] T. A. Brown. *Genomes.* Wiley-Liss, 1999. → 11, 152

[33] D. Bryant. The complexity of the Breakpoint Median problem. Technical Report CRM-2579, Centre de recherches mathématiques, Université de Montréal, 1998. → 109, 110, 113, 114

[34] D. Bryant and M. Steel. Constructing optimal trees from quartets. *Journal of Algorithms*, 38:237–259, 2001. → 88

[35] J. Buhler and M. Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, 2002. → 45, 64, 65, 68

[36] P. Buneman. The recovery of trees from measures of dissimilarity. In F. R. Hodson, D. G. Kendall, and P. Tautu, editors, *Anglo-Romanian Conference on Mathematics in the Archaeological and Historical Sciences*, pages 387–395. Edinburgh University Press, 1971. → 87, 88, 89

[37] J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22(3):560–572, 1993. → 37

[38] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996. → 168

[39] A. Caprara. On the practical solution of the Reversal Median problem. In *Proceedings of the First Workshop on Algorithms in Bioinformatics (WABI)*, number 2149 in LNCS, pages 238–251. Springer, 2001. → 5, 133, 135

[40] M. Cesati. Compendium of parameterized problems, 2001. Available via `http://bravo.ce.uniroma2.it/home/cesati/research/`. → 26

[41] M. Cesati and L. Trevisan. On the efficiency of polynomial time approximation schemes. *Information Processing Letters*, 64(4):165–171, 1997. → 6, 46, 66, 84, 159

[42] J. Chen and I. Kanj. Improved exact algorithms for MAX-SAT. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, number 2286 in LNCS, pages 341–355. Springer, 2002. → 163, 175

[43] J. Chen, I. Kanj, and W. Jia. Vertex Cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001. → 3, 4, 23, 43, 157, 175

[44] W. Chen. A more efficient algorithm for ordered tree inclusion. *Journal of Algorithms*, 26(2):370–385, 1998. → 139

[45] B. Chor. From quartets to phylogenetic trees. In *Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, number 1521 in LNCS, pages 36–53. Springer, 1998. → 85, 87

[46] B. Chor. Personal communication, August 2001. → 107

[47] B. Chor. Personal communication, January 2002. → 27

[48] H. Colonius and H. H. Schultze. Tree structure for proximity data. *British Journal of Mathematical and Statistical Psychology*, 34:167–180, 1981. → 87

[49] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetical progression. *J. Symbolic Computations*, 9:251–280, 1990. → 23

[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2nd edition, 2001. → 5

[51] M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L. S. Wang, T. Warnow, and S. Wyman. An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae. In D. Sankoff and J. Nadeau, editors, *Comparative Genomics*, pages 99–121. Kluwer, 2000. → 109, 110, 128, 133, 134

[52] M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L. S. Wang, T. Warnow, and S. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 104–115. AAAI Press, 2000. → 109, 110, 133, 134

[53] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002. → 26

[54] M. Davis and H. Putnam. A computing procedure for quantification. *Journal of the ACM*, 7:201–215, 1960. → 4

[55] C. de la Higuera and F. Casacuberta. Topology of strings: Median String is NP-complete. *Theoretical Computer Science*, 230(1–2):39–48, 2000. → 84

[56] X. Deng, G. Li, Z. Li, B. Ma, and L. Wang. A PTAS for distinguishing (sub)string selection. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, number 2380 in LNCS, pages 740–751. Springer, 2002. → 47, 48, 56, 66, 157, 159

[57] R. G. Downey and M. R. Fellows. Parameterized computational feasibility. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 219–244. Boston: Birkhäuser, 1995. → 25

[58] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999. → 3, 21, 23, 24, 25, 26, 45, 141, 156

[59] R. G. Downey and M. R. Fellows. Parameterized complexity after (almost) ten years: review and open questions. In *Combinatorics, Computation & Logic, DMTCS'99 and CATS'99*, Australian Computer Science Communcations, Volume 21 Number 3, pages 1–33. Springer-Verlag Singapore, 1999. → 23

[60] R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: a framework for systematically confronting computational intractability.

In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49 of *AMS-DIMACS*, pages 49–99. AMS Press, 1999. → 2, 23

[61] O. Elemento and O. Gascuel. An efficient and accurate distance based algorithm to reconstruct tandem duplication trees. In *Proceedings of the First European Conference on Computational Biology (ECCB)*, volume 18 (Supplement 2) of *Bioinformatics*, pages S92–S99. Oxford University Press, 2002. → 174

[62] O. Elemento, O. Gascuel, and M.-P. Lefranc. Reconstructing the duplication history of tandemly repeated genes. *Molecular Biology and Evolution*, 19(3):278–288, 2002. → 174

[63] E. Eskin, E. Halperin, and R. M. Karp. Efficient reconstruction of haplotype structure via perfect phylogeny. In *Proceedings of the 7th Annual International ACM Conference on Computational Biology (RECOMB)*, pages 104–113. ACM Press, 2003. → 171, 172, 173

[64] P. A. Evans. Algorithms and complexity for annotated sequence analysis. PhD thesis, Department of Computer Science, University of Victoria, Canada, 1999. → 4, 31, 137, 139, 140, 141, 153

[65] P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proceedings of the 10th Annual Symposium Combinatorial Pattern Matching (CPM)*, number 1645 in LNCS, pages 270–280. Springer, 1999. → 31, 137, 139, 140, 141, 153

[66] P. A. Evans, A. D. Smith, and H. T. Wareham. On the complexity of finding common approroximate substrings. *Theoretical Computer Science*, 2003. To appear. → 66

[67] P. A. Evans and H. T. Wareham. Practical algorithms for universal DNA primer design: An exercise in algorithm engineering. In N. El-Mabrouk, T. Lengauer, and D. Sankoff, editors, *Currents in Computational Molecular Biology 2001*, pages 25–26. Publications CRM, Montreal, 2001. → 48, 50, 66, 68, 137

[68] M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC)*, number 2223 in LNCS, pages 291–307. Springer, 2001. → 3, 110, 140, 160

[69] M. R. Fellows. Parameterized complexity: the main ideas and connections to practical computing. In *Experimental Algorithmics*, number 2547 in LNCS, pages 51–77. Springer, 2002. → 3, 6, 66, 157, 160

[70] M. R. Fellows, J. Gramm, and R. Niedermeier. On the parameterized intractability of Closest Substring and related problems. In *Proceedings of the 19th International Symposium on Theoretical Aspects of Computer*

Science (STACS), number 2285 in LNCS, pages 262–273. Springer, 2002. Long version submitted. → vi, 195

[71] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.5c, 1993. Distributed by the author. Department of Genetics, University of Washington, Seattle. Available through `http://evolution.genetics.washington.edu/phylip`. → 103, 104

[72] P. Flajolet and R. Sedgewick. Analytic combinatorics, 2003. To appear, see `http://algo.inria.fr/flajolet/Publications/books.html`. → 42

[73] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997. → 46, 47, 66

[74] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979. → 2, 7, 24, 173

[75] L. Gąsieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the Hamming Center problem. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 905–906. ACM Press, 1999. → 46, 47

[76] L. Gąsieniec, J. Jansson, and A. Lingas. Approximation algorithms for Hamming Clustering problems. In *Proceedings of the 11th Annual Symposium Combinatorial Pattern Matching (CPM)*, number 1848 in LNCS, pages 108–118. Springer, 2000. → 46

[77] R. Giegerich and C. Meyer. Algebraic dynamic programming. In *Proceedings of the 9th International Conference on Algebraic Methodology And Software Technology (AMAST)*, number 2422 in LNCS, pages 349–364. Springer, 2002. → 140

[78] D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 512–521. IEEE Computer Society, 1999. → 137, 153, 154

[79] J. Gramm. Exact algorithms for Max2Sat and their applications. Diploma thesis, WSI für Informatik, Universität Tübingen, October 1999. Available from `http://www-fs.informatik.uni-tuebingen.de/~gramm/`. → 43

[80] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automatized generation of search tree algorithms for graph modification problems. In *Proceedings of the 11th Annual European Symposium on Algorithms*, number 2832 in LNCS. Springer, 2003. To appear. Long version submitted. → 168, 176, 195

[81] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: fixed-parameter algorithms for clique generation. In *Proceedings of the 5th Italian Conference on Algorithms and Complexity (CIAC)*, number 2653 in LNCS, pages 108–119. Springer, 2003. Long version invited for submission to *Theory of Computing Systems.* → 168, 175, 176, 195

[82] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, number 2556 in LNCS, pages 182–193. Springer, 2002. Long version submitted. → vi, 140, 141, 152, 195

[83] J. Gramm, J. Guo, and R. Niedermeier. On exact and approximation algorithms for Distinguishing Substring Selection. In *Proceedings of the 14th International Symposium on Fundamentals of Computation Theory (FCT)*, number 2751 in LNCS, pages 195–209. Springer, 2003. Long version submitted. → 58, 66, 157, 158, 159, 195

[84] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. Worst-case upper bounnds for MAX-2-SAT with application to MAX-CUT. *Discrete Applied Mathematics*, 130(2):139–155, 2003. → 195

[85] J. Gramm, F. Hüffner, and R. Niedermeier. Closest strings, primer design, and motif search. In L. Florea, B. Walenz, and S. Hannenhalli, editors, *Currents in Computational Molecular Biology 2002*, pages 74–75. 2002. Poster abstracts of RECOMB 2002. → 64, 195

[86] J. Gramm and R. Niedermeier. Faster exact solutions for Max2Sat. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC)*, number 1767 in LNCS, pages 174–186. Springer, 2000. → 195

[87] J. Gramm and R. Niedermeier. Evaluating an algorithm for parameterized Minimum Quartet Inconsistency. In N. El-Mabrouk, T. Lengauer, and D. Sankoff, editors, *Currents in Computational Molecular Biology 2001*, pages 195–196. Les Publications CRM, Montréal, 2001. Poster abstracts of RECOMB 2001. → 195

[88] J. Gramm and R. Niedermeier. Minimum Quartet Inconsistency is fixed parameter tractable. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 2089 in LNCS, pages 241–256. Springer, 2001. Long version to appear in *Journal of Computer and System Sciences*. In press. → vi, 106, 195

[89] J. Gramm and R. Niedermeier. Minimum Quartet Inconsistency is fixed parameter tractable. Technical Report WSI-2001-3, WSI für Informatik, Universität Tübingen, Jan 2001. Report available through `http://www-fs.informatik.uni-tuebingen.de/~gramm/`. → vi, 98, 106, 162

[90] J. Gramm and R. Niedermeier. Breakpoint medians and breakpoint phylogenies—a fixed-parameter approach. In *Proceedings of the First European Conference on Computational Biology (ECCB)*, volume 18 (Supplement 2) of *Bioinformatics*, pages S128–S139. Oxford University Press, 2002. → vi, 4, 195

[91] J. Gramm, R. Niedermeier, and P. Rossmanith. Exact solutions for Closest String and related problems. In *Proceedings of the 12th Annual Symposium on Algorithms and Computation (ISAAC)*, number 2223 in LNCS, pages 441–453. Springer, 2001. Long version to appear in *Algorithmica*. → 185

[92] J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for Closest String and related problems. *Algorithmica*, 37(1):25–42, 2003. Long version of [91]. → v, 59, 195

[93] J. Guo. Exact algorithms for the Longest Common Subsequence problem for arc-annotated sequences. Diploma thesis, WSI für Informatik, Universität Tübingen, February 2002. Available from `http://www-fs.informatik.uni-tuebingen.de/~guo`. → vi, 140, 142, 143, 144, 146, 147

[94] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. → 2, 18, 19, 139, 141, 164

[95] D. Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proceedings of 6th Annual International ACM Conference on Computational Biology (RECOMB)*, pages 166–175. ACM Press, 2002. → 171, 172

[96] M. T. Hallett. An integrated complexity analysis of problems from computational biology. PhD thesis, Department of Computer Science, University of Victoria, Canada, 1996. → 4

[97] M. T. Hallett and J. Lagergren. New algorithms for the duplication-loss model. In *Proceedings of the 4th Annual International ACM Conference on Computational Biology (RECOMB)*, pages 138–146. ACM Press, 2000. → 4

[98] S. Hannenhalli and P. Pevzner. To cut ... or not to cut (applications of comparative physical maps in molecular evolution). In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 304–313, 1996. → 4, 29

[99] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79:191–215, 1997. → 168

[100] D. S. Hirschberg. Recent results on the complexity of common subsequence problems. In D. Sankoff and J. Kruskal, editors, *Time Warps, String Edits, and Macromolecules—The Theory and Practice of Sequence Comparison*, pages 325–330. CSLI Publications, 1983. → 139

[101] D. S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. Boston, MA: PWS Publishing Company, 1997. → 2, 6, 157

[102] B. D. Hughey, G. C. Adams, T. D. Bruns, and D. S. Hibbett. Phylogeny of Calostoma, the gelatinous-stalked puffball, based on nuclear and mitochondrial ribosomal DNA sequences. *Mycologia*, 92(1):94–104, 2000. → 104, 105, 106

[103] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. → 26

[104] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. → 26

[105] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1988. → 168

[106] D. Jaitly, P. Kearney, G. Lin, and B. Ma. Methods for reconstructing the history of tandem repeats and their application to the human genome. *Journal of Computer and Systems Sciences*, 65:494–507, 2002. → 157, 174, 175

[107] T. Jiang, P. Kearney, and M. Li. Orchestrating quartets: approximation and data correction. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 416–425. IEEE Computer Society, 1998. → 88, 89, 161

[108] T. Jiang, P. Kearney, and M. Li. Some open problems in computational molecular biology. *Journal of Algorithms*, 34:194–201, 2000. → 85, 86, 89, 158

[109] T. Jiang, P. Kearney, and M. Li. A polynomial time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM Journal on Computing*, 30(6):1942–1961, 2001. → 86, 88, 89, 158, 161

[110] T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The Longest Common Subsequence problem for arc-annotated sequences. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 1848 in LNCS, pages 154–165. Springer, 2000. Long version to appear in *Journal of Discrete Algorithms.* → 137, 139

[111] T. Jiang and L. Wang. Algorithmic methods for multiple sequence alignment. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 71–110. MIT Press, 2002. → 18

[112] R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(415–440), 1987. → 58

[113] H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 29(3):880–892, 1999. → 28

[114] P. Kearney. Phylogenetics and the quartet method. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 112–134. MIT Press, 2002. → 85, 162

[115] S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science*, 289:997–1008, 2002. → 161, 173

[116] S. Khuller. Algorithms column: The Vertex Cover problem. *ACM SIGACT News*, 33(2):31–33, 2002. → 175

[117] P. Kilpeläinen. Tree matching problems with applications to structured text data bases. PhD thesis, University of Helsinki, Finland, 1992. → 139

[118] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995. → 139

[119] B. Korte and J. Vygen. *Combinatorial Optimization—Theory and Algorithms.* Springer, 2002. → 7, 155

[120] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999. → 123

[121] J. C. Lagarias. Point lattices. In R. L. Graham, M. Grötschel, and L. Lovasz, editors, *Handbook of Combinatorics*, pages 919–966. MIT Press, 1995. → 58

[122] G. Lancia, V. Bafna, S. Istrail, and R. Schwartz. SNPs problems, complexity, and algorithms. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 182–193. Springer, 2001. → 32

[123] G. Lancia, R. Carr, B. Walenz, and S. Istrail. 101 optimal PDB structure alignments: a branch-and-cut algorithm for the Maximum Contact Map Overlap problem. In *Proceedings of the 5th Annual International ACM Conference on Computational Biology (RECOMB)*, pages 193–202. ACM Press, 2001. → 137, 153

[124] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing String Selection problems. *Information and Computation*, 185(1):41–55, 2003. → 46, 47, 48, 56, 65, 66, 67

[125] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983. → 58, 156

[126] B. Lewin. *Genes VII.* Oxford University Press, 2000. → 11

[127] M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. In *Proceedings of the 31st ACM Symposium on Theory of Computing (STOC)*, pages 473–482. ACM Press, 1999. Preliminary version of [128] and [129]. → 68

[128] M. Li, B. Ma, and L. Wang. Finding similar regions in many sequences. *Journal of Computer and System Sciences*, 65(1):73–96, 2002. → 5, 45, 65, 67, 68, 188

[129] M. Li, B. Ma, and L. Wang. On the Closest String and Substring problems. *Journal of the ACM*, 49(2):157–171, 2002. → 5, 45, 46, 47, 65, 67, 68, 157, 158, 188

[130] G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The Longest Common Subsequence problem for sequences with nested arc annotations. *Journal of Computer and System Sciences*, 63(3):465–480, 2002. → 137, 138, 139, 158, 159

[131] C. Linhart and R. Shamir. The Degenerate Primer Design problem. In *Proceedings of the 10th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, volume 18 (Supplement 1) of *Bioinformatics*, pages S172–S180. Oxford University Press, 2002. → 157

[132] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail. Algorithmic strategies for the nucleotide polymorphism haplotype assembly problem. *Briefings in Bioinformatics*, 3(1):23–31, 2002. → 170

[133] F. Lisacek, Y. Diaz, and F. Michel. Automatic identification of group I intron cores in genomic DNA sequences. *Journal of Molecular Biology*, 235:1206–1217, 1994. → 5

[134] R. B. Lyngsø and C. N. S. Pedersen. Pseudoknot prediction in energy based models. *Journal of Computational Biology*, 7(3):409–427, 2000. → 153

[135] M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999. → 107, 163, 173

[136] Z. Michalewicz and D. B. Fogel. *How to Solve it: Modern Heuristics.* Springer-Verlag, 2000. → 2

[137] V. Morell. TreeBASE: the roots of phylogeny. *Science*, 273:569, 1996. → 104

[138] B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. *Journal of Supercomputing*, 22:99–111, 2002. → 109, 132

[139] B. M. E. Moret, A. C. Siepel, J. Tang, and T. Liu. Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in LNCS, pages 521–536. Springer, 2002. → 128, 135

[140] B. M. E. Moret, J. Tang, and T. Warnow L. Wang. Steps toward accurate reconstruction of phylogenies from gene-order data. *Journal of Computer and System Sciences*, 65(3):508–525, 2002. → 2, 109, 134

[141] B. M. E. Moret, L. Wang, T. Warnow, and S. K. Wyman. New approaches to phylogeny reconstruction from gene order data. In *Proceedings of the 9th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, volume 17 of *Bioinformatics*, pages S165–S173. Oxford University Press, 2001. → 109, 110, 133, 134

[142] B. M. E. Moret, S. K. Wyman, D. A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium on Bioinformatics*, pages 583–594, 2001. Available via `http://psb.stanford.edu/psb-online/`. → 5, 30, 109, 110, 127, 129, 132, 133, 134

[143] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995. → 2

[144] G. L. Nemhauser and L. E. Trotter. Vertex packing: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975. → 37, 175

[145] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988. → 7, 58, 155

[146] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985. → 23

[147] F. Nicolas and E. Rivals. Complexities of the centre and median string problems. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 2676 in LNCS, pages 315 – 327. Springer, 2003. → 84

[148] R. Niedermeier. Invitation to fixed-parameter algorithms. Habilitation thesis, WSI für Informatik, Universität Tübingen, 2002. → 164

[149] R. Niedermeier and J. Alber. Vorlesungsskript Parametrisierte Algorithmen WS 2002/3, 2003. Class notes and slides available via `http://www-fs.informatik.uni-tuebingen.de/~niedermr`. → 7, 163

[150] R. Niedermeier and P. Rossmanith. Upper bounds for Vertex Cover further improved. In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, number 1563 in LNCS, pages 561–570. Springer, 1999. → 43, 175

[151] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125–129, 2000. → 37, 53

[152] R. Niedermeier and P. Rossmanith. New upper bounds for Maximum Satisfiability. *Journal of Algorithms*, 36:63–88, 2000. → 175

[153] R. Niedermeier and P. Rossmanith. An efficient fixed parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 1:89–102, 2003. → 33, 98, 160

[154] R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for Weighted Vertex Cover, 2003. → 3, 4, 23, 43, 157

[155] T. Oda, C. Tanaka, and M. Tsuda. Molecular phylogeny of Japanese Amanita species based on nucleotide sequences of the internal transcribed spacer region of nuclear ribosomal DNA. *Mycoscience*, 40:57–64, 1999. → 104, 105

[156] A. M. Odlyzko. Asymptotic enumeration methods. In R. L. Graham, M. Grötschel, and L. Lovasz, editors, *Handbook of Combinatorics*, pages 1063–1229. MIT Press, 1995. → 58

[157] R. D. M. Page. Modified mincut supertrees. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in LNCS, pages 537–552. Springer, 2002. → 5

[158] R. D. M. Page and E. C. Holmes. *Molecular Evolution—A Phylogenetic Approach.* Blackwell Science, 1998. → 5, 18, 19, 85

[159] A. Panconesi. Personal communication, April 2003. → 171

[160] C. N. S. Pederson. Algorithms in computational biology. PhD thesis, Department of Computer Science, Aarhus, Denmark, 2000. → 11

[161] I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. Technical Report 98-071, Electronic Colloquium on Computational Complexity, Trier, 1998. → 109

[162] I. Pe'er and R. Shamir. Approximation algorithms for the Permutations Median problem in the breakpoint model. In D. Sankoff and J. Nadeau, editors, *Comparative Genomics*, pages 225–241. Kluwer, 2000. → 157

[163] I. Pe'er, R. Shamir, and R. Sharan. On the generality of phylogenies from incomplete directed characters. In *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT)*, number 2368 in LNCS, pages 358–367. Springer, 2002. → 169

[164] P. A. Pevzner. *Computational Molecular Biology—An Algorithmic Approach.* MIT Press, 2000. → 2, 5, 45

[165] P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 269–278. AAAI Press, 2000. → 45, 64, 65, 68, 109, 110, 129

[166] K. Pietrzak. On the parameterized complexity of the fixed alphabet Shortest Common Supersequence and Longest Common Subsequence problems. *Journal of Computer and System Sciences*, 2003. To appear. → 27

[167] S. Rahmann. Rapid large-scale oligonucleotide selection for microarrays. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in LNCS, page 134. Springer, 2002. → 63

[168] T. Richter. A new algorithm for the Ordered Tree Inclusion problem. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 1264 in LNCS, pages 150–166. Springer, 1997. → 139

[169] E. Rivas and S. R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285:2053–2068, 1999. → 153

[170] R. Rizzi, V. Bafna, S. Istrail, and G. Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *Proceedings of the Second Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in LNCS, pages 29–43. Springer, 2002. → 4, 32, 171

[171] F. S. Roberts. *Applied Combinatorics.* Prentice-Hall, 1984. → 40

[172] M.-F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proceedings of the 3rd Latin American Symposium on Theoretical Informatics (LATIN)*, number 1380 in LNCS, pages 111–127. Springer, 1998. → 65, 66, 68

[173] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4, 1987. → 19

[174] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998. → 5, 30, 110, 122, 127, 129, 130, 132, 134

[175] M. Schena. *DNA Microarrays: A Practical Approach.* Oxford University Press, 2001. → 167

[176] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1999. → 7, 58, 155

[177] R. Sedgewick and P. Flajolet. *Analysis of Algorithms.* Addison Wesley, 1996. → 42

[178] C. Semple and M. Steel. *Phylogenetics.* Oxford University Press, 2003. → 18

[179] J. C. Setubal and J. Meidanis. *Introduction to Computational Biology.* PWS Publishing Company, 1997. → 2

[180] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. In *Proceedings of the 28th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, number 2573 in LNCS, pages 379–390. Springer, 2002. → 168

[181] R. Sharan and R. Shamir. CLICK: A clustering algorithm with applications to gene expression analysis. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 307–316. AAAI Press, 2000. → 168

[182] R. Sharan and R. Shamir. Algorithmic approaches to clustering gene expression data. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 269–300. MIT Press, 2002. → 168

[183] A. C. Siepel and B. M. E. Moret. Finding an optimal inversion median: experimental results. In *Proceedings of the First Workshop on Algorithms in Bioinformatics (WABI)*, number 2149 in LNCS, pages 189–204. Springer, 2001. → 5, 135

[184] M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9:91–116, 1992. → 87, 88

[185] M. Steel. Personal communication, May 2001. → 86

[186] U. Stege. Resolving conflicts in problems from computational biology. PhD thesis, diss. no. 13364, Department of Computer Science, ETH Zürich, 2000. → 22, 33

[187] U. Stege and M. Fellows. An improved fixed-parameter-tractable algorithm for Vertex Cover. Technical Report 318, Department of Computer Science, ETH Zürich, Swizerland, April 1999. → 43

[188] N. Stojanovic, P. Berman, D. Gumucio, R. Hardison, and W. Miller. A linear-time algorithm for the 1-mismatch problem. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS)*, number 1272 in LNCS, pages 126–135. Springer, 1997. → 46, 47, 48, 56

[189] N. Stojanovic, L. Florea, C. Riemer, D. Gumucio, J. Slightom, M. Goodman, W. Miller, and R. Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 1999. → 47, 56, 63

[190] K. Strimmer and A. von Haessler. Quartet puzzling: a quartet maximum-likelihood method for reconstructing tree topologies. *Molecular Biology and Evolution*, 13(7):964–969, 1996. → 86, 89

[191] L. Stryer, J. M. Berg, and J. L. Tymoczko. *Biochemistry, 5th Ed.* W. H. Freeman, 2002. → 11

[192] A. Tanay, R. Sharan, and R. Shamir. Discovering statistically signifnicant biclusters in gene expression data. In *Proceedings of the 10th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, volume 18 (Supplement 1) of *Bioinformatics*, pages S136–S144. Oxford University Press, 2002. → 4, 169

[193] M. Tang, M. Waterman, and S. Yooseph. Zinc finger gene clusters and tandem gene duplication. *Journal of Computational Biology*, pages 429–446, 2002. → 157, 174

[194] G. Della Vedova and H. T. Wareham. Optimal algorithms for local vertex quartet cleaning. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC)*, pages 173–177. ACM Press, 2002. → 88, 100, 161

[195] S. Vialette. Pattern matching problems over 2-interval sets. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, number 2373 in LNCS, pages 53–63. Springer, 2002. → 140, 141

[196] L. Wang, T. Jiang, and D. Gusfield. A more efficient approximation scheme for tree alignment. *SIAM Journal on Computing*, 30(1):283–299, 2000. → 157

[197] Z. Wang and K. Zhang. RNA secondary structure prediction. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 345–364. MIT Press, 2002. → 14

[198] M. Waterman. *Introduction to Computational Biology.* Chapman & Hall, 1995. → 2

[199] M. Weiß. Molecular investigations on phylogeny in the genus Amanita. PhD thesis, Fakultät für Biologie, Universität Tübingen, Fed. Rep. of Germany, 1999. → 103

[200] M. Weiß, Z. Yang, and F. Oberwinkler. Molecular phylogenetic studies in the genus Amanita. *Canadian Journal of Botany*, 76:1170–1179, 1998. → 103, 104

Lebens- und Bildungsgang

Name: **Jens Gramm**
Geboren: 3.11.1972 in Radolfzell am Bodensee
Familienstand: verheiratet

1979–1983 Besuch der Grundschule in Stockach

1983–1992 Besuch des Nellenburg-Gymnasiums in Stockach

Mai 1992 Abitur

7/1992–9/1993 Zivildienst beim Roten Kreuz, Konstanz

10/1993–10/1999 Studium der Informatik an der Universität Tübingen

8/1996–7/1997 Auslandsstudium an der University of Massachussetts, Amherst, USA

4/99 Studienarbeit (Betreuer Prof. K.-J. Lange) über

"Sytactic Recognition of NL-Complete Problems"

10/1999 Diplom in Informatik, Diplomarbeit (Betreuer Prof. K.-J. Lange, PD R. Niedermeier) über

"Exact Algorithms for Max2Sat with Applications"

seit 1/2000 Promotion an der Fakultät für Informations- und Kognitionswissenschaften, Lehrstuhl für Theoretische Informatik/Formale Sprachen (Prof. K.-J. Lange), Betreuer PD R. Niedermeier

Publikationen: [2, 3, 70, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 92]