Fixed-parameter Algorithms for some Combinatorial Problems in Bioinformatics

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik

der Friedrich-Schiller-Universität Jena

von Dipl.-Inf. Quang Bao Anh Bui

geboren am 25. Feb. 1979 **in** Ho Chi Minh Stadt, Vietnam

Gutachter:

1. **Prof. Dr. Sebastian Böcker, Friedrich-Schiller-Universität Jena**
2. **Dr. Andreas Hildebrandt, Universität des Saarlandes**

Tag der öffentlichen Verteidigung: 11. Januar 2011

Gedruckt auf alterungsbeständigem Papier nach DIN-ISO 9706

Zusammenfassung

NP-schwere Probleme kommen in nahezu jedem Bereich in der Bioinformatik vor. Zudem sind biologische Probleminstanzen meistens sehr groß. Dadurch scheitern die klassische Ansätze zur exakten Lösung NP-schwere Probleme in der Biologie.

Fest-parametrisierte Algorithmik wurde in den neunziger Jahren als ein neuer Ansatz entwickelt, um exakte Lösungen NP-schwerer Probleme zu berechnen. Die Hauptidee von fest-parameterisierter Algorithmik besteht darin, die kombinatorische Explosion des Lösungsraums eines NP-schweren Problems auf einen kleinen Eingabeparameter anstatt der Größe der zu lösenden Probleminstanz einzuschränken. Somit bietet fest-parametrisierte Algorithmik eine Möglichkeit, NP-schwere Probleme mit großen Instanzen zu lösen. In dieser Arbeit wenden wir fest-parametrisierte Algorithmen an, um drei NP-schwere Probleme in der Bioinformatik zu lösen:

- Flip Consensus Tree Problem ist ein Spezialfall des Flip Supertree Problem, das von dem Flip-Supertree-Ansatz [38] hervorgerufen wurde. Wir untersuchen das Flip Consensus Tree Problem anhand einer graph-theoretischen Formulierung dieses Problems, das sucht nach der minimalen Anzahl der Kanten-Modifikationen, um einen bipartite Graphen *M-frei* zu machen. Wir präsentieren eine Reihe von Datenreduktionsregeln und zwei Suchbaum-Algorithmen für dieses Problem. Unsere Algorithmen sind fest-parameterisiert mit der minimalen Anzahl der Kanten-Modifikationen. Zudem diskutieren wir verschiedene heuristische Verbesserungen unserer Algorithmen. Wir berichten auch über unsere Untersuchungsergebnisse auf phylogenetischen Daten.

- Weighted Cluster Editing Problem ist ein graph-modifikation Problem, das nach den minimalen Kosten der Kanten-Modifikationen sucht, um einen Graphen in einen Cluster-Graph umzuwandeln. Dieses Problem tritt in der Bioinformatik auf, wenn eine Aufteilung von Objekten im Bezug auf eine bestimmte Ähnlichkeit oder Distanz erforderlich ist. Wir beschreiben einen Suchbaum-Algorithmus für dieses Problem, der auch in [16, 18] zu finden ist. Dieser Algorithmus ist fest-parameterisiert mit der minimalen Anzahl der Kanten-Modifikationen. Außerdem präsentieren wir die Hauptidee unseres neuen Algorithmus, der die beste theoretische Laufzeit für das Weighted Cluster Editing Problem und das Unweighted Cluster Editing Problem hat.

- Bond Order Assignment Problem sucht nach eine Zuordnung der Bindungsordnungen eines Moleküls, die eine Bestrafungsfunktion minimiert. Wir zeigen, dass das Bond Order Assignment Problem NP-schwer und und unter der

Annahme P≠NP nicht approximierbar ist. Zudem zeigen wir die MAX SNP-Schwierigkeit der Maximierungsversion des Bond Order Assignment Problem. Wir stellen zwei dynamische Prgrammierungsalgorithmen für dies Problem vor, die eine optimale Zuordnung der Bindungsordnungen eines Moleküls anhand dessen Baumzerlegung berechnen. Unsere Algorithmen sind fest-parameterisiert mit der Baumweite des Molekülgraphen und der maximalen Atomvalenzen. Wir haben einen unserer Algorithmen mit mehreren heuristischen Verbesserungen implementiert. Die Evaluation unserer Implementierung auf eine Reihe von realen Moleküle hat ergeben, dass unser Algorithmus sehr schnell auf diese Datensatz ist. Insbesondere ist unser Algorithmus sogar schneller als der heuristischer Algorithmus aus [152] für das Bond Order Assignment Problem.

Abstract

NP-hard problems occur often in bioinformatics. The typical huge sizes of biological problem instances often prohibit solving NP-hard problems in bioinformatics optimally with classical approaches.

Fixed-parameterized algorithmics has been developed in 1990s as a new approach to solve NP-hard problem optimally in a guaranteed running time. The essential idea of fixed-parameter algorithms is to restrict the combinatorial explosion of the solution space of an NP-hard problem to a small input parameter, instead of the size of the problem instance. Thus, fixed-parameter algorithms offer a new opportunity to solve NP-hard problems with large instances.

In this thesis, we apply fixed-parameter algorithms to cope with three NP-hard problems in bioinformatics:

- Flip Consensus Tree Problem is a special case of the Flip Supertree Problem, a combinatorial problem arising in *computational phylogenetics*. Using a graph-theoretical formulation of the Flip Consensus Tree Problem that asks for a minimum set of edge modifications to transform a bipartite graph into an *M-free* bipartite graph, we present a set of data reduction rules and two depth-bounded search tree algorithms for this problem that are fixed-parameter with respect to the minimum number of edge modifications. Additionally, we discuss several heuristic improvements to accelerate the running time of our algorithms in practice. We also report computational results on phylogenetic data.

- Weighted Cluster Editing Problem is a graph-theoretical problem, that asks for a set of *edge modifications* with minimum cost to transform a graph into a *cluster graph*. This problem often arises in computational biology when clustering objects with respect to a given similarity or distance measure is required. We present one of our depth-bounded search tree algorithms for this problem that is a fixed-parameter algorithm with respect to the minimum modification cost. We also describe the main idea of our fastest algorithm [16, 18] for the Weighted Cluster Editing Problemand Unweighted Cluster Editing Problem.

- Bond Order Assignment Problem asks for a bond order assignment of a molecule graph that minimizes a penalty function. We show that the Bond Order Assignment Problem is NP-hard and inapproximable unless P=NP. Furthermore, we show that the maximization version of Bond Order Assignment Problem is MAX SNP-hard. We then give two exact fixed-parameter algorithms for the problem, where bond orders are computed via dynamic programming on a tree decomposition of the molecule graph. Our algorithms are fixed-parameter

with respect to the treewidth of the molecule graph and the maximum atom valence. We implemented one of our algorithms with several heuristic improvements and evaluate our algorithm on a set of real molecule graphs, that is known to contain hard instances for the Bond Order Assignment Problem. It turns out that our algorithm is very fast on this dataset. Particularly, our algorithm even outperforms the heuristic algorithm introduced in [152] for the Bond Order Assignment Problem.

Acknowledgements

This thesis would not have been possible without support of many people. It is difficult for me to find the right expression of gratitude for all of them.

First and foremost, I would like to thank Prof. Sebastian Böcker for being a great supervisor, who has been extremely supportive with lots of valuable ideas and stimulating discussions. Without his ideas, comments and contributions, this thesis would not exist. Moreover, I have profited so many experiences in doing scientific research under his guidance.

I gratefully acknowledge financial support from the Deutsche Forschungsgemeinschaft (DFG) within the project *Parameterized Algorithms in Bioinformatics* (BO 1910/5).

My thanks also go to all members of the Chair Bioinformatics at the Universität Jena, including Malte Brinkmeyer, Thasso Griebel, Franziska Hufsky, Florian Rasche, Imran Rauf, Kerstin Scheubert, Anke Truss and Sascha Winter for an enjoyable working atmosphere. Particularly, I would like to thank Anke Truss for many fruitful discussions and for being a great co-author in most of my publications. I also appreciate my office neighbor Florian Rasche, from whom I have learned lots of biological knowledge, which I could not gather during my undergraduate study in computer science. I would like to thank Malte Brinkmeyer and Thasso Griebel for sharing their knowledge in phylogenetics. A special thank goes to Kathrin Schowtka, our secretary, who always helps with administrative work. I also thanks my student assistants Kai Dührkop and Patrick Seeber for implementing of our algorithms.

I wish to acknowledge Anna Katharina Dehof and Dr. Andreas Hildebrandt for introducing us to the Bond Order Assignment Problem, from which I achieve a substantial result in this thesis. I would like to thank Anna Katharina Dehof and Johannes Uhlmann for helpful discussions concerning this problem.

I heartly appreciate Malte Brinkmeyer, Thasso Griebel, Florian Rasche, Anke Truß, Sascha Winter and Hoi-Ming Wong not only for proofreading parts of this thesis, but also for their advices to improve its legibility.

Finally, I would like to thank my wife Hai Yen Do for her understanding and love during the last three years. I am so grateful to my grandparents, and other members of my big family, who were, are and will always be there for me, whenever necessary. My parents Quang Ly Bui and Thi Thanh Nguyen receive my deepest gratitude and love for everything.

Contents

1 Introduction **3**

2 Computational Background **7**
2.1 Graph Theory . . . 7
2.2 Combinatorial Problems and their Complexities . . . 8
2.2.1 In P versus NP-hard . . . 10
2.2.2 Classical Approaches for NP-hard Problems . . . 11
2.3 Fixed-Parameter Tractability . . . 12
2.3.1 Depth-bounded Search Tree and Graph Modification . . . 15
2.3.2 Tree Decomposition-based Dynamic Programming . . . 17
2.3.3 Data Reduction and Kernelization . . . 21

3 Biological Background **23**
3.1 Basic Chemistry . . . 23
3.2 Fundamental Molecular Biology . . . 24
3.2.1 Biomolecules . . . 25
3.2.2 Gene Expression . . . 25
3.3 Computational Phylogenetics . . . 27

4 Flip Consensus Tree **31**
4.1 Motivation . . . 31
4.2 Previous Work . . . 34
4.3 Graph-theoretical Model . . . 34
4.4 Algorithms . . . 36
4.4.1 Data Reductions . . . 38
4.4.2 The $O(4.83^k + poly(m, n))$-Algorithm . . . 41
4.4.3 The $O(4.42^k + poly(m, n))$-Algorithm . . . 46
4.5 Algorithm Engineering . . . 52
4.6 Computational Results . . . 53
4.7 Summary and Outlook . . . 54

5 Weighted Cluster Editing **57**
5.1 Motivation . . . 57
5.2 Previous Work . . . 58
5.3 Algorithms . . . 60
5.3.1 A Straightforward Algorithm . . . 60
5.3.2 Refined Branching Strategy . . . 62

5.4 Further Results . . . 66
5.5 Summary and Outlook . . . 67

6 Bond Order Assignment 69
6.1 Motivation . . . 69
6.2 Previous Work . . . 69
6.3 The Bond Order Assignment Problem . . . 70
6.4 Hardness of the Bond Order Assignment Problem . . . 72
6.5 Polynomial Algorithm on Trees . . . 76
6.6 Algorithms for Molecule Graphs of Bounded Treewidth . . . 78
6.6.1 The $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$ Algorithm . . . 78
6.6.2 The $O(\alpha^{3\omega} \cdot \omega \cdot m)$ Algorithm . . . 81
6.7 Algorithm Engineering . . . 84
6.8 Computational Results . . . 85
6.8.1 Implementation and Dataset . . . 85
6.8.2 Comparison with other approaches . . . 86
6.9 Summary and Outlook . . . 88

7 Conclusion 91

Preface

This thesis covers a substantial part of my research concerning fixed-parameter algorithms for combinatorial problems in bioinformatics while working at the Bioinformatics Group of Professor Sebastian Böcker at the Friedrich-Schiller Universität Jena. My research was supported by the project "PABI" funded by the Deutsche Forschung Gemeinschaft (DFG). Most of the results presented in this work have been published in [16–18, 22–24] and have been achieved in cooperation with my supervisor Professor Sebasitian Böcker, my colleague Anke Truß, our former diploma student Sebastian Briesemeister, and our former student assistant Patrick Seeber. Further publications [21, 49], on which I am collaborating, are in preparation.

This thesis consists of seven chapters. The main results are presented in Chapter 4, 5 and 6.

In Chapter 4, I describe two fixed-parameter algorithms for the Flip Consensus Tree Problem with running time $O(4.83^k + poly(m,n))$ and $O(4.42^k + poly(m,n))$. The algorithm with running time $O(4.83^k + poly(m,n))$ has been presented by me at the *3rd International Workshop on Exact and Parameterized Computation* (IWPEC 2008) [23]. Most of the data reduction rules described in Section 4.4.1 were done by me. The algorithm with running time $O(4.83^k + poly(m,n))$ was developed by Anke Truss and me. The algorithm with running time $O(4.42^k + poly(m,n))$ is accepted for publication in *Association for Computing Machinery Transactions on Algorithms* [24]. This algorithm was developed by me. Implementation of the algorithm with running time $O(4.83^k + poly(m,n))$ was done by our student assistant Patrick Seeber and supervised by me.

In Chapter 5, I describe an algorithm with running time $O(2.42^k + |V|^3 \log |V|)$ for the Weighted Cluster Editing Problem, which was published in [16]. My main contribution to [16] is the idea of splitting case (C1) into (C11) and (C12), as described in Section 5.3.2.

Chapter 6 investigates the Bond Order Assignment Problem. Parts of the results introduced in this chapter have been presented by me at the *15th International Computing and Combinatorics Conference* (COCOON 2009) [22]. A further part of this chapter is submitted to the journal of *Theoretical Computer Science* [25]. The main contribution of [22] consists of an NP-hardness proof and a tree decomposition-based dynamic programming algorithm for the Bond Order Assignment Problem with running time $O(\alpha^{2\omega} \cdot 3^\beta \cdot \omega \cdot m)$. The NP-hardness proof was done by me. The tree decomposition-based dynamic programming algorithm for this problem was developed by my supervisor and me. The first implementation of our algorithm was done by our student assistant Patrick Seeber and supervised by me. In addition to our results in [22], we introduced several further complexity results and an improved tree-decomposition-based

algorithm with running time $O(\alpha^{3\omega} \cdot \omega \cdot m)$, in our manuscript submitted to the journal of *Theoretical Computer Science*. Those complexity results have been done mainly by me. The improved version of our algorithm was developed by my supervisor and me. Computational results reported in this thesis are computed by the new implementation of our algorithm done by our student assistant Kai Dürkop and supervised by me.

1 Introduction

Bioinformatics applies mathematical and computational approaches to deal with problems occurring in biology, such as exploring relationships among biological entities in a large data set, locating a gene within a long DNA sequence, predicting structure and function of proteins, clustering genes or protein sequences into families of related sequences, just to name a few. After formulating a biological problem as a mathematical model, the remaining task is to develop an algorithm to solve the problem. Since biological data are often discrete nature, mathematical models of biological problems usually rely on discrete structures such as graphs or strings. In this work, we investigate algorithms for solving biological problems that have already been formulated as mathematical models; and whenever we refer to a *biological problem*, we mean the mathematical formulation of the problem.

Actually, most of biological problems require large amount of computation resource (time and memory requirement) to be solved optimally, in particular when dealing with practical relevant problem instances. Prominent examples include Multiple Sequence Alignment [54, 153], Shortest Common Super Sequence [65], Motif Finding [2, 131] and many phylogenetic tree reconstruction approaches [38, 42, 43, 60, 135]. In fact, these problems and many other biological problems belong to the class of NP-hard problems (see Section 2.2), for which it is widely believed that efficient algorithms, whose running times are bounded by polynomial functions in sizes of the problem instances, may not exist.

For a long time, algorithms with exponential running times in sizes of the problem instances seemed to be the only way to solve NP-hard problems optimally with guaranteed running times. Since biological problem instances of interest are typically very large, exponential running time algorithm usually cannot be applied to NP-hard problems arising in biology. Thus, after showing the NP-hardness of a problem, one usually applies *heuristic* or *approximation algorithms* (see Section 2.2.2) to estimate good solutions, rather than to solve the problem optimally. In many cases, biologists are also satisfied to have a good solution of an NP-hard problem, although an optimal solution is naturally preferable to an approximated result.

In the 1990s, Downey and Fellows introduced the concept of *parameterized complexity* and *fixed-parameter algorithms* [52] as a new approach to solve NP-hard problems optimally in provable upper bounded running times. The most valuable characteristic of fixed-parameter algorithms is their polynomial running times in the sizes of the problem instances. However, an exponential running time seems to be unavoidable when solving an NP-hard problem optimally, even by a fixed-parameter algorithm. In fact, the running time of a fixed-parameter algorithm contains an exponential factor but in a problem-specific parameter, which is usually much smaller than the sizes of problem

instances. Polynomial running times in the sizes of the problem instances are the main reason for the efficiency of fixed-parameter algorithms on large problem instances. This enables fixed-parameter algorithms to deal with the large instances of NP-hard problems in biology.

Moreover, when analysing the running time of an algorithm, we usually investigate its worst-case running time, which only holds for carefully designed worst-case input examples, or its average-case running time, which only indicate the behavior of the algorithm on randomly generated data. However, biological data usually possess particular structures, that are totally different from the artificial worst-case examples and random data. Utilizing the particular structure of biological data may help to reduce running time in both theoretical analysis and practical application.

In fact, fixed-parameter algorithms have been successfully applied to many NP-hard problems in bioinformatics, for example:

- Xu *et al.* [158–160] successfully applied fixed-parameter algorithms to solve NP-hard problems encountered when studying protein structures.
- Blelloch *et al.* [15] introduced a fixed-parameter algorithm for computing maximum parsimonious phylogenetic trees.
- Hüffner *et al.* [87, 88] and Bruckner *et al.* [32] applied fixed-parameter approaches to solve problems encountered when studying biological networks.
- Ma and Sun [112] introduced a fixed-parameter algorithm for the Closest String Problem, which, for example, appears in the context of finding approximate gene clusters.
- Möhl *et al.* [118] proposed a parameterized algorithm for the RNA Structures Alignment problem that aligns two RNA structures with pseudoknots in matter of hours.
- Böcker *et al.* [18,20] successfully applied fixed-parameter algorithm to the Weighted Cluster Editing Problem.

See [89] for a survey of fixed-parameter algorithms in bioinformatics.

In this thesis, we investigate three biological problems, the Flip Consensus Tree Problem, the Weighted Cluster Editing Problem and the Bond Order Assignment Problem, under the aspect of fixed-parameter tractability. In addition to the development of fixed-parameter algorithms for those problems, we are also interested in the practical application of our algorithms. Thus, we implemented our algorithms and evaluated their performance on real biological data.

Structure of the Thesis

This thesis consists of seven Chapters. Chapter 2 introduces terminologies from graph theory and theoretical computer science that are frequently used throughout this thesis.

In particular, Chapter 2 also gives a brief introduction to the basic concepts of fixed parameter algorithmics and several fixed-parameter algorithm design techniques, which we applied to develop our algorithms. In Chapter 3, we give a short introduction to molecular biology and computational phylogenetics.

In Chapter 4, we study the Flip Consensus Tree Problem as a special case of the Flip Supertree Problem that is encountered when reconstructing phylogenetic trees using the *flip supertree* approach introduced by Chen *et al.* [38]. We introduced two fixed-parameter algorithms for the Flip Consensus Tree Problem along with a set of data reduction rules and heuristic improvements to speed up the performance of our algorithms in practice. We implemented one of our algorithms and evaluated its performance on perturbed matrix representations of phylogenetic trees. Our algorithm turned out to be significantly faster than the straightforward algorithm introduced by Chen *et al.* [38] and also much faster than the worst-case running time suggests.
Actually, we wanted to solve the Flip Supertree Problem optimally with a fixed-parameter algorithm. Unfortunately, we recently proved that the Flip Supertree Problem is W[2]-hard [21], that means there is no hope for a fixed-parameter algorithm for this problem. However, we discuss some approaches that may cope with the W[2]-hardness of the Flip Supertree Problem in the end of Chapter 4.

The Weighted Cluster Editing Problem is encountered in many biological studies when classifying objects according to a similarity measure. Our group has achieved many remarkable results with respect to fixed-parameter tractability on this problem. Chapter 5 gives a survey of our results on the Weighted Cluster Editing Problem. We also describe one of our algorithms in detail. However, we refer to [18] for our main results regarding the Weighted Cluster Editing Problem.

The last problem we investigate in this thesis is the Bond Order Assignment Problem, that asks for an assignment of bond orders to a molecule graph, where bond order information is omitted. In Chapter 6, we prove several results on the complexity of the Bond Order Assignment Problem. We then show that the Bond Order Assignment Problem can be solved in polynomial time if the molecule graph is a tree. Based on this result, we introduced two fixed-parameter algorithms for the Bond Order Assignment Problem on molecule graphs with *bounded treewidth*. We implemented one of our algorithms and evaluated its performance on the MMFF94 molecule database, which is known to contain hard instances for the Bond Order Assignment Problem. Despite the unimpressive theoretical running time, our algorithm turns out to be very efficient on this database due to a special structure of molecule graphs. Furthermore, our evaluation on more than hundred thousand molecule graphs randomly chosen from the PubChem database also possess this special structure, which allows for the efficiency of our algorithms.

Chapter 7 concludes this thesis by recalling our main results and summarizing some general techniques of fixed-parameter algorithms that we used to achieve our results and plan to apply to the open problems mentioned along this thesis.

2 Computational Background

The main purpose of this chapter is to provide the necessary knowledge about *fixed-parameter tractability* to readers who are not familiar with this concept. Furthermore, we give a short introduction to the NP-hardness of combinatorial problems as a motivation for the fixed-parameter tractability concept. Interested readers may have a look at [45, 65, 154] for more details on complexity theory. A detailed introduction to fixed-parameter tractability concept can be found in [52, 59, 121].

2.1 Graph Theory

This section introduces the basic terminology and notation from graph theory that is frequently used in this work.

Definition 2.1 (Graph)**.** An *undirected graph* $G = (V, E)$ consists of a finite vertex set V and an edge set $E \subseteq \binom{V}{2}$, where $\binom{V}{2}$ denotes the collection of all two-element subsets of V. An edge $e = \{u, v\}$ connects vertices u and v. If the order of the vertices of every edge in G is fixed, we say that G is a *directed graph* and denote an edge e connecting a vertex u with a vertex v by an ordered pair $e = (u, v)$. In an undirected graph, we write $e = uv$ instead of $e = \{u, v\}$ for brevity.

In this work, we mainly deal with undirected graph. Thus, the following notation is defined on undirected graphs. However, we will explicitly mention directed graph at appropriate points, where it is necessary.

Vertices belonging to an edge are *end vertices* of the edge. Two vertices connected by an edge are called *adjacent* to each other and *incident* to the edge.

Two adjacent vertices are *neighbors* of each other. We denote the *neighborhood* of a vertex v by $N(v) := \{u \in V \mid uv \in E\}$. The *degree* of a vertex v is defined as the number of its neighbors and is denoted by $\deg(v) := |N(v)|$.

Definition 2.2 (Subgraph)**.** A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V'$ and $E' \subseteq E'$. A subgraph G' of a graph G is an *induced subgraph* of G if $E' = E \cap \binom{V'}{2}$ holds .

We denote an induced subgraph of G that consists of vertices in $V' \subseteq V$ by $G[V']$.

A *length-l path* is a sequence $p := (v_{p_1}, v_{p_2}, \ldots, v_{p_l})$ of l vertices, where two subsequent vertices are connected by an edge. A path is called *simple* if the vertices are pairwise different. Vertex v_{p_1} is the *starting point* of p and v_{p_l} is the *end point* of p.

A *connected component* of a graph is a subgraph of the graph, in which every pair of two vertices are connected by a path.

A *cycle* is a path whose starting point and end point are identical. A graph that does not contain any cycle as a subgraph, is an *acyclic* graph.

Definition 2.3 (Tree)**.** An undirected graph is a *tree* if it is acyclic.

A vertex of a tree is usually referred to as a *node*. A node of degree one is called a *leaf*. Nodes of degree at least two are *inner nodes*.

A tree whose inner nodes have degree tree is a *binary tree*.

A *rooted* tree is a tree in which one node is distinguished from other nodes, and designated as the *root* of the tree. Edges of a rooted tree have a natural direction, either all *towards* or all *away* from the root. In a rooted tree, the *parent* (node) of a node v is a node u if $(u, v) \in E$. The node v is a *child* of its parent node u. A leaf does not have child nodes and the root does not have parent. The *depth* of a rooted tree is the length of the longest path from the root to a leaf of the tree.

We conclude this section with a set of several special graphs that are used in this work.

- A graph $G = (V, E)$ is a *bipartite graph* if V can be partitioned into two disjoint subsets V_1 and V_2 such that each edge of G connects a vertex in V_1 with a vertex in V_2.
- An undirected graph $G = (V, E)$ is *complete* if it holds for every two vertices $u, v \in V$ that $uv \in E$.
- A complete subgraph is called a *clique*.
- An undirected graph $G = (V, E)$ is a *cluster graph* if every connected component of G is a clique.

2.2 Combinatorial Problems and their Complexities

Combinatorial problems are problems that deal with discrete structures. The tasks of combinatorial problems include counting structures of a given property, generating structures with a given property, searching for structure with a given property or searching for a structure optimizing an objective function. In this work, we focus on combinatorial problems on graphs that search for structures that optimize given *objective functions*. Such problems are usually referred to as *combinatorial optimization problems* in literature [46, 65, 96].

To solve a combinatorial problem, we need an appropriate algorithm. The *running time* of an algorithm is defined as the maximum number of operations executed by the algorithm to solve a problem instance. However, the number of those operations not only depends on the size of the problem instance, but also on the computational model, on which the algorithm works. In this work, we use the *Random Access Machine* (RAM) [141] as our model of computation. RAM is a theoretical computational model that is on the one hand similar to common computational models, and on the other hand sufficiently universal to be considered as a machine-independent model.

Random Access Machine. RAM is a simple model of computation, it supports a set of basis operations such as addition, multiplication, load, store, if-statements, etc. Each of those operations can be executed in one time unit. Assuming RAM as computational model, algorithms are described in pseudo-code. Furthermore, every algorithm on a common computational models can be simulated by RAM. In particular, the number of operations executed by an algorithm on a concrete computer differs from the number of operations executed by the same algorithm on a RAM by at most a constant factor [141].

In computer science, the running time of an algorithm is often specified by a function in the size of the problem instance, which is usually measured with the number of objects that are contained in the problem instances, for example, the number of vertices in an input graph or the number of elements in a countable set. However, in some cases, one also considers the bit-length of the problem instance as its size, for example, in case of the *primality testing* problem that asks if a given number is a prime number.

Since constant factors in the running time of an algorithm are usually ignored, *big-O-notation* is used to upper-bound the running time of an algorithm.

Big-O-Notation. Let f and g be two functions over $\mathbb{R}^+$, the set of positive real numbers. We write $f = O(g)$ if there exist two constants x_0, $c > 0$ such that $f(x) \leq c \cdot g(x)$ for $x \geq x_0$. Informally, $f = O(g)$ can be interpreted as "f does not grow asymptotically faster than g". Big-O-notation is one of the Landau notations, that are widely used in computer science to investigate running time of algorithms or complexity of problems. See [46, 65, 96] for other Landau notation. However, in this work, we only use the big-O-notation as described above.

The following terminologies are frequently used in this work.

- We say that the running time of an algorithm is $O(f(n))$ if the algorithm executes at most $O(f(n))$ operations to solve the problem instance of size n. The running time of an algorithm is also referred to as its *complexity* or more precisely *time complexity*.
- If the running time of an algorithm is bounded by $n^{O(1)}$, where n is the size of the problem instance, we say that the algorithm is a *polynomial time algorithm*.
- If there is a polynomial time algorithm for a combinatorial problem, we say that the problem is *solvable in polynomial time*.
- The *complexity of a problem* is the complexity of the most efficient algorithm that solves the problem.

In general, polynomial time algorithms run faster than superpolynomial time algorithms. Therefore, given a combinatorial problem, it is preferable to have a polynomial time algorithm for that problem. In fact, for many combinatorial problems, there are polynomial time algorithms available. In textbooks such as [46,96,101], polynomial time algorithms for a number of problems are described systematically in detail. However, for many other combinatorial problems, there is no polynomial time algorithm. Moreover,

there is a class of problems, called NP-hard problems, for which it is widely believed that polynomial time algorithm cannot exist.

In the next section, we give an over-simplified introduction to NP-hardness concept and techniques for showing the NP-hardness of a problem. That is a small basic part of *complexity theory*, a research field in computer science, that focuses on classifying problems according to their complexities. For more information on complexity theory, we refer interested readers to relevant textbooks, e.g. [65, 82, 85, 125, 154].

2.2.1 In P versus NP-hard

Actually, solving a combinatorial problem can be subdivided into two parts: finding a solution and verifying the correctness of the solution. The class of problems for which the second part can be done in polynomial time is referred to as NP (Nondeterministic Polynomial time solvable), whereas the class of problems, for which both parts can be done in polynomial time is referred to as P (Polynomial time solvable). It is obviously that $P \subseteq NP$.

However, the question if $P = NP$ is still a long-open question in the complexity theory, since the class NP does contain some problems, for which it is not known if there is polynomial time algorithm available. Although it is widely believed that there will be no polynomial time algorithm for such problem, there is no proof for that conjecture. For example, the Hamiltonian Cycle Problem, which asks if an undirected graph contains a cycle that visits every vertex once, is in *NP* since we can easily verify if a cycle visits every vertex of a graph once. However, we do not know if the Hamiltonian Cycle Problem is in P, since we do not have an algorithm that can find such a cycle in polynomial time.

There is a class of problems, for which no polynomial time algorithm is available and it is proved that if there is a polynomial time algorithm for one of those problem, then every problem in this class is solvable in polynomial time. Problems in this class are called *NP-hard problems*. An NP-hard problem is said to be *NP-complete*, if it is in NP. Thus, the class of NP-complete problems are the class of hardest problem in NP.

If we face a problem, for that we cannot find or develop a polynomial time algorithm, the next step that we can do is trying to show that the problem is NP-hard. So we need a technique to prove the NP-hardness of a problem.

Polynomial time reduction. In essence, a *polynomial time reduction* from a problem A to a problem B is an algorithm that solves problem A instance in polynomial time, assuming that there is a polynomial time algorithm for problem B. It holds, if problem A is an NP-hard problem and there is a polynomial time reduction from problem A to problem B, then problem B is also an NP-hard problem. Thus, to prove the NP-hardness of a problem B, we have to show a polynomial time reduction from an NP-hard problem to B.

The first NP-hardness proof was done by Cook in 1971 [44] as he proved the NP-hardness of Satisfiability Problem (SAT), that asks if a logical formula in conjunctive normal form is satisfiable. Afterwards, the NP-hardness of many of other combina-

torial problems was proved by polynomial time reductions to SAT and to other NP-hard problems. See [65] for a list of NP-hard problems.

Since it is widely believed that there is no polynomial time algorithm for NP-hard problems, we can safely stop searching for polynomial algorithm for a problem if we can prove that the problem is NP-hard. However, there are several approaches to cope with the NP-hardness of combinatorial problems. In Section 2.2.2 and 2.3, we describe some approaches that can be used to cope with the NP-hardness of combinatorial problems.

2.2.2 Classical Approaches for NP-hard Problems

NP-hard problems arise in almost every research area and applications in computer science [65], for examples, in graph theory, set theory, network design, sequencing and scheduling, and in bioinformatics like the problems we deal with in this work. Since it is unlikely that there will be polynomial time algorithms for NP-hard problems, the following approaches are usually applied to deal with the NP-hardness of combinatorial problems:

Heuristic. The main goal of *heuristic algorithms* is to find a good solution of the corresponding problem in a reasonable time, without any guarantee about the optimality of the computed solutions. Apart from a large number of problem-specific heuristic algorithms, there are also several general techniques to develop heuristic algorithms, the prominent representations of those techniques are heuristic branch-and-bound [48, 105] and metaheuristics [66, 147].

Approximation. An *approximation algorithm* $\mathcal{A}$ for an NP-hard optimization problem is a polynomial time algorithm that approximates the optimal solution of the problem within a guaranteed range $r(n)$, where n is the size of the problem instance. Let k_{opt} denote the value of the optimal solution, and $k_{\mathcal{A}}$ denote the value of the solution computed by $\mathcal{A}$. For a minimization problem, the guaranteed range of $\mathcal{A}$ is $k_{opt} \leq k_{\mathcal{A}} \leq r(n) \cdot k_{opt}$. For a maximization problem, the guaranteed range of $\mathcal{A}$ is $k_{opt} \geq k_{\mathcal{A}} \geq \frac{k_{opt}}{r(n)}$. We say that $r(n)$ is the *approximation factor* of $\mathcal{A}$. Obviously, there must be an n_0, such that $r(n) > 1$ for every $n \geq n_0$, otherwise $\mathcal{A}$ is a polynomial time algorithm for an NP-hard problem, which is unlikely. If $r(n)$ is a constant function, $\mathcal{A}$ is a *constant-factor approximation algorithm*. The class of optimization problems that have constant-factor approximation algorithms is referred to as APX.

If $\mathcal{A}$ approximates the optimal solution within any approximation factor $r(n) = 1 + \epsilon$ for arbitrary given number $\epsilon > 0$, $\mathcal{A}$ is called a *polynomial time approximation scheme* (PTAS). For a *fixed* ϵ, the running time of a PTAS is polynomial in n. The class of optimization problems that have polynomial time approximation schemes is also referred to as PTAS. Obviously it holds $PTAS \subset APX$. However, APX does contain some problems, for which there is no PTAS available under the assumption that $P \neq NP$ [120]. Thus, it holds $PTAS \subsetneq APX$, under the assumption that $P \neq NP$.

For many optimization problems, there is no approximation algorithm unless $P = NP$, one of the most well-known examples for inapproximability is the Traveling Salesperson Problem (TSP) that receives an undirected edge-weighted graphs as input and asks for a Hamiltonian cycle of minimum weight. It is believed that there will be no polynomial time approximation algorithm for TSP, since any polynomial time approximation algorithm for TSP will solve the NP-hard Hamiltonian Cycle Problem in polynomial time. In Section 6.4 of this work, we show that the *minimization* Bond Order Assignment Problem is inapproximable and the *maximization* Bond Order Assignment Problem is not in PTAS under the assumption that $P \neq NP$.

See textbooks [8,151] for more details on approximation algorithms and the complexity of optimization problems.

Integer Linear Programming. *Integer Linear Programming* (ILP) is a mathematical approach to optimize a linear objective function with respect to a set of linear constraints on a set of integer variables. ILP itself is an NP-hard problem [95]. Many optimization problems can be formulated as ILPs, for instance, all optimization problems considered in this work can be formulated as ILPs: In [20], Böcker *et al.* reported that ILP-approach in combination with some preprocessing procedures (called *data reduction*, see Section 2.3.3) can be applied to solve Weighted Cluster Editing Problem efficiently in practice. In [41], Chimani *et al.* solved the Flip Supertree Problem by applying an ILP approach. In [50], Dehof *et al.* used an ILP approach to solve the Bond Order Assignment Problem. In contrast to heuristic and approximation algorithms, ILP approaches solve combinatorial optimization problems exactly, but without any guarantee about the time requirement.

Exponential Running Time Algorithms. *Exponential Running time algorithms* are algorithms that solve NP-hard problems exactly, with exponential running time in the size of problem instance. Unless $P = NP$, there is no hope for polynomial time algorithms for NP-hard problems and exponential running time algorithms seem to be unavoidable when it is required to solve NP-hard problems optimally. There are a large number of exponential running time algorithms for many NP-hard problems. See [157] for a survey of exponential running time algorithms. For small problem instances, a good exponential running time algorithm may even faster than a bad polynomial running time algorithm in practice. However, exponential running time algorithms are usually inefficient when coping with the NP-hardness of large problem instances, which are usually the case in bioinformatics. In the next section, we give a brief introduction to the *fixed-parameter tractability* concept, which offers an opportunity for exactly solving large problem instances of combinatorial problems, despite their NP-hardness.

2.3 Fixed-Parameter Tractability

Under the assumption that $P \neq NP$, exponential factors in the running time of an algorithm that solves an NP-hard problem optimally, is unavoidable. However, in many

cases, it is possible to develop algorithm for a NP-hard problem whose running time is exponential in a problem-specific parameter, but polynomial in the size of problem instance. For example, in the 1980s it was independently observed by several authors that many NP-complete problems defined on graphs turn out to be solvable in polynomial time on graphs with *bounded treewidths* (see Section 2.3.2). However, the running times of the corresponding algorithms are exponential in the treewidth of input graphs [6, 14, 26].

In [52], Downey and Fellows introduced the *fixed-parameter tractability* concept as a method of algorithm design and analysis, that proposes to restrict the combinatorial explosion of search space of an NP-hard problem to a small problem-specific parameter, instead of to the size of the problem instance. The problem-specific parameter may be the treewidth of an input graph as mentioned above, or the maximum vertex degree of an input graph [93], and often an upper-bound of the optimal solution of an optimization problem, just to name a few possible parameters that are usually chosen. The main concept of fixed-parameter tractability can be formalized as following:

Definition 2.4 (Fixed-parameter Tractability)**.** A *parameterized* problem consist of an input pair (I, k), where I is the problem instance, and k is an input *parameter*. An algorithm that solves a parameterized problem in time $f(k) \cdot |I|^{O(1)}$, where $f(k)$ is a computable function independent of $|I|$, is called a *fixed-parameter algorithm*. A parameterized problem is *fixed-parameter tractable* if there is a fixed-parameter algorithm for the problem.

For NP-hard problem, $f(k)$ cannot be a polynomial function, otherwise we would have a polynomial time algorithm for an NP-hard problem. However, the parameter k is usually much smaller than the size of problem instances, so that fixed-parameter algorithms are usually much more efficient in practical use compared with exponential running time algorithms. To give an example of the superiority of fixed-parameter algorithms compared to classical exponential running time algorithm, we consider a simple exponential running time algorithm and a simple fixed parameter algorithm for the Vertex Cover Problem, which is defined as following:

Definition 2.5 (Vertex Cover Problem)**.** Given an undirected graph $G = (V, E)$, a *vertex cover* V_c of G is a subset of V, such that each edge in G is incident to at least one vertex in V_c. The Vertex Cover Problem asks for a vertex cover of G with minimum cardinality, called *minimum vertex cover*.

The Vertex Cover Problem is an NP-hard problem [65]. A trivial exponential algorithm for the Vertex Cover Problem may consider each vertex $v \in V$ with $N(v) \neq \emptyset$, recalling that $N(v)$ denotes the neighborhood of v, and distinguishes two cases:

1. *The vertex v belongs to the minimum vertex cover.* So v is inserted into the vertex cover and removed from G, edges incident to v are also removed from G. Subsequently, the algorithm calls itself recursively for the subgraph $G[V \setminus \{v\}]$.

2. *The vertex v does not belong to the minimum vertex cover.* Under this assumption, the algorithm has to include every vertex in $N(v)$ into the vertex cover. Afterwards, the algorithm calls itself recursively for the subgraph $G[V \setminus N(v)]$.

The algorithm stops if there is no vertex $v \in V$ with $N(v) \neq \emptyset$. At the end of the algorithm, each vertex of G was considered as most once and either included to a vertex cover or removed from G. In total, up to $O(2^{|V|})$ possible vertex covers are generated. Thus, the minimum vertex cover computed by the algorithm is an optimal solution of the Vertex Cover Problem. The running time of the algorithm is obviously $O(2^{|V|} \cdot |V|^{O(1)})$.

To solve the Vertex Cover Problem with fixed-parameter approach, we choose the number of vertices in a minimum vertex cover as a parameter to investigate and parameterize the Vertex Cover Problem as following:

- Instead of directly computing the minimum vertex cover of a given undirected graph G, we consider its decision version, that asks if G has a vertex cover of size at most k. This problem is a parameterized problem with respect to the parameter k.

- To solve the original Vertex Cover Problem, we have to find the minimum k, for which the parameterized Vertex Cover Problem is a yes-instance, i.e., there is a vertex cover of size k for G.

To answer if there is a vertex cover of size at most k, we slightly modified the exponential running time algorithm above by considering parameter k as a part of the input of the recursive algorithm described above. At the beginning of every recursion, we lower k by the number of vertices that were included in the vertex cover by the previous recursion and abort the recursion if $k \leq 0$. By doing this, we generate all possible vertex covers of size at most k. The running time of the modified algorithm for testing if there is a vertex cover of size at most k is obviously bounded by $O(2^k \cdot |V|^{O(1)})$. Thus, this algorithm is a fixed-parameter algorithm with respect to the input parameter k. As abovementioned, to find a minimum vertex cover, we start our modified algorithm for $k = 0$. If there is a vertex cover of size $k = 0$, the algorithm will find it. If the algorithm cannot find a vertex cover of size at most k, we call the algorithm repeatedly with increasing k until a vertex cover of size k is found. Let k_{opt} denote the size of the minimum vertex cover, we have to repeat our algorithm at most k_{opt} time until a minimum vertex cover is found. All in all, our algorithm finds an optimal vertex cover in total time $O(k_{opt} \cdot 2^{k_{opt}} \cdot |V|^{O(1)}) = O(2^{k_{opt}} \cdot |V|^{O(1)})$. Thus, the algorithm is fixed-parameter tractable with respect to the minimum vertex cover of G.

Since k_{opt} is always smaller than $|V|$, the fixed-parameter algorithm is more efficient in practice in comparison with the exponential running time algorithm. The main focus of the exponential running time algorithm and the fixed-parameterized algorithm above is to give the readers an illustration of the advances of fixed-parameter algorithm in comparison with exponential running time algorithm. In fact, Vertex Cover Problem is one of the most well-studied parameterized problems. The best known fixed-parameter

algorithm for Vertex Cover Problem is $O(1.2738^k + k\,|V|)$ where k is the size of an optimal vertex cover [39].

In general, it is preferable that an NP-hard problem is fixed-parameter tractable with respect to a small parameter. Unfortunately, many NP-hard problems are fixed-parameter tractable with respect to a certain parameter but intractable with respect to other parameters. In [52], Downey and Fellows introduced the *parameterized complexity* concept, that focuses on classifying parameterized problem according to their hardness, analogously to the concept of classical complexity theory. To understand this work, it is sufficient to know that every fixed-parameter tractable problem is said to be in the class FPT, whereas parameterized problems that are W[i]-hard, for any integer $i \geq 1$, are unlikely to be fixed-parameter tractable.

An example for a fixed-parameter intractable problem is the Independent Set Problem, which takes an undirected graph $G = (V, E)$ and an integer k as input and asks if there is an *independent set* V_I of size at least k that is a subset of V with $\binom{V_I}{2} \cap E = \emptyset$. The W[1]-hardness of this problem is proved in [52]. Recently, our group proved that the Flip Supertree Problem (see Chapter 4) is a W[2]-hard problem [21].

In the following, we give a brief introduction to the *depth-bounded search tree*, the *tree decomposition* and *data reduction* techniques with all of the related terminology that is used throughout this work. Interested readers may consider [121] for more detail on general techniques of fixed-parameter algorithm design.

2.3.1 Depth-bounded Search Tree and Graph Modification

Search tree algorithms, also called *backtracking algorithms* or *branching algorithms*, belong to the most commonly used algorithmic approaches to compute exact solution for NP-hard problems.

A search tree algorithm is a recursive algorithm that consists of a set of *branching strategies*. For a certain problem instance, an appropriate branching strategy chooses a part of the input instance and calls the algorithm recursively for each subproblem instance, in which the chosen part is assigned one of its possible values. The exponential running time algorithm described above for the Vertex Cover Problem is a search tree algorithm with only one branching strategies and can be formalized with the following recursion:

Let $VC(G)$ denote the size of the minimum vertex cover of G and v be a vertex with $N(v) \neq \emptyset$. It holds:

$$VC(G) = \min \begin{cases} VC(G[V \setminus \{v\}]), & v \text{ belongs to the minimum vertex cover.} \\ VC(G[V \setminus N(v)]), & N(v) \text{ is a subset of the minimum vertex cover.} \end{cases}$$

The execution of a search tree algorithm can be illustrated as a *search tree*, where each inner node corresponds to a recursive call and each feasible solution corresponds to a leaf of the search tree. If the depth of a search tree is upper-bounded, the corresponding search tree algorithm is a *depth-bounded search tree algorithm.*

To upper-bound the running time of a depth-bounded search tree algorithm, we need to upper-bound the size of its corresponding search tree, that is, the number of nodes of the search tree. In the following, we describe how to upper-bound the size of a depth-bounded search tree.

Let $\mathcal{A}$ be a search tree algorithm, that calls itself recursively for subproblem instances of size $n - d_1, n - d_2, \ldots, n - d_i$ when it is applied to solve a problem instance of size n. The vector $(d_1, d_2, \ldots, d_i)$ is referred to as the *branching vector* of $\mathcal{A}$. To upper-bound the size of the search tree, we first upper-bound the number of its leaves. Let $T(n)$ be the number of leaves of the search tree when solving problem instance of size n. It holds that

$$T(n) = T(n - d_1) + T(n - d_2) + \cdots + T(n - d_i).$$

The asymptotic solution of the above recursive equation is the root of the *characteristic polynomial*

$$z^d - z^{d-d_1} - \cdots - z^{d-d_i},$$

where $d = \max\{d_1, \ldots, d_i\}$ [102]. Let α be the positive root with maximum absolute value of the characteristic polynomial, then $T(n) = O(\alpha^n)$. We call α the *branching number* of $\mathcal{A}$.

Since each inner node of the search tree have at least two children, the number of leaves of the search tree is larger than the number of its inner nodes. Thus, the size of the search tree is also bounded by $O(\alpha^n)$.

If the depth of a search tree is not upper-bounded by the input size but by a parameter k, the size of the search tree is bounded by $O(\alpha^k)$. If $\mathcal{A}$ consists of several branching strategies, we compute the branching number of every strategy and choose α as the largest branching number over all branching strategies.

In this work, we apply depth-bounded search trees to the Flip Consensus Tree Problem and the Weighted Cluster Editing Problem investigated in Chapter 4 and 5. These problems belongs to the class of *graph modification problems*. In the following, we recapitulate several attributes of graph modification problems, which allow for fixed-parameter algorithms with respect to the minimum number of edge modifications.

Definition 2.6 (Graph Modification Problem)**.** A graph modification problem consists of a graph $G = (V, E)$, a desired property Π, an integer k; and asks if it is possible to modify G to fulfill property Π by at most k modifications.

A *modification*, also referred to as an *edit operation*, may be an edge deletion, edge insertion, or vertex deletion.

Definition 2.7 (Hereditary Property and Forbidden Subgraph Characterization)**.** A property Π is a *hereditary property* if every induced subgraph of a graph fulfilling the Π property also fulfills the Π property.

A property Π has a *forbidden subgraph characterization* if there is a set F of graphs, such that a graph G fulfills property Π if and only if G does not contain any graph in F as an induced subgraph. If F is a finite set, property Π is said to have a *finite forbidden subgraph characterization*.

In [34], Cai showed the following theorem.

Theorem 1. *A graph modification problem that is defined by a hereditary property with a finite set of forbidden subgraphs is fixed-parameter tractable with respect to the number of modifications.*

In the proof of this theorem, Cai [34] applied the depth-bounded search tree algorithm technique by first identifying a forbidden subgraph, and branching into every possibility to eliminate the forbidden subgraph, and calling the algorithm recursively on each modified problem instance.

Since both the Flip Consensus Tree Problem and the Weighted Cluster Editing Problem that are considered in this work admit a finite forbidden subgraph characterization, they are fixed-parameter tractable with respect to the number of modifications, as well as the minimum modification cost in case of Weighted Cluster Editing Problem (we require that the cost for every modification is at least one).

2.3.2 Tree Decomposition-based Dynamic Programming

Dynamic programming is also a standard technique to design algorithm for combinatorial problems. Dynamic programming algorithm usually can be applied to problem, whose solution can be assembled from solutions of its subproblems and minimum size problem instances can be solved easily. Thus, a dynamic programming algorithm can be described as a recursion, where initial values are available. As an example, let us consider a dynamic programming algorithm for a special case of the Vertex Cover Problem, where the input graph is a tree. Let $T = (V, E)$ be an input tree of the Vertex Cover Problem. We root T at an arbitrary node v_r. For each node $v \in V$, let $D(v)$ denote the minimum vertex cover of the subtree rooted at v and $C(v)$ denote the set of children of v. Since every edge incident to v in the subtree rooted as v have to be incident to a vertex in $D(v)$, it holds:

$$D(v) := \begin{cases} \bigcup_{u \in C(v)} (D(u) \cup \{v\}) & \text{if } \left|\bigcup_{u \in C(v)} D(u) \cup \{v\}\right| \leq \left|\bigcup_{u \in C(v)} (D(u) \cup \{u\})\right| \\ \bigcup_{u \in C(v)} (D(u) \cup \{u\}) & \text{otherwise.} \end{cases}$$

It holds $D(v) := \{v\}$ if v is a leaf of T. With the initial value at the leaves of T and the above recursion, our dynamic programming algorithm computes $D(v)$ for each node $v \in V$ after computing $D(u)$ for every $u \in C(v)$. After the course of bottom-up processing, $D(v_r)$ is the minimum vertex cover of T. The running time of this dynamic programming algorithm is bounded by a polynomial function in the size of T.

Besides Vertex Cover Problem, many other NP-hard graph problems can be solved by dynamic programming algorithms in polynomial time if the input graphs are trees. Thus, a natural question is whether polynomial running time dynamic programming algorithms for trees can be extended for "tree-like" input graphs. In [134], Robertson and Seymour presented the *tree decomposition* concept that can reflect the "tree-likeness" of graphs.

Definition 2.8 (Tree decomposition)**.** A *tree decomposition* of $G = (V, E)$ is a pair $\langle \{X_i \mid i \in I\}, T \rangle$ where I is an index set, and each X_i is a subset of V, called a bag, and T is a tree containing every bag X_i as node and the following properties hold:

1. $\bigcup_{i \in I} X_i = V$;
2. for every edge $\{u, v\} \in E$, there is a bag X_i such that $\{u, v\} \subseteq X_i$; and
3. for all $i, j, k \in I$, if X_j lies on the path between X_i and X_k in T then $X_i \cap X_k \subseteq X_j$.

The *width* of the tree decomposition $\langle \{X_i \mid i \in I\}, T \rangle$ equals $\max\{|X_i| \mid i \in I\} - 1$. The *treewidth* of G is the minimum number ω such that G has a tree decomposition of width ω. We call a tree decomposition with minimum width an *optimal tree decomposition.*

Figure 2.1 shows an example of tree decomposition.

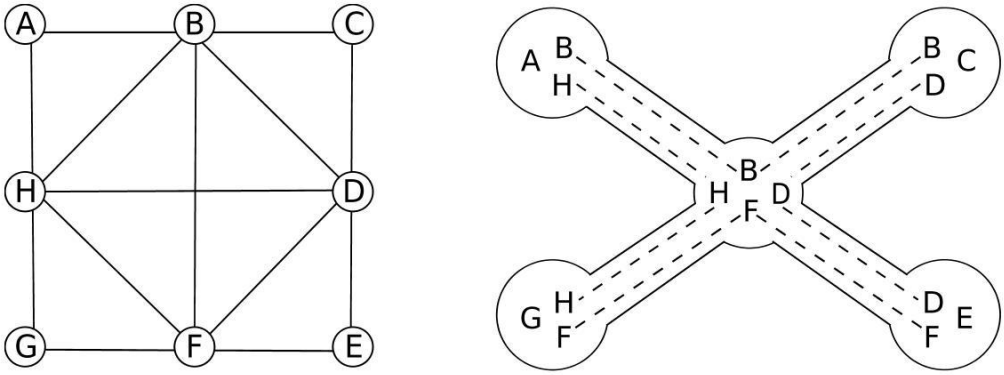


Figure 2.1: A graph and its tree decomposition.

Intuitively, a tree decomposition of a graph is a mapping of the graph into a tree. In fact, it can be easily shown that removing vertices of a graph that are contained in an inner node of a tree decomposition of the graph, separates the graphs into disjoint connected components. This is similar to removing an inner node of a tree. However, in a tree we only need to remove one inner node to separate the tree into disjoint subtrees, whereas an inner node of a tree decomposition of a graph may contain more than one vertex, and we have to remove more than one vertices to separate the graph into connected components.

These observations lead to an intuition that the smaller the treewidth of a graph, the more "tree-like" the graph is. Furthermore, the treewidth of a clique of size n is $n - 1$ (the corresponding tree decomposition has only one bag containing all vertices of the clique), whereas the treewidth of a tree is one (each bag of the corresponding tree decomposition is a two-elements vertex set corresponding to an edge of the tree).

Several dynamic programming algorithms based on tree decomposition have been introduced for NP-hard problems on graphs. For example, in [3], Alber and Niedermeier introduced a tree decomposition-based dynamic programming algorithm for Independent Set Problem and Dominating Set Problem, which are fixed-parameter

with respect to the treewidth of input graphs, in [159] Xu *et al.* also applied tree decomposition-based dynamic programming approach to solve two problems occurring in protein structure prediction, their algorithms are fixed-parameter with respect to the treewidth of the input graphs. In Chapter 6, we introduce two tree decomposition-based dynamic programming algorithms for the Bond Order Assignment Problem, our algorithms are also fixed parameter with respect to the treewidth of the input graphs.

To apply tree decomposition-based dynamic programming algorithms to a graph problem, we have to compute a tree decomposition of the input graph, ideally, an optimal tree decomposition, since this has the smallest width. Unfortunately, computing an optimal tree decomposition is an NP-hard problem [5]. In [27], Bodlaender showed that computing an optimal tree decomposition is fixed-parameter tractable, however the corresponding fixed-parameter algorithm has an impractical running time. Luckily, there are several exact and running-time heuristic algorithms to compute an optimal tree decomposition that are efficient in practice [30, 67]. In Chapter 6, we also report the practical running time of the branch-and-bound algorithm introduced in [67] and implemented by van Dijk *et al.* (`http://www.treewidth.com`), together with the running time of our tree decomposition-based dynamic programming algorithm for the Bond Order Assignment Problem.

To improve legibility and to simplify description and analysis of our algorithms, instead of arbitrary tree decompositions we use *nice tree decomposition*, which is defined as follow:

Definition 2.9. A tree decomposition is a nice tree decomposition if it satisfies the following conditions:

1. Every node of the tree has at most two children.
2. If a node X_i has two children X_j and X_k, then $X_i = X_j = X_k$; in this case X_i is called a *join node*.
3. If a node has one child X_j, then one of the following situations must hold:
   a) $|X_i| = |X_j| + 1$ and $X_j \subset X_i$; in this case X_i is called an *introduce node*.
   b) $|X_i| = |X_j| - 1$ and $X_i \subset X_j$; in this case X_i is called a *forget node*.

The following lemma holds:

Lemma 2.1. *Given a tree decomposition of width ω and m bags of a graph G, a nice tree decomposition of width ω and $O(m)$ bags of G can be computed in linear time.*

A formal proof of this lemma can be found in [97] (proof of Lemma 13.1.3 in [97]).

In the following, we sketch a straightforward algorithm that transforms a tree decomposition of width ω into a nice tree decomposition of the same width in linear time.

Let $\langle \{X_i \mid i \in I\}, T \rangle$ be a tree decomposition of width ω of a graph G. We refer to T as a tree decomposition instead of $\langle \{X_i \mid i \in I\}, T \rangle$ of legibility. To transform T into a nice tree decomposition, we first root T at an arbitrary bag X_r. Rooting T defines a parent-children relation on the nodes of T.

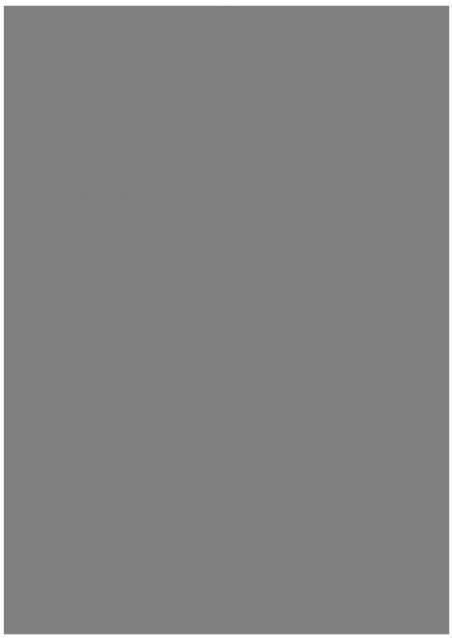


Figure 2.2: A nice tree decomposition.

First, we transform T into a binary tree. Let X_i be an inner node of T with at least two children. We first disconnect every subtree rooted at each child of X_i and construct a binary tree rooted at X_i with as many leaves as the number of children of X_i. Each node of the constructed binary tree is a bag containing the same vertices of G as X_i. Subsequently, we reconnect subtrees rooted at the children of X_i to T at the leaves of the newly constructed binary tree. We apply this procedure to every inner node of T with at least two children in a top-down manner. Of course, it is not necessary to execute this procedure to an inner node with two children, if the bags at that inner node and its children contain the same vertices of G. Afterwards, every inner node of T has at most two children and two children of an inner node contain the same vertices as their parent node.

Thereafter, we replace every edge $\{X_i, X_j\}$, where $|(X_i \setminus X_j) \cup (X_j \setminus X_i)| \geq 2$, with a path consisting of corresponding forget and/or join nodes, such that every inner node of T with one child is either a forget node or an introduce node.

Now, T is a nice tree decomposition and the size of T increases by a factor of at most $d_{max} \cdot \omega$, where d_{max} is the maximum number of children of a node in the original tree decomposition. Furthermore, the width of T remains unchanged and the running time of this algorithm is $O(|I| \cdot \omega)$.

Figure 2.2 illustrates a nice tree decomposition of the tree decomposition shown in Figure 2.1.

2.3.3 Data Reduction and Kernelization

In the two previous sections, we described the depth-bounded search tree and tree decomposition-based dynamic programming as two general techniques that are used in this work to design fixed-parameter algorithms for NP-hard problems. However, given an instance of an NP-hard problem, there are usually some parts of the instance that are not "really hard" and it is a waste of computational resources to directly apply a cost-intensive algorithm to those parts. Thus, it is advisable to solve those trivial parts with a polynomial time preprocessing, to cut down the size of the input instance before applying a cost-intensive algorithm to solve the really hard part of the problem. The polynomial time preprocessing that is used to downsize a problem instance, is called *data reduction*. A data reduction consists of a set of *data reduction rules*, which are defined as follows.

Definition 2.10 (Data Reduction Rule)**.** Let L be a parameterized problem whose input is a pair (I, k). A *data reduction rule* is a mapping that maps an instance (I, k) to a *reduced instance* (I', k') in polynomial time, such that

1. $k' \leq k$,
2. $|I'| \leq |I|$,
3. and the instance (I', k') has a solution if and only if (I, k) also has a solution.

As an example, we consider the following trivial data reduction rules for the parameterized Vertex Cover Problem:

- **Rule 1:** Delete every vertex of degree one, insert its neighbor into the vertex cover and lower k by one.
- **Rule 2:** Insert every vertex of degree at least $k + 1$ into the vertex cover, lower k by one and delete the vertex from the graph.

If there is no vertex of degree one in the graph, we say that the problem instance is *reduced* with respect to Rule 1. If there is no vertex of degree at least $k + 1$, we say that the problem instance is *reduced* with respect to Rule 2. In general, we say that a problem instance is *reduced with respect to a reduction rule* if the reduction rule cannot be applied to the problem instance. Data reduction Rule 1 is *parameter-independent*, since parameter k is not involved in this reduction rule, whereas data reduction Rule 2 is *parameter-dependent*.

The correctness of Rule 1 is obvious, since it is never worth including a vertex of degree one into the vertex cover instead of its neighbor. The correctness of Rule 2 is also obvious, since a vertex cover of size at most k (if it exists) has to contain every vertex of degree at least $k + 1$, otherwise it cannot cover all edges adjacent with a vertex of degree larger than k. Furthermore, the reduced problem instance has a vertex cover of size at most k' if and only if the original problem instance has a vertex cover of size at most k. Moreover, this data reduction rule can be exhaustively executed in polynomial time in the size of the input graph.

For every problem considered in this work, we use data reduction rules to cut down problem instances. In Section 4.4.1, we introduce several data reduction rules for the Flip Consensus Tree Problem. Our data reduction rules for the Weighted Cluster Editing Problem (see Chapter 5) are not described in this work but can be found in [16–18]. For the Bond Order Assignment Problem, which is considered in Chapter 6, we do not explicitly mention any data reduction rule. However, the fact that we lower the valences of atoms by their degrees, can also be considered as a data reduction rule, since the search space of our algorithms and therewith the running times of our algorithms strongly depend on the maximum valence of an atom (see Chapter 6 for more details).

For the Flip Consensus Tree Problem and Weighted Cluster Editing Problem, it was shown that data reduction rules introduced in [99] and [20], are in fact two *kernelizations*, whereby a *kernelization* for a parameterized problem is defined as follows.

Definition 2.11 (Kernelization)**.** A *kernelization* is a data reduction that reduces a parameterized problem instance (I, k) to a parameterized problem instance (I', k') with $k' \leq k$ and $|I'| \leq g(k)$, where $g(k)$ is a function independent of $|I|$. The reduced problem instance (I', k') is called a *problem kernel* of *size* $|I'|$, which is bounded by $g(k)$.

The following lemma states the relation between the fixed-parameter tractability and the kernelizability of a parameterized problem [52].

Lemma 2.2. *A parameterized problem is fixed-parameter tractable if and only if it possesses a problem kernel.*

Although the result of the above lemma does not have a direct practical use, it does suggest to look for kernelizations of parameterized problem, for which there exist fixed-parameter algorithms. In turn, a kernelization may speed up the existing algorithm in practice. Particularly when using kernelization in addition with a depth-bounded search tree algorithm to solve a fixed-parameter tractable problem, Niedermeier and Rossmanith [122] proposed to apply the kernelization at the beginning of every recursive call. This technique is referred to as *interleaving technique*. The essential idea of interleaving technique is: Although applying kernelization in every recursive call consumes time near the root of the search tree, it helps saving much more time near the leaves of the search tree. Niedermeier and Rossmanith [122] proved the following theorem:

Theorem 2. *Let (I, k) denote the input instance of a parameterized problem that can be solved by a depth-bounded search tree algorithm in addition with a kernelization in time $O(P(|I|) + R(q(k)) \cdot \alpha^k)$, where $O(P(|I|))$ is the running time of the kernelization, $q(k)$ denotes problem kernel size, and $O(R(q(k)))$ denotes the running time of the algorithm within each node of the search tree. By applying the kernelization at the beginning of every recursive call, this parameterized problem can be solved in time $O(P(|I|) + \alpha^k)$.*

Due to the above Theorem, we can mention in Chapter 4 and 5 that applying interleaving with the kernelizations in [99] and [20] results in better running time for our algorithms for the Flip Consensus Tree Problem and the Weighted Cluster Editing Problem, respectively.

3 Biological Background

Although the focus of this work is on the complexity and algorithmic aspects of some combinatorial biological problems, a minimum biological knowledge is still required to understand the necessarily of our algorithms. In this chapter, we give a brief introduction to the biological background, that covers all of the biological concepts mentioned in this work. We will stay at a most abstract level and describe biological entities with their function and interaction without going into details at a biochemical level. However, in the next section, we will recapitulate a bit of *school chemistry*, since it may helpful to understand the Bond Order Assignment Problem investigated in Chapter 6. Section 3.2 serves as an introduction to some fundamental biological entities, like DNA, RNA, protein, that will be mentioned throughout this work. In Section 3.3, we give an overview of *computational phylogenetics*, a research field that applies algorithmic approaches to study evolutionary relatedness among organisms. We refer interested users to relevant textbooks, e.g. [4, 31, 108] for more details.

3.1 Basic Chemistry

Atoms are the smallest part of matter that have chemical properties. An atom consists of a dense central *nucleus* surrounded by a cloud of negatively charged *electrons*. A nucleus contains a mix of positively charged *protons* and electrically neutral *neutrons* (except for hydrogen, which is the only stable nucleus with no neutrons). The mass of an atom is called its *atomic mass*.

The number of protons and the number of neutrons in the nucleus of an atom are two important parameters. The number of protons, called *atomic number*, determines the *chemical element* of an atom. The number of neutrons determines the *isotopes* of an element, that are atoms with the same atomic number but differ in atomic mass.

The electrons surrounding the nucleus of an atom are arranged in *electron shells*, which are gravitationally curved paths around the nucleus. An electron shell must be completely filled, before electrons can be added to an outer shell. An atom always tends to fully complete its outermost shell by attracting more electrons or giving up all electrons from its incomplete outermost shell, so that the new outermost shell is full. If the outermost shell is almost full, the atom has a strong tendency to attract electrons. If the outermost shell is almost empty, the atom tends to give up the electrons in the outermost shell. In some cases, atom is also "satisfied" with an almost full outermost shell.

If an atom contains an equal number of protons and electrons, it is electrically neutral, otherwise it has a positive or negative charge. Due to electromagnetic force, atoms with

opposing charges attract each other. An attraction caused by electromagnetic force between atoms is referred to as a *chemical bond*.

In a chemical bond, one of the attending atoms gains electrons, the other one loses electrons or both atoms share electrons to fill their outermost electron shell. In case both atoms share electrons, the chemical bond is referred to as a *covalent bond*. In general, the number of electron pairs involved in a covalent bond, is referred to as its *bond order*. There are three common bond orders: *single bond, double bond* and *triple bond*. In some advance context, bonds with fractional or higher orders than three, such as *quadruple bond, quintuple bond*, and *sextuple bond* are also possible. However, we do not go into details of those cases, since they are out of the scope of this work. An electrically neutral group of two or more atoms joined together by chemical bonds is a *molecule*.

Valence of a chemical element is defined as the sum of bond orders of chemical bonds adjacent an atom of the element in a molecule. Many chemical elements usually have one valence in natural chemical compounds, for example, carbon has valence four, oxygen has valence two, hydrogen has valence one. Some other chemical elements may have more than one possible valences. For example, nitrogen and phosphorous have valence three in some molecules, but in some other molecules their valences are five. However, knowing the bond orders of every bond in a molecule, we can easily induce the valence of each atom in that molecule. Unfortunately, bond order information can be omitted in many database formats. In Chapter 6, we present two fixed-parameter algorithms to compute *optimal* bond order assignments of *molecule graphs*.

3.2 Fundamental Molecular Biology

The smallest functional unit of life that can be considered as a living object is a *cell*. The reasons why a cell is considered as a living object, are its abilities to reproduce by dividing itself into two or more daughter cells; to absorb nutrients and transfer them in energy to use in other activities; and to synthesize *proteins*, which is the most essential building blocks of life. Every living organism is composed of a single or multiple cells (*unicellular* or *multicellular*). Most bacteria are unicellular, whereas other organisms, such as human, animals and plants are multicellular.

In general, one distinguishes two types of cells: *eukaryotic* cells and *prokaryotic* cells. Eukaryotic cells are mostly observed in plants or animals, they have a membrane-enclosed *nucleus* containing their genetic materials, which are encoded as *deoxyribonucleic acid* molecules (DNA) in most organisms or *ribonucleic acid* molecules (RNA) in some bacteria. Prokaryotic cells do not have a nucleus and they are mostly unicellular organisms.

During its life cycle, a cell caries out numerous cellular activities. All of cellular activities are, in fact, chemical reactions, that work through the interaction of mainly three types of biomolecules: DNA, RNA and proteins. Actually, there are other biomolecules, that are also important in maintaining structures of cells, like lipids. However, DNA, RNA and proteins play the most important roles in a cell. In the next section, we briefly describe the functional roles of those biomolecules.

3.2.1 Biomolecules

Proteins are biomolecules consisting of 20 different types of *amino acids* arranged in a linear chain. Almost every protein molecule fold into a unique three-dimensional structure, which highly impact on their biochemical functions. Proteins participate in almost every cellular function, for example, catalyzing biochemical reactions, sending signal to other cells, acting as antibodies in the immune system, transporting other molecules.

A DNA molecule consists of two linear strands winding together in a double-helical structure. Each strand is a long chain built from four types of *nucleotide bases*: *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T). Two strands of a DNA are held together by hydrogen bonds (a type of chemical bond) between opposing nucleotide bases, that are A-T and G-C. With their double-helical structure of two complementary strands, DNA molecules are mostly chemically inactive and serve as libraries of hereditary information and instructions for cellular activities. The smallest unit of a DNA molecule that encodes information is a *gene*. In prokaryotic organisms, genes are continuous strings. In eukaryotic organisms, genes are broken into pieces, called *exons* and *introns*. While exons encode information, that are used by *gene expression* process to synthesize proteins, the functionality of introns are not yet explored. The complete DNA molecule of an organism is called its *genome*. Since two strands of a DNA are fully complement to each other, a DNA molecule can be described with the nucleotide sequence of one of its two strands.

The chemical structure of RNA is similar to a single DNA strand, the main difference between RNA and a single DNA strand is that RNA molecules contain *uracil* nucleotide (U) instead of thymine (T). Because of their single-strand structure (versus double-strand structure of DNA), RNA are more chemically active than DNA. They participate in many important cellular activities, particularly in *gene expression*, a process that uses information from protein-encoding genes to synthesize proteins needed by cells or organism to react to internal or environmental conditions.

In the next section, we describe *gene expression*, that is one of the most important process where DNA, RNA, and proteins play the central role. We also point out a combinatorial problem arising in gene expression analysis, which can be formalized with the Weighted Cluster Editing Problem investigated in Chapter 5.

3.2.2 Gene Expression

Gene expression is the process, in which proteins or functional genetic products are synthesized. In an over-simplified view, gene expression consists of the following steps:

1. *Transcription.* Transcription is the process that copies a DNA sequence into an equivalent RNA sequence. At the beginning of a transcription, an enzyme, called *RNA polymerase*, binds to the DNA sequence at a position that is a few bases before a gene and transcripts (copies) the information of the gene into a complementary RNA sequence, called *messenger RNA* (mRNA).

2. *RNA splicing.* RNA splicing is a process that only occurs in eukaryotic organisms. It removes non-coding sections in the transcripted mRNA, that was translated from the introns of the respective gene, and joins coding sections together.

3. *Translation.* Translation is a process that uses information encoded in the mRNA to synthesize protein. In this process, a particular type of RNA, called *transfer RNA* (tRNA), binds to a nucleotide triplet on the mRNA, that encodes a type of amino acid, and transfer the corresponding amino acid to the so far created protein sequence. A nucleotide triplet of an mRNA, that encodes a type of amino acids, is called a *codon*.

4. *Folding and post-translation modification.* The newly created protein folds into the desired three-dimensional structure to function. Usually, the newly created protein also undergoes several chemical modifications that may attach further biochemical functional groups to the protein or modify some amino acids on the protein.

5. *Targeting.* Targeting is a process that transports the newly created protein to the cellular location, where it is needed.

Gene expression is the most important process of a cell to react to internal and environmental condition. Thus, depending on which protein are needed, the *expression* of one or more corresponding genes may up- or down-regulated. The *expression level* of a gene are usually quantified by the amount of the corresponding mRNA transcripted in the transcription. The amount of mRNA can be measured with aid of a *microarray*, which is an array with thousands of *DNA spots* assembled on a rectangle slide, and each DNA spot contains a huge number of short identical nucleotide sequences [111,115,132]. To quantify gene expression levels of genes in a cell, the cell extract is washed over a microarray, mRNA fragments in the cell extract are sticked to spots with complementary nucleotide sequences on the microarray. Afterwards, the types and amount of mRNAs is detected to infer the gene expression levels of corresponding genes. See [111] for more details about applications of microarray.

In a gene expression analysis, expression levels of genes are measured under different conditions (including time of measurements, physical and chemical agents). The measured expression levels of a gene are stored in a vector, called the *characteristic vector* of the gene or the *expression pattern* of the gene. Information obtained by monitoring gene expression level of particular genes are useful in many biological and medical applications. For example, those information may help to detect viral infection, susceptibility to cancer of an individual, resistant of a bacterium to antibiotics, or reactions of cell to medical treatments.

One of the most important tasks in gene expression analysis is to classify genes into clusters of genes with similar expression patterns. In turn, genes with similar expression patterns indicate genes that are tightly co-expressed. Moreover, gene expression clusters also help to infer functional roles of unknown genes.

The problem is now, how to classify genes according to their expression patterns. In Chapter 5, we describe the Cluster Editing approach, that can be used to classified

biological entities with respect to any given similarity measures. In particular, we investigate a combinatorial problem arising with this approach, called Weighted Cluster Editing Problem, and present one of our fixed-parameter algorithms for that problem.

3.3 Computational Phylogenetics

Already in the early ancient time, philosophers from many cultures assumed that species may change over time and stem from a single common ancestor. In the 18th. and 19th. centuries, several fundamental scientific works have been published in this research area. In 1809, Jean-Baptiste Lamarck came up with the idea that altering of one species into another results from the inheritance of adjustments, which are acquired by the parents to adapt to living environment. In 1859, Charles Darwin published *On the Origin of Species*, one of the most important book in this field. In this book, Darwin explained the *natural selection* as a process, by which traits become more or less common in a population, depending on the survival and reproduction of their bearers. Moreover, Darwin also presented various evidences of *evolution*, a process by which the inherited traits of a population change through successive generations. In 1900, Gregor Mendel discovered that traits were inherited in a predictable manner. However, this appeared to be a contradiction to the natural selection concept of Darwin. In 1930, Ronald Fisher resolved this contradiction in his publication *Genetical Theory of Natural Selection* and set the foundations for the establishment of *population genetics*. In the 1940s, Oswald Avery and colleagues identified DNA as the genetic material. In 1953, James Watson and Francis Crick published the structure of DNA, which provides the physical basis for inheritance. Since then, population genetics and molecular biology are parts of evolutionary biology. Subsequently, *phylogenetics* was developed as a research field in evolutionary biology, that investigates the evolutionary relationship of species, also called *taxa* (singular *taxon*).

In phylogenetics, evolutionary relations of taxa are usually represented as *phylogenetic trees*, also called *phylogenies*, that are trees (an undirected graph without cycle, in term of graph-theory) where each leaf corresponds to a taxon.

One distinguishes between *rooted* phylogeny and *unrooted* phylogeny. In a *rooted* phylogeny, an inner node represents a hypothetical last common ancestor of taxa located at leaves of the subtree rooted at the inner node. Furthermore, in a rooted phylogeny, an inner node also represent the cluster of taxa located at the leaves of its subtree. In an *unrooted* phylogeny, inner nodes can be considered as speciation events occurring in the past, since unrooted phylogenetic trees only represent the relationship of the taxa without any assumption about their ancestry.

In general, phylogenetic trees are binary trees, because speciation events usually occur when an ancestral lineage splitted in two new independent lineages. However, there are some phenomena in nature that might be best modeled by a *mutifurcating* tree, where inner nodes may have more than two children [109, 124]. Furthermore, evolutionary relations are also represented as directed acyclic graphs, called *phylogenetic networks*, if

input data support more than one phylogenies equally well or horizontal gene transfer occurred [33, 78, 90–92]. However, we only focus on rooted binary phylogenetic trees in this work.

Phylogenetic Reconstruction

Apart from their historical evolutionary meaning, phylogenies also find applications in biological and medical research, where historical or hierarchical structures of the evolution of taxa are of interest [145]. Therefore, reconstructing phylogenies is an important task in biology. Traditionally, phylogenies are reconstructed from morphological data, that are outward appearances of species such as body size, length or size of particular bones, pattern of their movements, and many other physical features of species. Nowadays, biomolecular data such as nucleotide sequences, amino acid sequences and other data types like gene frequencies, quantitative traits, restriction sites, microsatellites, DNA hybridization are used, in addition to morphological data, to reconstruct phylogenies. In general, it is not certain whether morphological data or biomolecular data are preferable for inferring phylogenies. For extinct species, it is difficult or even impossible to obtain biomolecular data, thus using morphological data of mummies or fossil record is the only way to estimate their relationships, whereas phylogeny of viruses can only be inferred from their biomolecular data. Moreover, phylogenies are also usually reconstructed without information about the extinct species, since many organisms, such as viruses, does not leaves fossil records. In this case, the only way to infer phylogenies is to use genetic information of existing species.

Given a set of taxa, the task of phylogenetic reconstruction is to estimate a phylogeny, that contains the taxa at its leaves, and represents a hypothesis about the evolutionary ancestry of the taxa. Since it is in general unknown how the evolution of the taxa happened, phylogenetic reconstruction methods usually infer phylogenies optimizing a certain criteria, that is believed to lead to a good phylogeny. Even so, there is no guarantee that the resulting tree represents the true evolutionary process of the input taxa. In fact, there is no uniquely correct methods for inferring phylogeny. A biologist has to choose a method that best fits his demand. Given a set of taxa with a criteria to optimize, the naïve algorithm chooses a tree-topology among all possible tree-topologies that optimizes the given criteria. Unfortunately, the number of tree-topologies, and thus the running time of the algorithm, grows super-exponentially in the number of taxa. Many phylogenetic reconstruction methods with different optimization criteria have been introduced to deal with the super-exponentially large search space. See [57] for an overview. Most of these methods can be classified in two broad categories: *distance-based methods* and *character-based methods*.

Distance-based method. Distance-based methods reconstruct phylogenies with respect to the evolutionary distances involved in the input data such as edit-distance of biological sequences, melting temperature of DNA hybridization, the strength of antibody cross reaction, etc. Phylogenies reconstructed by distance-based methods are

weighted trees, where pairwise distances between species reflect the evolutionary distance involved in the input data. Representations of distance-based methods include neighbor-joining [72, 136], agglomerative clustering [113] or least-squares [58]. See relevant textbooks [57, 137] for more details about distance-based methods.

Character-based method. Character-based methods reconstruct phylogenies directly from the *characters* of taxa, that are features, which can have different states. From the algorithmical point of view, one distinguishes two kinds of character according to the number of their states: *Binary character* and *numerical character*. Numerical characters have more than two states, such as nucleotide at each position in DNA or RNA sequences, amino acid at each position of protein sequences. Binary characters have exactly two states 0 or 1, and thus can be used to represent features with exactly two different states, such as "taxon is mammal", "taxon is vertebrate", "taxon has swings".

In character-based method, each taxon is characterized by a character vector, where the j^{th} component of the vector represents the status of the j^{th} character. The goal of character-based methods is to construct a rooted tree with the following attributes:

- Each leaf corresponds to a taxon given in the input.
- Each inner node is marked with the character vector of the hypothetical common ancestor of taxa located at the subtree rooted at the inner node.
- Each edge corresponds to a character mutation and every taxon in the subtree below an edge undergoes the corresponding character mutation.

Most of the character-based methods follow the Maximum Parsimony or the Maximum Likelihood criteria.

Maximum Likelihood-based methods take biomolecular sequences, such as DNA sequences or protein sequences, as input and computes a phylogeny with the highest likelihood for the input data, under a given evolutionary model. Recently, Maximum Likelihood was proven to be NP-hard [42, 43, 135]. We refer interested readers to relevant textbooks e.g. [57, 137] for more details.

Maximum Parsimony criteria requires the constructed phylogenies to contain the minimum number of mutations for a given taxa set [57]. An advantage of Maximum Parsimony criteria is that it agrees with the common assumption that unnecessary mutations are not likely to occur often in nature. Unfortunately, reconstructing maximum parsimonious phylogenies is also an NP-hard problem, even for input data with binary characters [60]. See [57, 137] for more details about Maximum Parsimony.

To cope with the NP-hardness of computing maximum parsimonious phylogeny, Blelloch *et al.* [144] introduced a fixed-parameter algorithm that computes a most parsimonious tree of length at most $m + q$ in time $O(21^q + 8^q \cdot m^2 n)$ if there is one, where the length of a phylogeny is defined as the number of its edges, n is the number of taxa, m is the number of binary characters and q is an input parameter.

As an alternative to maximum parsimony, Chen *et al.* [38] introduced an approach that takes a set of n taxa characterized by m binary characters and constructs a phylogeny

with at most m edges. To guarantee the lengths of constructed phylogenies, Chen *et al.* proposed to "correct" the input data with a minimum number of edit operations. Unfortunately, it is NP-hard to find a minimum number of edit operations [38]. In the following Chapter, we introduce two fixed-parameter algorithms for this problem.

4 Flip Consensus Tree

In this Chapter, we investigate the Flip Consensus Tree Problem, a special case of the Flip Supertree Problem that is encountered when using the approach introduced by Chen *et al.* [38] to reconstruct phylogeny. After giving a biological motivation in Section 4.1, we formalize the Flip Consensus Tree Problem problem as a graph modification problem in Section 4.3. In Section 4.4.1, we describe a set of data reduction rules to downsize problem instances. In Section 4.4.2 and 4.4.3, we introduce two fixed-parameter algorithms for the Flip Consensus Tree Problem based on the depth-bounded search tree technique. Section 4.5 discuses several heuristic improvements to speed up our algorithms in practice. We implemented one of our algorithms in Java and report our computational results in Section 4.6. Section 4.1 recapitulates our results introduced in this chapter and points out several future research directions regarding the Flip Supertree Problem.

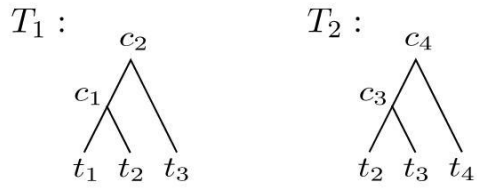
4.1 Motivation

Besides inferring phylogenies directly from a taxa set as described in Section 3.3, phylogeny can also be inferred from smaller phylogenies over overlapping taxa sets. The resulting phylogeny, called the *supertree* of the input phylogenies, contains all or most taxa of the input trees. In ideal case, the input trees allow for a single supertree that simultaneously *displays* all input trees, that means, each input tree can be obtained from the supertree by removing all taxa that the input tree does not contain. In practice, input trees almost never allow for a single supertree due to the incompatibilities among the input trees. The main goal of all *supertree methods* is to construct supertrees that preserve the maximum of phylogenetic information in the input trees.

Most of supertree methods can be classified into two broad categories: The *MRP* (Matrix Representation with Parsimony) approaches [10, 11, 129] and Build-like approaches [1, 123, 139].

Given a set of input trees, an MRP approach encodes the input trees in a matrix M as following: The matrix M contains a row for each input taxon and a column for each inner node of every input tree. A matrix item M_{ij} is set to '?' if the input tree with the inner node corresponding to column j does not contain the taxon corresponding to row i; or 1 if the input tree with the inner vertex corresponding to column j contains the taxon corresponding to row i at a leaf of the subtree rooted at the inner vertex corresponding to column j; otherwise '0'. The matrix M is called the *matrix representation* of the input trees. If M contains '?', it is said to be *incomplete*. See Figure 4.1 for an illustration.

After constructing the matrix representation of the input trees, MRP supertree method compute a maximum parsimonious phylogeny for the resulting matrix.



M_1	c_1	c_2
t_1	1	1
t_2	1	1
t_3	0	1

M_2	c_3	c_4
t_2	1	1
t_3	1	1
t_4	0	1

M	c_1	c_2	c_3	c_4
t_1	1	1	?	?
t_2	1	1	1	1
t_3	0	1	1	1
t_4	?	?	0	1

Figure 4.1: Matrix M_1 and M_2 represent phylogenies T_1 and T_2. Matrix M represents both phylogenies T_1 and T_2.

MRP approaches are known to compute better solution on simulated data than BUILD-like approaches [56]. Unfortunately, MRP approaches suffer from the same computational hardness of computing maximum parsimonious phylogeny [60]. However, if the matrix representation of M shows a particular property, called *perfect phylogeny*, a maximum parsimonious phylogeny of M can be computed in polynomial time [77, 127]. On a complete binary matrix representation M, the perfect phylogeny concept is stated as following:

Definition 4.1 (Perfect Phylogeny)**.** Let M be an $n \times m$ binary matrix representation of n taxa. A *perfect phylogeny* of M, if it exists, is a rooted tree T with the following properties:

- Each leaf is labeled by a row of M, and corresponds with the taxon characterized by that row.
- Each column of M labels at most one edge.
- The characters associated with the edges along the unique path from the root to a leaf exactly specify the character vector of the corresponding taxon, i.e., the character vector of the taxon has '1' entries in all columns corresponding to characters associated to the edges of the path and a '0' entry otherwise.

See Fig. 4.2 for an illustration. The definition of perfect phylogeny implies that the root of the perfect phylogeny is labeled by the null vector and each character changes from state '0' to state '1' at most once and never changes back from state '1' to state '0'. Thus, a perfect phylogeny is a phylogeny with maximum parsimony.

Gusfield [77] proved the following theorem.

Theorem 3. *Let M be a complete binary matrix representation of n taxa characterized by m characters, and O_i denote the set of taxa with a '1' in column i. M admits a perfect phylogeny if and only if every two characters i and j are* compatible*, that means, it holds*

	c_1	c_2	c_3	c_4	c_5
t_1	1	0	0	0	0
t_2	1	1	0	0	0
t_3	1	1	1	0	0
t_4	0	0	0	1	0
t_5	0	0	0	1	1

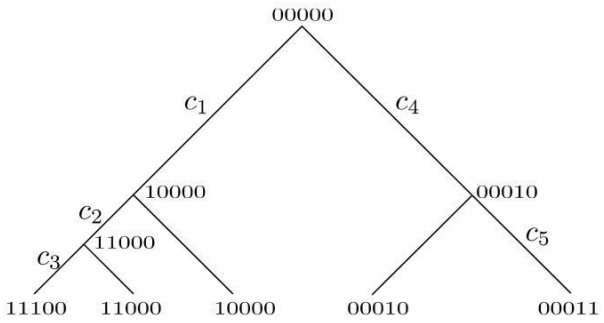


Figure 4.2: Perfect Phylogeny

- $O_i \subseteq O_j$, *or*
- $O_j \subseteq O_i$, *or*
- $O_i \cap O_j = \emptyset$.

In [77], Gusfield also introduced an algorithm with running time $O(mn)$ to test if a binary matrix M admits perfect phylogeny, and construct a perfect phylogeny of M, if it exists.

For *incomplete* binary matrix representation, that contains at least one '?' as an item instead of '1' or '0', Pe'er *et al.* [127] introduced a near-linear running time (in the size of the matrix) algorithm to replace each '?' with '0' or '1' in a way admitting a perfect phylogeny, if it is possible.

The main problem is: The matrix representation of input trees almost never admits perfect phylogeny in practice. In [38], Chen *et al.* proposed to edit the matrix representation by flipping the minimum number of items from '1' to '0' or from '0' to '1', to make the resulting matrix admitting perfect phylogeny. This approach is called the *flip supertree* approach. In [56], Eulenstein *et al.* showed that on simulated data, the flip supertree approach computes better supertrees than BUILD-like methods and, at least, as good as MRP approaches, regarding various distance and similarity measures . However, the problem is now, how to find the minimum set of flippings that make a matrix admitting perfect phylogeny. This problem is known as FLIP SUPERTREE PROBLEM that can be formalized as following:

Definition 4.2 (FLIP SUPERTREE PROBLEM)**.** Given an $n \times m$ matrix M, where $M_{ij} \in \{0, 1, ?\}$, the FLIP SUPERTREE PROBLEM asks for the minimum number of entries of M to be flipped from '0' to '1' or from '1' to '0' to make M admitting perfect phylogeny, i.e., there exist a replacement of every '?' with '1' or '0', that make M admitting perfect phylogeny.

In this work, we investigate a special case of the flip supertree approach, the *flip consensus tree* approach, where the input trees contain the same set of taxa. This implies that the matrix representation of the input trees does not contain '?'. In this

case, the underlying problem, called Flip Consensus Tree Problem, is formally defined as following:

Definition 4.3 (Flip Consensus Tree Problem)**.** Given an $n \times m$ binary matrix M, the Flip Consensus Tree Problem asks for the minimum number of entries of M to be flipped from '0' to '1' or from '1' to '0' that makes M admitting perfect phylogeny.

4.2 Previous Work

In the following, we recapitulate previous results for the Flip Supertree Problem and the Flip Consensus Tree Problem.

In [37, 38], Chen *et al.* showed that the Flip Consensus Tree Problem is NP-hard. Therefore, the Flip Supertree Problem is also NP-hard. A branch-and-bound algorithm was introduced for the Flip Consensus Tree Problem in [35], but this algorithm can only solve problem instances with very limited number of taxa.

Based on a graph-theoretical interpretation of the Flip Consensus Tree Problem, Chen *et al.* [38] introduced a simple fixed-parameter algorithm with running time $O(6^k\, mn)$, where k is the minimum number of flips. Furthermore, they introduced an approximation algorithm with approximation ratio $2d$ where d is the maximum number of '1's in a column for this problem [38]. In [99], Komusiewicz *et al.* proved a problem kernel of size $O(k^3)$ vertices for this problem.

All of the abovementioned results refer to the Flip Consensus Tree Problem. For the Flip Supertree Problem several heuristic algorithms exist [36, 56, 68]. Recently, our group proved that the Flip Supertree Problem is W[2]-hard [21] with respect to the number of flips. This result prohibits the hope for a fixed-parameter tractable algorithm with respect to the minimum number of flips for the Flip Supertree Problem. However, Chimani *et al.* [41] recently introduced an integer linear programming (ILP) formulation and a set of data reductions for the Flip Supertree Problem as well as the Flip Consensus Tree Problem that can solve Flip Supertree Problem instances with up to 100 taxa in reasonable time.

4.3 Graph-theoretical Model

In this work, we use a graph-theoretical model of the binary matrix representation to analyze the Flip Consensus Tree Problem. This model was first used by Chen *et al.* [37] and can be defined as following:

Definition 4.4 (Character graph)**.** The *character graph* $G = (V_t \cup V_c, E)$ of an $n \times m$ binary matrix M is an undirected and unweighted bipartite graph with $n + m$ vertices $t_1, \ldots, t_n, c_1, \ldots, c_m$ where $\{c_i, t_j\} \in E$ if and only if $M[i, j] = 1$. The vertices in V_c represent characters and those in V_t represent taxa. We call the vertices c- or t-vertices, respectively.

See Figure 4.3 for an illustration of a character graph.

	c_1	c_2	c_3	c_4
t_1	1	0	0	0
t_2	1	1	0	1
t_3	1	1	1	1
t_4	0	0	1	1

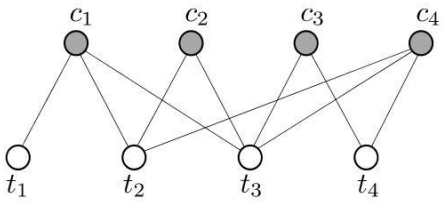


Figure 4.3: A binary matrix and its character graph

From the set-theoretical point of view, the neighborhood of a character vertex c_i corresponds to the taxa set, that have state '1' for the character associated with c_i. Therefore, the neighborhood of a character vertex c_i corresponds to the set O_i defined in Theorem 3. Hence, two characters are compatible if the neighborhoods of their corresponding character vertices contain each other or do not intersect.

Therefore, two incompatible characters corresponds to two c-vertices with overlapping neighborhoods. An M-graph, which is defined as following, is the minimal graph structure describing two character vertices with overlapping neighborhoods.

Definition 4.5 (M-graph)**.** An *M-graph* is an induced subgraph of a character graph that is a path of length four starting and ending at t-vertices.

Fig. 4.4 illustrates an M-graph and its corresponding submatrix.

	c_1	c_2
t_1	1	0
t_2	1	1
t_3	0	1

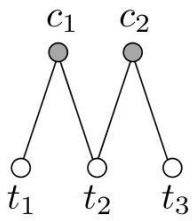


Figure 4.4: M-graph

Using this graph theoretical model, Chen *et al.* [38] restated Theorem 3 as following:

Theorem 4. *An $n \times m$ binary matrix M admits perfect phylogeny if and only if it does not contain an* M-graph *as induced subgraph.*

We call a character graph *M-free* if it does not contain an M-graph as an induced subgraph.

From the definition of character graph, it is obvious that flipping a matrix item from '1' to '0' corresponds to the deletion of the corresponding edge in the character graph and flipping a matrix item from '0' to '1' corresponds to the insertion of the corresponding absence edge in the character graph. In particular, the number of flips needed to make a binary matrix admitting perfect phylogeny equals the number of edge modifications needed to make the corresponding character graph M-free. Therefore, the Flip Consensus Tree Problem is equivalent to the following graph modification problem:

Definition 4.6 (Graph-theoretical Model of Flip Consensus Tree Problem)**.** Given an undirected bipartite graph $G = (V_t \cup V_c, E)$, the graph-theoretical model of Flip Consensus Tree Problem asks for the minimum number of edge modifications, that is, edge deletions and edge insertions that transform the character graph into an M-free bipartite graph.

In this work, we investigate the Flip Consensus Tree Problem on the basis of its graph-theoretical model as described in the above definition. The following notations are frequently used throughout this chapter:

For two c-vertices $c_i, c_j \in V_c$, we denote

- $X(c_i, c_j) := N(c_i) \setminus N(c_j)$
- $Y(c_i, c_j) := N(c_i) \cap N(c_j)$
- $Z(c_i, c_j) := N(c_j) \setminus N(c_i)$

recalling that $N(v)$ denotes the neighborhood of a vertex v. See Fig. 4.6 for a illustration. Actually, $Z(c_i, c_j)$ can be written as $X(c_j, c_i)$, but we use both notations for the sake of clarity.

We call c_i and c_j *c-neighbors* if and only if $Y(c_i, c_j)$ is not empty. An edge modification is said to be *associated* with a vertex v (v can be a c-vertex or a t-vertex), if it inserts or deletes an edge incident to v. Furthermore, we denote $m := |V_c|$ and $n := |V_t|$.

4.4 Algorithms

In this Section, we investigate the *parameterized* version of the Flip Consensus Tree Problem, which is defined as following:

Definition 4.7. The *parameterized* Flip Consensus Tree Problem takes a character graph $G = (V_t \cup V_c, E)$ and an integer k as input, and asks if it is possible to transform G into an M-free character graph with at most k edge modifications.

Since M-free is a hereditary property with a finite set of forbidden subgraph, the parameterized Flip Consensus Tree Problem is fixed-parameter tractable with respect to parameter k (see Theorem 1). In fact, Chen *et al.* [38] introduced a straightforward depth-bounded search tree algorithm for the parameterized Flip Consensus Tree Problem. This algorithm simply identifies an M-graph in G and branches into six cases (four cases of deleting one existing edge and two cases of inserting one absence edge) to eliminate the M-graph. Thus, the size of the corresponding search tree is bounded by $O(6^k)$. Chen *et al.* adapted the Gusfield's algorithm in [77] to identify M-graph in time $O(mn)$. All in all the running time of this algorithm is bounded by $O(6^k\, mn)$. Figure 4.5 illustrates the branching strategy of Chen *et al.* [38].

To compute the minimum number of edge modifications, we follow the depth-bounded search tree paradigm and first solve the parameterized Flip Consensus Tree Problem for $k = 1$. If there is no solution with at most k edge modifications, we solve the

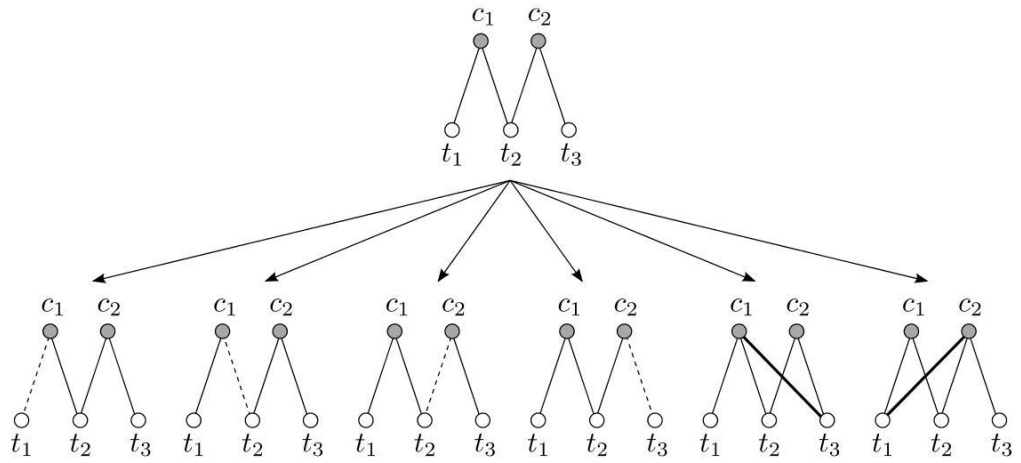


Figure 4.5: The initial branching strategy. Dashed edges denote deleted edges. Bold edges denote inserted edges.

parameterized Flip Consensus Tree Problem with increasing k repeatedly, until finding the first k for that the character graph can be transformed into a M-free character graph by k edge modifications. If k_{opt} is the minimum number of edge modifications to transform G into an M-free character graph, we have to call the algorithm for the parameterized Flip Consensus Tree Problem k_{opt} times to find the optimal solution for the Flip Consensus Tree Problem. Thus, the Flip Consensus Tree Problem can be solved in $O(k_{opt}(6^{k_{opt}}\, mn)) = O(6^{k_{opt}}\, mn)$ using the straightforward algorithm of Chen *et al.* described above. All in all, the straightforward algorithm of Chen *et al.* is a fixed-parameter algorithm for the Flip Consensus Tree Problem with respect to the minimum number of edge modifications.

In the following, we describe two improved fixed-parameter algorithms for the parameterized version of the Flip Consensus Tree Problem. The main idea of our algorithms bases on the following observation:

- As long as there is a c-vertex of degree at least three, a special structure of intersecting M-graphs occurs in G, that allows for a branching strategy with branching number 4.83 as shown in Section 4.4.2.1. If every c-vertex of the character graph G has degree at most two, the Flip Consensus Tree Problem problem can be solved in polynomial time as shown in Section 4.4.2.2.

- Besides the case, where every c-vertex has degree at most two, there are further special cases, where the Flip Consensus Tree Problem can be solved in polynomial time. Thus, we improve our branching strategy by pruning certain subtrees of the search tree, for which the corresponding subproblem can be solved in polynomial time. This improves the branching number of our branching strategy to 4.42.

Based on those branching strategies, we present two fixed parameter algorithms with running times $O(4.83^k\,(m+n)+mn+n^2\log n)$ and $O(4.42^k\,(m+n)+mn+n^2\log n)$ for the parameterized Flip Consensus Tree Problem, and thus, also for the Flip Consensus Tree Problem if the minimum number of edge modification is denoted by k. The theoretical running times of our algorithms can be further improved to $O(4.83^k + poly(m,n))$ and $O(4.42^k + poly(m,n))$ by using the interleaving technique (see Section 2.3.3) with the kernelization of Komusiewicz *et al.* [99], where $poly(m,n)$ denotes a polynomial function in n and m. However, this polynomial function is not specified in [99] and can be rather large.

In the following, we assume that G is a connected graph. If G is not connected or decomposes during the execution of the algorithms, we compute the solutions for each connected components separately, because it is never worth to connect different connected components.

4.4.1 Data Reductions

Given a character graph G as input, we use the data reduction rules introduced in this section to cut down the size of G. In our search tree algorithms, these data reduction rules are also executed in the beginning of each recursive call. Reduction Rules 1 and 2 are fairly simple:

Rule 1: Delete all c-vertices $v \in V_c$ of degree $|V_t|$ from the graph.

Lemma 4.1. *Rule 1 is correct.*

Proof. Let c be a c-vertex of degree $|V_t|$ in the input graph G. Then, c is connected to all t-vertices in G and there cannot be an M-graph containing c. Furthermore, it is not possible to insert any new edge incident to c. Assume there is an optimal solution for G that deletes edges incident to c in G. If we execute all edge modifications of the optimal solution except deletions of edges incident to c, we also obtain an M-free graph, since every M-graph that does not contain c is destroyed by edge modifications of the optimal solution and there is no M-graph containing c. This is a contradiction to the assumption that the solution is optimal, so edges incident to c-vertices of degree $|V_t|$ are never deleted in an optimal solution. Thus, the corresponding c-vertices need not be observed and can be removed safely. □

Rule 2: Delete all c-vertices $v \in V_c$ of degree one from the graph.

The correctness of Rule 2 is obvious.

After computing and saving the degree of every vertex in G in time $O(mn)$, Rules 1 and 2 of the data reduction can be exhaustively applied in time $O(m+n)$. Furthermore, after applying Rule 1 is exhaustively, the c-vertex with maximum degree is contained in at least one M-graph. In the proof of Theorem 5, we show that identifying this M-graph takes time $O(m+n)$.

Rule 3 of our data reduction bases on the following lemma.

Lemma 4.2. *Let $V_c' \subseteq V_c$ be a set of c-vertices with the same neighborhood. If there is an optimal solution that executes edge modifications associated with a c-vertex in V_c', there is an optimal solution that executes the same edge modifications on all c-vertices in V_c'.*

Proof. Let c_1, c_2 be two arbitrary c-vertices in V_c'. Since c_1 and c_2 share the same neighborhood, there is no M-graph containing only c_1 and c_2. Assume that the length-four path (t_1, c_1, t_2, c, t) with $t_1, t_2 \in N(c_1)$, $c \in V_c \setminus V_c'$ and $t \in V_t \setminus N(c_1)$ forms an M-graph. The length-four path (t_1, c_2, t_2, c, t) is also an M-graph. To eliminate these M-graphs, we have to execute suitable edge modifications.

Case 1: If the optimal solution comprises edge modifications that are associated with c, then no edge modifications associated with c_1 or c_2 are necessary to eliminate M-graphs (t_1, c_1, t_2, c, t) and (t_1, c_2, t_2, c, t).

Case 2: Assume that the optimal solution comprises different edge modifications associated with c_1 and c_2 to eliminate the M-graphs (t_1, c_1, t_2, c, t) and (t_1, c_2, t_2, c, t) . Without loss of generality, the optimal solution inserts an edge connecting c_1 with t and deletes at least an edge incident to c_2. The edge deletions associated with c_2 can be replaced by inserting the edge $\{c_2, t\}$ without creating a new M-graph containing c_1 and c_2, because c_1 and c_2 have the same neighborhood. This modification does not increase the cost of the optimal solution in this step.

Thus, there is an optimal solution that executes analogous edge modifications for all vertices in V_c. □

As a consequence of the above lemma, whenever we execute an edge modification associated with a c-vertex in a set of c-vertices with the same neighborhood, we execute the analogous edge modification on all other c-vertices in this set. To make use of this observation, we introduce the following *merge operation*.

Merge operation. Let $V_c' \subsetneq V_c$ be a set of c-vertices that have the same neighborhood. The merge operation joins every c-vertex in V_c' into a new c-vertex c and connects c with all t-vertices in the neighborhood of V_c' and sets the *cost* of deleting or inserting an edge incident to c to $|V_c'|$.

The merge operation ensures that every edge modification associated with a c-vertex in a set of c-vertices with the same neighborhood will be executed to all c-vertices in this set, and the corresponding edit costs are correctly estimated.

Rule 3: Merge c-vertices with the same neighborhood.

The correctness of Rule 3 can be derived directly from Lemma 4.2.

Now, we describe how to identify all sets of c-vertices with the same neighborhoods efficiently. Since each set of c-vertices with the same neighborhood is a *critical independent set*, that is a set of vertices with the same neighborhood and there is no edge incident to two vertices of the set, the algorithm of Hsu *et al.* [86] can be applied to compute all sets of c-vertices with the same neighborhoods in time $O(n + m)$. Thus, Rule 3 can be exhaustively executed in time $O(n + m)$.

Note that applying merge operation causes the character graph to contain edges with different edit cost, but the edge modifications associated with a c-vertex always have the same cost. Furthermore, neither algorithm introduced in this work can be used to solve arbitrary *weighted* Flip Consensus Tree Problem instances, where edit costs are different even if they are associated with the same c-vertex. However, these algorithms can be used to solve instances, where weights are assigned to characters, i.e., edge modifications associated with one c-vertex must have identical weight.

Rule 4: For every c-vertex c of degree two, set $\{c, t\}$ as *forbidden* for all $t \in V_t \setminus N(c)$, i.e., in later stages of our algorithms, we will not insert any edge incident to c.

Lemma 4.3. *Rule 4 is correct.*

Proof. Let G be a character graph and c be a c-vertex of degree two in G. Let G' be the graph resulting from an optimal set of edge modifications on G. Assume that G' owns an edge (c, t) not contained in G.

We will now show that there is another optimal solution for G without inserting (c, t). Let t_i, t_j be the t-vertices connected with c in G. Since all M-graphs eliminated by inserting $\{c, t\}$ into G contain edges $\{c, t_i\}$ and $\{c, t_j\}$, we can delete $\{c, t_i\}$ or $\{c, t_j\}$ from G to eliminate these M-graphs instead of adding $\{c, t\}$ to G. Deleting an edge can only cause new M-graphs containing the c-vertex incident to this edge. But after the removal of one of the edges $\{c, t_i\}$ or $\{c, t_j\}$, c has degree one and cannot be contained in any M-graph. Therefore, the resulting graph is still M-free and the number of edge modifications does not increase since an insertion is replaced with only one deletion associated with the same c-vertex. Hence, we can prohibit inserting edges incident to degree-two c-vertices without changing the cost of the optimal solution. □

After computing and saving the degree of every vertex in G, Rule 4 can be easily done in time $O(m)$.

Corollary 4.4. *If every c-vertex in a character graph has degree two, there is an optimal solution for the* Flip Consensus Tree Problem *without inserting any edge into the character graph.*

With the following lemma, we summarize the running time of our data reduction.

Lemma 4.5. *All data reduction rules introduced above can be exhaustively applied in time $O(n+m)$, after computing and storing the degree of every vertex in G. Computing the degrees of all vertices of G can be done in time $O(mn)$.*

The correctness of Lemma 4.5 is obvious.

In [99], Komusiewicz *et al.* generalized our data reduction Rule 1 and 2 by allowing to remove c-vertices that are not occurring in an M-graph. This generalized reduction Rule can be exhaustively applied in time $O(m^2 n)$. Furthermore, they used a more sophisticated reduction rule to show that Flip Consensus Tree Problem admits an $O(k^3)$-vertex problem kernel. However, the practical use of that reduction rule is in question, since it seems to be highly time-consuming and rarely applicable in practice.

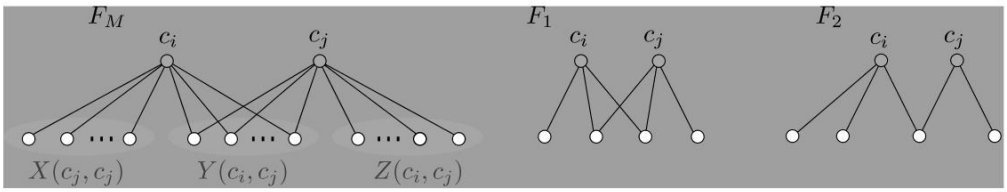


Figure 4.6: The big M-graph F_{M} and its special cases F_1 and F_2.

4.4.2 The $O(4.83^k + poly(m, n))$-Algorithm

In this algorithm we distinguish two cases: (i) the character graph G contains c-vertices of degree at least three, and (ii) every c-vertex of G has degree two. Note that every c-vertex of degree one is removed by our data reduction. In Case (i), we use the branching strategy described in Section 4.4.2.1. In Section 4.4.2.2, we show that in Case (ii) the Flip Consensus Tree Problem can be solved in polynomial time.

4.4.2.1 Solving instances with c-vertices of degree at least three

Before describing the branching strategy that is applied as long as there are c-vertices of degree three or higher, we consider the following structure of intersecting M-graphs, called *big M-graph*: A big M-graph is a subgraph of the character graph and consists of two c-vertices c_i, c_j and t-vertices in the nonempty sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$, where at least one of these sets has to contain two or more t-vertices. Figure 4.6 illustrates a big M-graph F_M and its two special cases F_1 and F_2.

The following lemma states the existence of big M-graphs in a character graph containing c-vertices of degree at least three.

Lemma 4.6. *A character graph G that is reduced with respect to our data reduction rules, has a c-vertex of degree at least three if and only if G contains big M-graph (see Fig. 4.6) as an induced subgraph.*

Proof. Assume that there are c-vertices in G with degree at least three. Let c_i be a c-vertex with maximum degree in G. Then, c_i must have a c-neighbor c_j, which has at least one neighbor t_j outside $N(c_i)$ because otherwise c_i would be removed from G by the data reduction. Let t_k be a common neighbor of c_i and c_j that has to exist since c_i and c_j are c-neighbors. There also exists a neighbor t_i of c_i that is not a neighbor of c_j. If t_i is the only t-vertex that is a neighbor of c_i but not of c_j, then c_i and c_j must share another common neighbor t'_k besides t_k (since t_i has degree at least three) and the M-graph $t_i c_i t_k c_j t_j$ and edges $\{c_i, t'_k\}, \{c_j, t'_k\}$ form an F_2 graph. Otherwise let t'_i be an other t-vertex, which is a neighbor of c_i but not of c_j, the M-graph $t_i c_i t_k c_j t_j$ and the edge $\{t'_i, c_i\}$ form an F_1 graph. We conclude that if G has t-vertices with degree at least three, then G contains F_1 or F_2 as induced subgraph.

If G contains F_1 or F_2 as induced subgraph, it is obvious that G has c-vertices of degree at least three.

Since the graphs F_1 and F_2 are special cases of big M-graph, a character graph G that is reduced with respect to the abovementioned data reduction rules contains big M-graph as induced subgraph if and only if it contains c-vertices of degree at least three. □

In view of Lemma 4.6, after exhaustively executing the reduction rules, any character graph with at least one c-vertex of degree three or higher has to contain big M-graphs as induced subgraphs. Furthermore, it should be clear that a big M-graph contains many M-graphs as induced subgraphs. Therefore, if there are big M-graphs in the character graph, our algorithm first branches into subcases to eliminate all M-graphs contained in those big M-graphs. The branching strategy to eliminate all M-graphs contained in a big M-graph is based on the following lemma:

Lemma 4.7. *If a character graph G is M-free, then for every two distinct c-vertices c_i, c_j of G it holds that at least one of the sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ must be empty.*

Since there is at least one M-graph containing c_i and c_j if $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ are simultaneously non-empty, the correctness of Lemma 4.7 is obvious.

Lemma 4.7 leads to the following branching strategy for big M-graphs. Given a character graph G with at least one c-vertex of degree three or higher, our algorithm chooses a big M-graph F_{M} in G and branches into subcases to eliminate all M-graphs in F_{M}. Let c_i, c_j be the c-vertices of F_{M}. According to Lemma 4.7, one of the sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ must be "emptied" in each subcase. Let x, y, z denote the cardinalities of sets $X(c_i, c_j), Y(c_i, c_j)$ and $Z(c_i, c_j)$. We now describe how to empty $X(c_i, c_j)$, $Y(c_i, c_j)$, and $Z(c_i, c_j)$.

For each t-vertex t in $X(c_i, c_j)$, there are two possibilities to remove it from $X(c_i, c_j)$ by one edge modification: we either disconnect t from the big M-graph by deleting the edge $\{c_i, t\}$, or move t to $Y(c_i, c_j)$ by inserting the edge $\{c_j, t\}$. Therefore, the algorithm has to branch into 2^x subcases to empty $X(c_i, c_j)$ and in each subcase, it executes x edge modifications. The set $Z(c_i, c_j)$ is emptied analogously.

To empty the set $Y(c_i, c_j)$, there are also two possibilities for each t-vertex t in $Y(c_i, c_j)$, namely moving it to $X(c_i, c_j)$ by deleting the edge $\{c_j, t\}$ or moving it to $Z(c_i, c_j)$ by deleting the edge $\{c_i, t\}$. This also leads to 2^y subcases and in each subcase, y edge modifications are executed.

Figure 4.7 illustrates our branching strategies on an F_1 induced subgraph.

Altogether, the algorithm branches into $2^x + 2^y + 2^z$ subcases when dealing with a big M-graph F_{M}. In view of Lemma 4.7, at least $\min\{x, y, z\}$ edge modifications must be executed to eliminate all M-graphs in F_{M}. In the worst case, each edge modification has weight 1 and our branching strategy has branching vector

$$(\underbrace{x, \ldots, x}_{2^x}, \underbrace{y, \ldots, y}_{2^y}, \underbrace{z, \ldots, z}_{2^z}).$$

This leads to a branching number of 4.83 as shown in the following lemma.

Lemma 4.8. *The worst-case branching number of the above branching strategy is* 4.83.

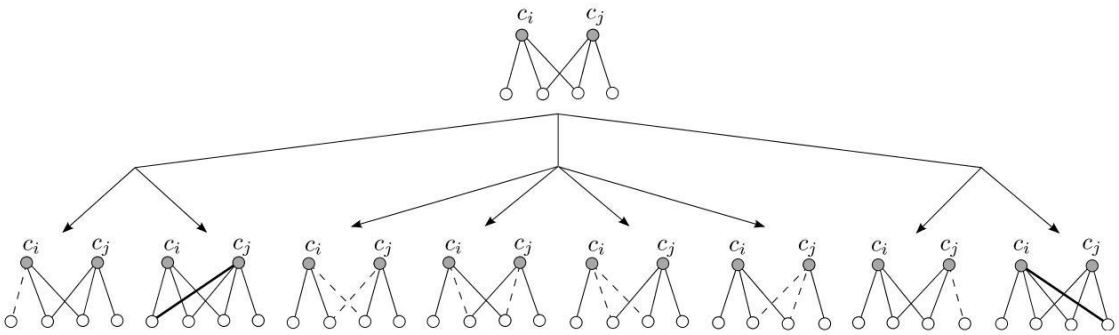


Figure 4.7: Branching strategy on an F_1 forbidden subgraph. Bold edges denote inserted edges and dash edges denote deleted edges.

Proof. The branching number b of the above branching strategy is the single positive root of the equation

$$2^x \frac{1}{b^x} + 2^y \frac{1}{b^y} + 2^z \frac{1}{b^z} = 1 \iff \frac{1}{(b/2)^x} + \frac{1}{(b/2)^y} + \frac{1}{(b/2)^z} = 1.$$

If we consider $\frac{b}{2}$ as a variable, the single positive root of the second equation is the branching number corresponding to the branching vector (x, y, z). The smaller values x, y, z take, the higher $\frac{b}{2}$ and, hence, b. Due to the definition of a big M-graph, x, y, and z cannot equal one simultaneously, so b is maximal if one of the variables x, y, z equals two and the other two equal one.

However, if either x or z is one and y is two (see the big M-graph F_2 in Figure 4.6 for an illustration), one of the two c-vertices has degree two. According to data reduction Rule 4, we do not insert edges to c-vertex of degree two. In this case, our branching strategy has a branching number of $(1, 1, 2, 2, 2, 2)$ and the corresponding branching number is 3.24.

Thus, to upper-bound the worst-case branching number of our branching strategy, we assume that $x = z = 1$ and $y = 2$. See the big M-graph F_1 in Figure 4.6 for an illustration. In this case, the single positive root of the equation $(\frac{2}{b})^2 + \frac{2}{b} + \frac{2}{b} = 1$ is $\frac{b}{2} = 2.414214$. Therefore, b is at most 4.83. □

From Lemma 4.7, we infer an interesting property.

Corollary 4.9. *There is no solution with at most k flips if there exist two c-vertices $c_i, c_j \in V_c$ satisfying* $\min\{|X(c_i, c_j)|, |Y(c_i, c_j)|, |Z(c_i, c_j)|\} > k$.

We use this property for pruning the search tree in the implementation of our algorithm, see Section 4.5.

Corollary 4.9 implies that we can abort a program call whenever we find a big M-graph where x, y, z simultaneously exceed k. Furthermore, if one or two of the values x, y, z are greater than k, we do not branch into subcases deleting the respective sets. Anyway, the number of subroutine calls in this step of the algorithm is fairly large, up

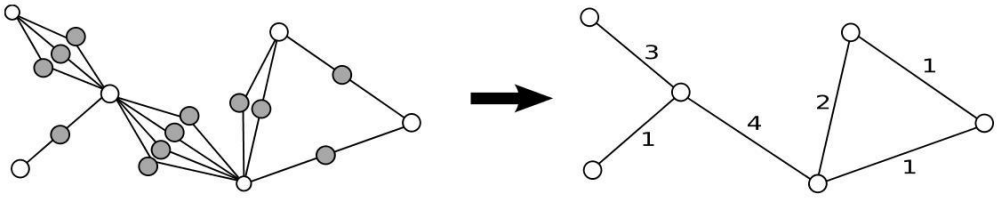


Figure 4.8: Left: When all c-vertices (gray) have degree two, we can regard each c-vertex with its incident edges as a single edge in a multigraph whose vertex set is the set of t-vertices (white). Right: We merge all those "edges" between two t-vertices into a single weighted edge whose weight equals the number of c-vertices adjacent to the t-vertices, and we obtain a simplified model of our graph.

to $3 \cdot 2^k$. But large numbers of program calls at this point are a result of large numbers of simultaneous edge modifications, which lower k to a greater extent. For simplicity, assume that $x = y = z = k$, we see that our algorithm branches in $3 \cdot 2^k$ subcases, but at the same time, it consumes all available flips. Therefore, the depth of the corresponding search tree is 1 and its size is $3 \cdot 2^k$. Thus, the branching number of our branching strategy goes to 2 for large x, y, z and this is confirmed by the growth of running times in our computational experiments (see Section 4.6).

4.4.2.2 Solving instances with c-vertices of degree at most two in polynomial time

In this section, we assume that the character graph G is reduced with respect to our data reduction and every c-vertex in G has degree at most two. Since every c-vertex of degree one is removed by our data reduction, every c-vertex in G has degree two, and there is a optimal solution for the Flip Consensus Tree Problem without inserting new edge into G.

In the following, we build a weighted undirected graph G_w from the character graph G and prove that with aid of G_w, the Flip Consensus Tree Problem can be solved in polynomial time. The weighted graph G_w is constructed as following:

We adopt the set V_t of t-vertices in G as vertex set for G_w. Two vertices t_1, t_2 are connected if and only if they possess a common neighbor in G. The weight of an edge $\{t_1, t_2\}$ is the total weight of the edges connecting t_1 (or t_2) with the common neighbors of t_1 and t_2 in the bipartite graph G, see Fig. 4.8. Furthermore, one can easily recognize that the number of edges in G_w is bounded by m, the number of character vertices. Thus, G_w contains at most n vertices and m edges.

One can easily see that there is an M-graph in G if and only if there is a path of length two in G_w.

According to Corollary 4.4, we never have to insert edges into the character graph G. Since deleting a weighted edge in G_w corresponds to deleting one of the edges incident to each of the respective c-vertices in G, the Flip Consensus Tree Problem turns out to be the problem of deleting a set of edges with minimum total weight such that

there are no paths of length two in G_w. Thus, the remaining graph is a *matching*, that is a set of edges without common vertices. Moreover, one can easily see that the larger the weight of the remaining edges, the smaller the weight of the deleted edges. Therefore, the Flip Consensus Tree Problem becomes the problem of finding a *maximum weighted matching*. In [63], Gabow introduced an algorithm to compute the maximum weighted matching of a graph (V, E) in time $O(|V||E| + |V|^2 \log(|V|))$. Since the number of edges in G_w is bounded by m, using Gabow's algorithm [63], we can solve the Flip Consensus Tree Problem in time $O(nm + n^2 \log n)$ if every c-vertex is of degree at most two.

Theorem 5. *The above algorithm solves the* Flip Consensus Tree Problem *in time* $O(4.83^k\,(m+n) + mn + n^2 \log n))$.

Proof. At the beginning of the algorithm, the execution of the data reduction takes $O(mn)$ time. By saving the degree of each vertex in G, the algorithm needs $O(m+n)$ time to execute the data reduction in each recursion call. After exhaustively executing our data reduction, we consider the following cases:

- If there is a c-vertex of degree at least three, G contains a big M-graph as induced subgraph, as shown in Lemma 4.6. After identifying a big M-graph in G, we use the branching strategy with branching number 4.83 described in Section 4.4.2.1 to branch into subcases eliminating the big M-graph. In the following, we describe how to identify a big M-graph efficiently.

  Let c_{max} denote the c-vertex with maximum degree in G. It holds that c_{max} has degree at least three and it has a c-neighbor, whose neighborhood contains t-vertices outside $N(c_{max})$. Therefore, c_{max} forms together with this c-neighbor and their neighborhoods a big M-graph. To find a big M-graph, we have to identify c_{max} and this c-neighbor of c_{max}.

  At the beginning of the algorithm, when computing the degree of every c-vertex, we notice the c-vertex c_{max} and its neighborhood $N(c_{max})$. When computing the degree of every t-vertex in $N(c_{max})$, we also identify the c-neighborhood of c_{max}. Afterwards, when computing the degree of every t-vertex, that is not contained in $N(c_{max})$, we can identify a c-neighbor c of c_{max} with $N(c) \neq N(c_{max})$. It holds that c_{max} forms together with c and their neighborhoods a big M-graph. All in all, a big M-graph can be found time $O(mn)$ during computing the degree of every vertex of G.

  During the course of the depth-bounded search tree algorithm, we update the information about c_{max}, $N(c_{max})$ and the c-neighborhood of c_{max} after every edge modification. Note that this can be done in linear time. To identify a big M-graph during the course of the depth-bounded search tree algorithm, we test for each t-vertex outside $N(c_{max})$, if a c-vertex in its neighborhood is a c-neighbor of c_{max}. We consider every c-vertex at most once in this test. Thus, this can be done in time $O(m+n)$.

- If very c-vertex has degree at most two, we construct the weighted graph G_w as

described above in time $O(m+n)$ and apply the Gabow's algorithm in [63] to compute the weighted maximum matching in G_w in time $O(nm + n^2 \log n)$.

All in all, the running time of this algorithm is $O(4.83^k (m+n) + mn + n^2 \log n)$. □

In [24], instead of using Gabow's algorithm when every c-vertex has degree two, we used a branching strategy with branching number 1.62. This improved the theoretical running time of our algorithm to $O(4.83^k (m+n) + mn)$. However, it is advisable to use the Gabow's algorithm in practice. Moreover, we can use the interleaving technique [122] with the kernelization in [99], to improve the theoretical running time of our algorithm to $O(4.83^k + poly(m,n))$ as claimed.

4.4.3 The $O(4.42^k + poly(m,n))$-Algorithm

In this Section, we present an improved version of the algorithm introduced in the previous section. The main idea of the improvement bases on the observation that besides the case, where every c-vertex has degree two, there are other cases where the Flip Consensus Tree Problem can be solved in polynomial time. In this new algorithm, we use the branching strategy for big M-graph introduced in Section 4.4.2.1 as long as there is a c-vertex of degree at least three in G and none of the polynomial-time-solvable cases occurs. By an involved case analysis, we show in the following theorem that our new algorithm has running time $O(4.42^k (m+n) + mn + n^2 \log n)$.

Theorem 6. *The* Flip Consensus Tree Problem *can be solved in time* $O(4.42^k (m+n) + mn + n^2 \log n)$.

Proof. We now give an overview of the case differentiation of our new algorithm and show that it generates a search tree of size at most 4.42^k.

If there are c-vertices of degree at least four in G, our branching strategy for big M-graphs introduced in Section 4.4.2.1 has a branching number of at most 4.42 as shown in Lemma 4.10.

If every c-vertex in G is of degree at most three, we distinguish the following cases:

Case 1: Every c-vertex in G has degree two. The Flip Consensus Tree Problem can be solved in time $O(nm + n^2 \log n)$, as shown in Section 4.4.2.2.

Case 2: There is a c-vertex of degree two occurring in an M-graph. We show in Section 4.4.3.2 that our branching strategy for big M-graphs has a branching number of at most 4.24 for this case.

Case 3: Every c-vertex occurring in M-graph has degree three. We observe the following subcases:

- *There are two c-vertices with one common neighbor.* Our branching strategy for big M-graphs has a branching number of at most 4, see Section 4.4.3.2.
- *Every two c-vertices have two common neighbors.* The character graph G has to be a graph similar to the graph shown in Fig. 4.11 and the Flip Consensus Tree Problem for G can be solved in polynomial time, as shown in Lemma 4.11, or

G contains four t-vertices and at most four c-vertices as shown in Figure 4.12 and the problem can even be solved in constant time.

Thus, our algorithm generates a search tree of size at most $O(4.42^k)$. As mentioned in Section 4.4.2, the execution of data reduction takes $O(mn)$ time. By saving the degree of every vertex in G, our new algorithm also needs $O(m+n)$ time for execution the data reduction and the case differentiations described above in every recursion call. Thus, the running time of our algorithm is $O(4.42^k\,(m+n) + mn + n^2 \log n)$. □

We again can improve the theoretical running time of our algorithm by applying the interleaving technique [122] with the kernelization from [99]. This results in the running time $O(4.42^k + poly(m,n))$ as claimed.

In the remaining part of this section, we will prove the running time of our algorithm in detail.

4.4.3.1 Solving instances with c-vertices of degree at least four

We now assume that G contains a c-vertex of degree at least four. The following lemma holds.

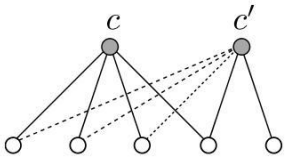
Lemma 4.10. *If there is a c-vertex of degree at least four in G, the branching strategy for big M-graphs described in Section 4.4.2.1 has branching number* 4.42 *when branching into subcases eliminating a big M-graph containing the c-vertex with the maximum degree.*

Proof. Assume that G contains a c-vertex of degree at least four. Let c be a c-vertex with maximum degree in G. It holds that c has degree at least four, and there must exist a c-vertex c' in the c-neighborhood of c with $N(c') \not\subseteq N(c)$, otherwise c is removed from G by data reduction Rule 1. In the following, we make a case analysis by means of the degree of c', and show that our branching strategy for big M-graphs described in Section 4.4.2.1 has branching number at most 4.42 when branching into subcases eliminating the big M-graph containing c and c'. Assume that c has degree four, so c' can have degree two, three, or four.

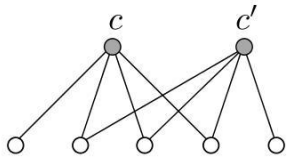
Case 1: c' has degree two. G contains the subgraph shown in Figure 4.9(a) as an induced subgraph. Since every pair $\{c', t\}$ for $t \notin N(c')$ is set to forbidden by the data reduction, our branching leads to the branching vector $(3,1,1,1,1)$, which corresponds to the branching number 4.07.

Case 2: c' has degree three. We distinguish two subcases: c and c' have one common neighbor, or c and c' have two common neighbors, see Figure 4.9(c). Branching on the big M-graph induced by c and c', our branching strategy results in branching vector $(3,3,3,3,3,3,3,3,1,1,2,2,2,2)$ with branching number 3.68, see Figure 4.9(c) left; or branching vector $(2,2,2,2,2,2,2,2,1,1)$ with branching numbers 4, see Figure 4.9(c) right.

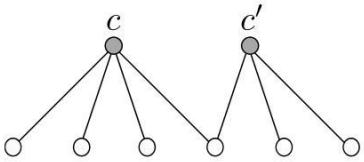
Case 3: c' has degree four. For the cases that c and c' have one or two common neighbors, the branching number of our branching strategy is dominated by the corresponding cases

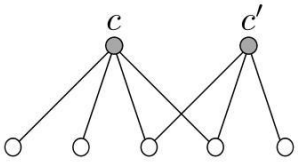


(a) Dashed edges denote forbidden edges.



(b) c' has degree four, c and c' have three common neighbors.





(c) *Left:* c and c' have one common neighbor. *Right:* c and c' have two common neighbors.

Figure 4.9: c-vertex of degree at least four in G

where c' has degree three. If c and c' have three common neighbors, G contains the subgraph shown in Figure 4.9(b) as induced subgraph and our branching strategy has a branching number of 4.42.

If c has degree five or more, c' can have degree larger than four, but the big M-graph induced by c, c' and their neighborhood contains one of the big M-graphs analyzed above as induced subgraph. This leads to the branching number of 4.42. □

Since the degrees of all vertices in G are known, the c-vertex with maximum degree and the big M-graph containing this c-vertex can be found in $O(m+n)$ time, as shown in the proof of Theorem 5.

4.4.3.2 Solving instances with c-vertices of degree at most three

In the following, we describe how to solve problem instances where all c-vertices have degree at most three. If every c-vertex is of degree at most two, the problem can be solved in polynomial time as described in Section 4.4.2.2. We now assume that there are c-vertices of degree three in G.

Assume that there exists a c-vertex c of degree two that occurs in an M-graph. Let c' be the other c-vertex in this M-graph. Vertex c' can have degree two or degree three. In any case, we do not have to insert edges incident to c, since c is of degree two. We can infer the following: If c' has degree two, we do not have to insert edges, so the branching strategy for big M-graphs has branching vector $(1,1,1,1)$ with branching number 4, see Figure 4.10 left. If c' has degree three, this branching strategy has branching vector $(1,1,1,1,2)$ that leads to a branching number of 4.24, see Figure 4.10 right.

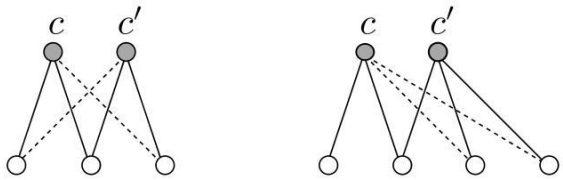


Figure 4.10: Every c-vertex has degree at most three and a c-vertex of degree two occurs in an M-graph. Dashed edges will not be inserted by the branching strategy.

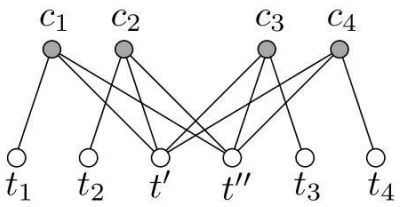


Figure 4.11: t-vertices t', t'' are the two common neighbors of all c-vertices.

In the following, we assume that every c-vertex that occurs in an M-graph has degree three. It follows that every c-vertex of degree two must have either zero or two common neighbors with other c-vertex. If there are two c-vertices c, c' with one common neighbor, it holds that $|X(c,c')| = 2$, $|Y(c,c')| = 1$ and $|Z(c,c')| = 2$. When branching into subcases that eliminate the big M-graph containing c and c', the branching strategy introduced in Section 4.4.2.1 results in the branching vector $(2,2,2,2,1,1,2,2,2,2)$, which corresponds to a branching number of 4.

We consider the last case where *every pair of c-vertices in G must have two or three common neighbors*. Since each c-vertex has three neighbors, all c-vertices with three common neighbors have the same neighborhood and are thus merged into a single c-vertex by our data reduction. This implies that *every pair of c-vertices have two common neighbors*. This case can again be divided into two subcases:

Case 1: There are two t-vertices that are connected to all c-vertices in G. Let t', t'' be these two t-vertices. An arbitrary c-vertex c_i must have t', t'' and a t-vertex t_i in his neighborhood and the neighborhood of t_i contains only c_i, since c-vertices with the same neighborhood are merged into a single one. See Figure 4.11. According to the following lemma, the Flip Consensus Tree Problem can be solved in polynomial time in this case.

Lemma 4.11. *If every c-vertex in G has degree at most three and the neighborhood of every c-vertex c_i with degree three shares two common t-vertices t' and t'', i.e. $N(c_i) = \{t_i, t', t''\}$ (see Fig. 4.11), then there is an optimal solution that deletes all edges $\{c_i, t_i\}$ except for one edge $\{c_{max}, t_{max}\}$ with $w(c_{max}, t_{max}) = \max_i\{w(c_i, t_i)\}$.*

Proof. Any c-vertex of degree two can only have t' and t'' in his neighborhood and never occurs in an M-graph even if we delete any edge (c_i, t_i). Recall that all edge modifications associated with the same c-vertex must have an identical cost. Let w_i denote the cost

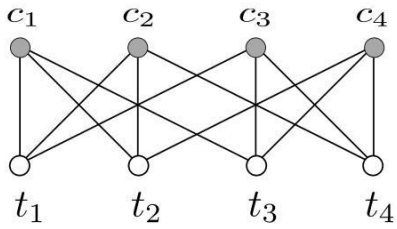


Figure 4.12: Every c-vertex has degree three, and every two c-vertices have two common neighbors, and there are no two t-vertices that are neighbors of all c-vertices.

of edge modifications associated with c-vertex c_i. Without loss of generality, we assume that $w_i \leq w_j$ if $i < j$. We prove this lemma by induction.

If there are only two c-vertices c_{m-1} and c_m in G, deleting $\{c_{m-1}, t_{m-1}\}$ is clearly an optimal solution for the minimum-flip consensus trees in G.

Let G' be the subgraph of G without c_1. Assume that the optimal solution for G' deletes all edges $\{c_i, t_i\}$ for $2 \leq i \leq m-1$. The cost of the optimal solution for G' is $k_{opt}(G') = \sum_{i=2}^{m-1} w_i$.

Assume that there is an optimal solution with cost $k_{opt}(G) < k_{opt}(G') + w_1$ for G. Let G_{opt} be the output graph of this optimal solution for G. In the following we show that there is a solution for G' with cost smaller than $k_{opt}(G')$, which is a contradiction to the assumption that the cost of the optimal solution for G' is $k_{opt}(G')$.

A solution for G' can be derived from G_{opt} by undoing edge modifications associated with c_1, if some is executed, and then removing c_1 from G_{opt}. Clearly, the resulting graph is a solution for G'.

If the optimal solution for G did execute some edge modifications associated with c_1, undoing these edge modifications reduces $k_{opt}(G)$ to at most $k_{opt}(G) - w_1$, which is smaller than $k_{opt}(G')$. This is a contradiction to the assumption that the cost of the optimal solution for G' is $k_{opt}(G')$.

If no edge modification associated with c_1 was executed by the optimal solution for G, at least one modification associated with each c_i, for $2 \leq i \leq m$, had to be executed to eliminate all (big) M-graphs containing c_1 and c_i. Therefore, the cost $k_{opt}(G)$ of the optimal solution for G can be lower-bounded by $\sum_{i=2}^{m} w_i$, whereby deleting all edges $\{c_i, t_i\}$ for $1 \leq i \leq m-1$ results in a solution for G with cost $\sum_{i=1}^{m-1} w_i \leq \sum_{i=2}^{m} w_i \leq k_{opt}(G)$. Since the equality holds if and only if $w_i = w_j$ for $1 \leq i, j \leq m$, there is no solution for G with cost smaller than $\sum_{i=2}^{m-1} w_i + w_1 = k_{opt}(G') + w_1$, and since our algorithm computes a solution for G with this cost, it computes the optimal solution for G. □

Case 2: There are no two t-vertices that are common neighbors of all c-vertices.

In the following, we show that in this remaining case, the graph G consists of four t-vertices and at most four c-vertices as shown in Fig. 4.12.

Since we merge c-vertices with the same neighborhood, there are no c-vertices with three common neighbors in G. If G contains at least two c-vertices, then G must also

contain at least four t-vertices, otherwise the two c-vertices are merged by our data reduction. Suppose G contains at least five t-vertices. Let $\{t_1, t_2, t_3, t_4, t_5\}$ be five t-vertices in G. Since every c-vertex in G is of degree three and every two c-vertices in G have two common neighbors, there must be at least three c-vertices in the neighborhoods of these five t-vertices. Let c_1, c_2, c_3 be these three c-vertices, where $N(c_1) = \{t_1, t_2, t_3\}$ and $N(c_2) = \{t_2, t_3, t_4\}$. Obviously, t_5 has to belong to $N(c_3)$. Since every two c-vertices in G must have two common neighbors, the others two neighbors of c_3 must be t_2, t_3. If G does contain further c-vertices, t_2, t_3 must be common neighbors of every c-vertex in G by the same argumentation. This implies that t_2, t_3 are common neighbors of every c-vertex and we are done. Therefore, we may assume that, G contains only four t-vertices. Since every c-vertex in G is of degree three, there are at most $\binom{4}{3}$ c-vertices in G after merging c-vertices with the same neighborhood. See Fig.4.12 for an illustration of this case. Thus, G contains only four t-vertices and at most four c-vertices and the Flip Consensus Tree Problem in G is solvable in constant time. This finishes the tedious case analysis of the proof that our algorithm solves the Flip Consensus Tree Problem in time $O(4.42^k\,(m+n) + mn + n^2 \log n)$.

In the following, we discuss an idea that may improve the theoretical running time of our algorithm, but has not been tested yet.

Improving the 4.42^k algorithm. One of the most obvious ideas to improve the 4.42^k algorithm is to improve our branching strategy for the case, where it has the worst-case branching number of 4.42. As shown in the proof of Lemma 4.10, our branching strategy has branching number 4.42 when branching on a big M-graph containing the c-vertex c with maximum degree and the c-neighborhood c' of c, where c and c' simultaneously have degree four (see Figure 4.9(b)) and three common neighbors. If there is an other c-neighbor of c or c' that forms one of the big M-graphs shown in Figures 4.9(a) and 4.9(c) with c or c', our branching strategy has a branching number of at most 4.07 when branching on this big M-graph. Assume that such c-neighbor of c or c' does not exist, it holds that every c-neighbor of c and c' has degree four and three common neighbors with c as well as with c'. We believe that in this case, the Flip Consensus Tree Problem can again be solved in polynomial time, analogously to the cases shown in Figures 4.11 and 4.12. Thus, our branching strategy has worst-case branching number 4.24, and this happens when branching on the big M-graph shown in Figure 4.10(left).

Conjecture 1. *The* Flip Consensus Tree Problem *can be solved in time* $O(4.24^k + poly(m, n))$.

Furthermore, we believe that using the technique introduced by Gramm *et al* [69] that generates search trees automatically, might lead to a search tree algorithm with better *theoretical* running time. However, algorithms that are generated automatically, are mostly complicated and do not have practical applicability.

4.5 Algorithm Engineering

In the course of algorithm design, we found some improvements that do not affect the theoretical worst-case running time or even increase the polynomial factor but as they manage to prune the search tree, they are highly advisable in practice. These are a few heuristic improvements we included in our implementation.

Treat connected components separately. As we already mention in the previous part of this chapter, if a given character graph is not connected or decomposes during the execution of the algorithm, we compute the solutions for each of its connected components separately because connecting different connected components never deletes an M-graph.

Avoid futile program calls. If there are two c-vertices c_i, c_j such that

$$\min\{|X(c_i, c_j)|, |Y(c_i, c_j)|, |Z(c_i, c_j|)\} > k$$

holds, we know that it is impossible to solve the current instance (see Corollary 4.9). Therefore, whenever we find such a big M-graph we abort the current search tree branch and call the algorithm with an appropriately increased parameter, thus skipping program runs that are doomed to failure. Furthermore, as mentioned in Section 4.4.2, if one or two of the values x, y, z are greater than k, we do not branch into subcases deleting the respective sets.

Try promising search tree branches first. In the first part of our branching strategy, branching on a big M-graph F_{M} with c-vertices c_i, c_j leads to $2^{|X(c_i,c_j)|} + 2^{|Y(c_i,c_j)|} + 2^{|Z(c_i,c_j)|}$ branches. It is likely that a minimum-size solution destroys the big M-graph with as few edge modifications as possible. As we use depth-first search and stop when we find a solution, we branch on the edges incident to the smallest of sets X, Y, Z first.

Calculate branching numbers in advance. When dealing with big M-graphs, for each pair of c-vertices, we save the branching number corresponding to a branching at the big M-graph associated with these vertices in a matrix. The minima of each row are saved in an extra column to allow faster searching for the overall minimum. We use a similar technique to deal with the weighted graph in the second part of the algorithm.

Since initializing the matrix used to calculate the branching numbers takes time $O(m^2n)$ and updating this matrix needs time $O(mn)$ in each recursive call, the polynomial factor in the running time proved in Theorem 5 and Theorem 6 cannot hold when applying the abovementioned heuristic improvements. While initializing or updating this matrix, we also check if the data reduction rules can be applied. Testing for futile program calls and redundant search tree branches can be executed in the same time. Altogether, the running time of our algorithms with the abovementioned heuristic

improvements are $O(4.83^k\, mn + m^2n + n^2 \log n)$ and $O(4.42^k\, mn + m^2n + n^2 \log n)$, respectively. Despite that, the heuristic improvements lead to drastically reduced running times in practice.

Due to the expense of development as well as a hidden large polynomial factor of the kernelization in [99], we did not implement this kernelization.

4.6 Computational Results

In this chapter, we have presented two fixed-parameter algorithms with running time $O(4.83^k\,(m+n) + mn + n^2 \log n)$ and $O(4.42^k\,(m+n) + mn + n^2 \log n)$ for the Flip Consensus Tree Problem. The only difference between the 4.83^k algorithm and the 4.42^k algorithm consists of dealing with the special cases of G shown in Figures 4.11 and 4.12, the 4.42^k algorithm uses the corresponding polynomial running time procedures instead of using the branching strategy for big M-graph. Since these special cases do not often occur in practice, we do not believe that the 4.42^k algorithm is faster than the 4.83^k algorithm in practice. Therefore, we implemented our 4.83^k algorithm, and compared it against Chen *et al.*'s 6^k algorithm [38]. Both algorithms were implemented in Java. Computations were done on an AMD Opteron-275 2.2 GHz with 6 GB of memory running Solaris 10.

Each program receives a binary matrix as input and returns a minimum set of flips needed to solve the instance. We start the program repeatedly and increase k by one until a solution is found. As soon as we find a solution with at most k flips, the program is aborted instantly and the solution is returned without searching further branches. All data reduction rules and heuristic improvements described in Section 4.4.1 and Section 4.5 were implemented and, where applicable, also used in the 6^k algorithm. For our experiments, we used matrix representations of real phylogenetic trees, namely two phylogenetic trees of marsupials with 21 and 51 taxa (data provided by Olaf Bininda-Emonds) and one tree of 97 bacteria computed using Tex protein sequences (data provided by Lydia Gramzow). Naturally, these matrices admit perfect phylogenies.

We perturbed each matrix by randomly flipping different numbers of entries, thus creating instances where the number of flips needed for resolving all M-graphs in the corresponding character graph is at most the number of perturbations. For each matrix representation and each number of perturbations, we created ten different instances and compared the running times of both algorithms on all instances. In many datasets we created, it was possible to solve the instance with a smaller number of flips.

Each instance was allowed ten hours of computation. For the largest instances with only ten flips, we ignored these running time constraints to show the performance of the 6^k algorithm. The results of the computations are shown in Table 4.1. When the average running time was below ten hours, all instances were finished in less than ten hours. When the average was more than ten hours, all ten instances took more than ten hours, except for the small Marsupial datasets with $k = 12$ for the 6^k algorithm and the Tex datasets with $k = 20$ for our algorithm. In both cases, six of ten instances were solved.

Dataset	$\|V_t\|$	$\|V_c\|$	# flips	avg. k	time 4.83^k	time 6^k
Marsupials	21	20	10	9.6	9.5 s	2 h
			12	10.7	25.5 s	$>$ 10 h*
			14	13.2	3 min	$>$ 10 h
			16	15.4	12 min	$>$ 10 h
			18	17	47 min	$>$ 10 h
			20	18.9	3.3 h	$>$ 10 h
Marsupials	51	50	10	10	17 s	19 h
			12	10.5	30 s	$>$ 10 h
			14	12.5	2.3 min	$>$ 10 h
			16	15.5	50 min	$>$ 10 h
			18	17.5	3 h	$>$ 10 h
			20	19	8 h	$>$ 10 h
Tex (Bacteria)	97	96	10	9.7	17 s	59.3 h
			12	11.9	12.5 min	$>$ 10 h
			14	13.9	18 min	$>$ 10 h
			16	15.4	1.1 h	$>$ 10 h
			18	17.3	4 h	$>$ 10 h
			20	19.7	10.3 h*	$>$ 10 h

Table 4.1: Comparison of average running times of our 4.83^k algorithm and the 6^k algorithm. $|V_t|$ and $|V_c|$ denote the number of t- and c-vertices, respectively. # flips is the number of perturbations in the matrix, whereas k is the true number of flips needed to solve the instance. Each row corresponds to ten datasets. *Six out of ten computations were finished in less than ten hours.

Our experiments show that our method is significantly faster than Chen *et al.*'s algorithm for all instances. We observed that, on average, increasing k by one resulted in about 2.2-fold running time for a program call of our algorithm and 5-fold running time for the 6^k algorithm. We believe that the reason for the factor of 2.2 is that big M-graphs can be fairly large in practice, so that the real branching number is close to two as analyzed in Section 4.4.2.1.

4.7 Summary and Outlook

In this section, we summarize our results for the Flip Consensus Tree Problem presented in this work and sketch some ideas for future research regarding the Flip Consensus Tree Problem and particularly the Flip Supertree Problem, which is a generalization of the Flip Consensus Tree Problem and more important from the biological point of view.

In Section 4.4.1, we introduced a set of data reduction rules, which do not yield

a problem kernel, but can be executed efficiently. Furthermore, our data reduction rules grant an important property of character graph, that allows for better branching strategy: When a character graph G is reduced with respect to our data reduction rules, we can make sure that G contains at least one big M-graph as induced subgraph if there is a c-vertex of degree at least three in G (see Lemma 4.6). Due to this property, we can use the branching strategy for big M-graphs, when G contains a c-vertex of degree at least three and apply a maximum matching algorithm to solve the Flip Consensus Tree Problem when every c-vertex in G is of degree two.

In Sections 4.4.2 and 4.4.3, we presented two depth-bounded search tree algorithms for the Flip Consensus Tree Problem with running time $O(4.83^k\, mn + n^2 \log n)$ and $O(4.42^k\, mn + n^2 \log n)$, which are our main results in this chapter. In Section 4.5, we discussed some heuristic improvements to reduce the running time of our algorithms in practice. To evaluate the performance of our algorithm, we implemented our 4.83^k algorithm and compared it against the 6^k algorithm of Chen *et al.* [38]. As reported in Section 4.6, our 4.83^k algorithm turns out to be significantly faster than the 6^k algorithm. In particular, the implementation of our algorithm is also much faster than the worst case running time suggests. Furthermore, the practical running time of the 4.83^k algorithm also empirically supports our conjecture in Section 4.4.2.1, that the real branching number of our branching strategy for big M-graph is close to two.

Future Research

Although our algorithm for the Flip Consensus Tree Problem is much faster than the straightforward 6^k-algorithm and far better than one would expect from the worst-case analysis, it is still too slow for practical use. To address this issue, we want to develop more efficient data reduction rules to further downsize problem instances. Furthermore, we also want to improve our branching strategy, particularly the branching strategy for big M-graphs. While this branching strategy executes many edit operations in each recursive call, which appear to positively affect the running times, it also generates exponentially many branching cases, which in turn increase the running time of our algorithm drastically.

However, our main focus in future research is to cope with the Flip Supertree Problem. In contrast to Flip Consensus Tree Problem instances, the input matrix of an Flip Supertree Problem instance contains large blocks of '?'-entries, which can be set to '1' or '0' without paying edit cost. Because of the '?'-entries, the concept of character graphs does not work for the Flip Supertree Problem. In fact, we recently showed that the Flip Supertree Problem is W[2]-hard with respect to the number of flips [21]. Therefore, one cannot hope for a fixed-parameter algorithm with respect to the number of flips. In the following, we list some approaches that may be used to cope with the W[2]-hardness of the Flip Supertree Problem:

- First we would like to improve our algorithm for the Flip Consensus Tree Problem such that it is capable to solve problem instances of larger sizes that requires more flips. Then, we can use a heuristic algorithm, such as the heuristic

algorithm in [56], to replace the '?'s in the input matrix of the Flip Supertree Problem with '1's or '0's and apply our algorithm to the resulting Flip Supertree Problem instance. We hope that this would lead to a better solution than only using the heuristic algorithm.

- To solve the Flip Supertree Problem exactly, we would like to develop further data reduction rules to cut down the size of the input and use the integer linear programming model of the Flip Supertree Problem introduced in [41] to solve the Flip Supertree Problem exactly.

- From the fixed-parameter tractability point of view, we want to investigate if the Flip Supertree Problem is fixed-parameter tractable with respect to another meaningful parameter, for example, the maximum number of '?' in a column or a row. Furthermore, it is also an interesting question if the Flip Supertree Problem is fixed-parameter tractable with respect to the number of flips but under another meaningful restriction than forbidding '?'s in the input matrix.

To conclude this chapter, we would like to emphasize that although the Flip Consensus Tree Problem is not as biologically relevant as the Flip Supertree Problem, it is an interesting problem from the theoretical point of view, and definitely worthwhile to be investigated. Furthermore, some of our observations when studying the Flip Consensus Tree Problem might be conducive to investigating the more difficult but also biologically more meaningful Flip Supertree Problem.

5 Weighted Cluster Editing

In this chapter, we investigate the Weighted Cluster Editing Problem, which asks for a set of edge modifications with minimum cost to transform a graph into a *cluster graph*, which is a union of disjoint cliques. Weighted Cluster Editing Problem is a graph modification problem with many biological applications. We list some of those applications in Section 5.1. In theoretical computer science, the unweighted counter part of this problem has been well-studied, in particular with respect to its fixed-parameter tractability. In Section 5.2, we recapitulate previous results for the Weighted Cluster Editing Problem and Unweighted Cluster Editing Problem. In Section 5.3, we describe a fixed parameter tractable algorithm with running time $O(2.42^k + |V|^3 \log |V|)$ for the Weighted Cluster Editing Problem. Section 5.4 serves as a survey of further results achieved by our group. For more details, we referred interested readers to our original articles [16, 18].

5.1 Motivation

In the past, biologists used to consider each individual biological entity one-by-one to understand its properties and functions. Although this approach allows for high accuracy and, in fact, has led to many remarkable results, it cannot be applied to the huge amount of biological data available nowadays. Fortunately, in many cases, it is not necessary to consider each individual object in a large dataset. Thus, we can subdivide a large dataset into smaller number of categories containing similar objects, and analyse these categories instead of analysing each individual object in the large dataset. Each of such categories is called a *cluster*. Usually, it is required that objects in the same cluster must be of high similarity (*homogeneity*) and objects from different clusters have low similarity to each other (*separation*).

In fact, the task of clustering biological entities according to some similarity or distance measures has been often addressed. In [100, 130, 155, 156] clustering was used to reconstruct families of orthologous proteins. In gene expression data analysis, clustering is an important step to classify genes according to their co-expressions [12, 119, 140]. Furthermore, clustering also finds many applications in medicine [74, 79, 110].

There are lots of clustering approaches, such as k-means [113], hierarchical clustering [94], Markov clustering [55], affinity propagation [61, 107] etc. The choice of a clustering approach and an adequate similarity measure depends on the aim of the clustering and the kind of objects to be clustered. For examples, a pairwise sequence similarity score can be used as similarity measure, when clustering biological sequences; correlation coefficient of genes expression patterns is usually used as similarity measure when clustering gene expression data. As examples for clustering methods, k-means, Markov

clustering, affinity propagation and cluster editing are usually preferable to hierarchical clustering when hierarchical structure is not desired in resulting clustering.

In this work, we investigate the *cluster editing* approach. Given set of objects and their pairwise similarities, the cluster editing approach builds a graph $G = (V, E)$, where each vertex $v \in V$ represents an object and two vertices u, v are connected by an edge uv if the corresponding objects are highly similar, i.e., the similarity of corresponding objects must exceed a user-defined *affinity threshold*.

If G is a cluster graph, each connected component of G corresponds to a cluster containing objects of high pairwise similarity and G corresponds to a possible clustering of the given objects. Unfortunately, the resulting graph G is usually not a cluster graph. Thus, the cluster editing approach asks for a set of edge modifications, that are edge deletions and edge insertions, with minimum cost to transforms G into a cluster graph. The cost of an edge modification is usually calculated from the similarity of the corresponding objects and the affinity threshold, for example the distance different between these two values.

In [12], Ben-Dor *et al.* applied the cluster editing approach to clustering gene expression data. They reported several advantages of cluster editing approach:

1. The clustering result does not have a hierarchical structure. This implies that the resulting clusters are unrelated, and the cluster boundaries are determined by the applied algorithm, without human intervention.

2. The number of clusters does not have to be specified.

3. The affinity threshold is the only parameter that has to be determined by the user.

Wittkop *et al.* [156] used the cluster editing to reconstruct families of orthologous proteins, and reported that cluster editing outperforms other common clustering approaches such as Markov clustering [55], affinity propagation [61, 107] and hierarchical clustering [94] in this task.

5.2 Previous Work

In contrast to the Weighted Cluster Editing Problem, the Unweighted Cluster Editing Problem is well-studied in the theoretical computer science. The Unweighted Cluster Editing Problem asks for the minimum number of edge modifications to transform a graph into a cluster graph. The NP-completeness of Unweighted Cluster Editing Problem was shown by Křivánek and Morávek [103]. Since the Weighted Cluster Editing Problem generalizes its unweighted counterpart, it is also NP-hard, thus obviously NP-complete. In [150], van Zuylen *et al.* showed that the Unweighted Cluster Editing Problem is APX-hard and has a constant approximation factor of 2.5.

From the graph-theoretical point of view, Cluster Editing Problem (weighted as well as unweighted) belongs to the class of graph modification problems with a hereditary property characterized by a finite set of forbidden subgraph. Thus, the Unweighted

Cluster Editing Problem is fixed-parameter tractable with respect to the number of edge modifications (see Theorem 1): By a straightforward depth-bounded search tree algorithm, the Unweighted Cluster Editing Problem is solvable in time $O(3^k\,|V|^3)$, where k is the minimum number of edge modifications. In [70], Gramm *et al.* introduced a refined search tree algorithm for the Unweighted Cluster Editing Problem with running time $O(2.27^k + |V|^3)$. In [69], Gramm *et al.* used an automated branching rule generator to find a set of branching rules that lead to a depth-bounded search tree algorithm with running time $O(1.92^k + |V|^3)$ for the Unweighted Cluster Editing Problem. However, this result is only interesting from the theoretical point of view, since the branching rule with 137 initial cases make it difficult to implement and inefficient in practice.

In [70], Gramm *et al.* showed that the Unweighted Cluster Editing Problem has a problem kernel with at most $2(k^2 + k)$ vertices and $2\binom{k+1}{2}k$ edges. In [76], Guo *et al.* proved a $4k$-vertex kernel for the Unweighted Cluster Editing Problem. Recently, Chen and Meng [40] improved the kernelization of Guo *et al.* and proved a problem kernel with $2k$ vertices for the Unweighted Cluster Editing Problem.

For the Weighted Cluster Editing Problem, besides an integer linear program in [73], several heuristic algorithms have been introduced. In [140], Shamir *et al.* considered several clustering approaches including the *highly connected subgraphs* HCS method. The underlying combinatorial problem of this method can be considered as a relaxation of the Weighted Cluster Editing Problem. Shamir *et al.* also introduced a probabilistic version of the HCS approach, named (CLICK), and a polynomial time algorithm for this problem.

In [155], Wittkop *et al.* considered the Weighted Cluster Editing Problem and proposed to cluster the vertices by arranging them in a two-dimensional plane, such that vertices with high similarity are arranged close to each other and far away from other vertices.

Since the weighted and unweighted version of the Cluster Editing Problem are closely related, some of the fixed-parameter tractability results for Unweighted Cluster Editing Problem can be easily adapted to Weighted Cluster Editing Problem, for example the straightforward 3^k branching strategy and the trivial data reduction rules in [70]. However, the data reduction rules in [70] only results in a problem kernel for the Unweighted Cluster Editing Problem, but not for the Weighted Cluster Editing Problem.

To the best of our knowledge, the fixed-parameter tractability of Weighted Cluster Editing Problem was first intensively investigated by us in [18]. In [18], we presented several theoretical and practical results for the Weighted Cluster Editing Problem, which can also be applied to the Unweighted Cluster Editing Problem. In particular, it is advisable to apply our algorithm to the Unweighted Cluster Editing Problem, since it has the best theoretical as well as practical running time for both problems (at the time of preparing this work). In the remaining part of this chapter, we subsume our results presented in [18]. In particular, we describe our depth-bounded search tree algorithm with running time $O(2.42^k + |V|^3 \log |V|)$ in detail. For more details on our other results regarding the Weighted Cluster Editing Problem, interested

readers may take a look at [18].

5.3 Algorithms

In the following, we investigate the Weighted Cluster Editing Problem. We first formalize this problem and state a simple property of cluster graphs.

Definition 5.1 (Weighted Cluster Editing Problem)**.** Given an undirected graph $G = (V, E)$ in form of a weight function $s : \binom{V}{2} \to \mathbb{R}$: For $s(uv) > 0$, an edge uv is present in G and has deletion cost $s(uv)$, whereas for $s(uv) \leq 0$ the edge uv is absent from G and has insertion cost $-s(uv)$. The Weighted Cluster Editing Problem asks for a set of *edge modifications* that are *edge deletions* and *edge insertions*, with minimum total cost, to transform G into a cluster graph.

To achieve a provable running time, we require the modification cost of every vertex pair to be at least one, i.e., $|s(uv)| \geq 1$ for every vertex pair u, v.

Since every connected component of a cluster graph is a clique, the following lemma is obvious.

Lemma 5.1. *A graph G is a cluster graph if and only if G does not contain an induced path of length two, i.e., there are no three vertices v, u, w where uv and uw are edges in G but vw is not.*

We call a tuple vuw a *conflict triple* if uv and uw are edges and vw is absent in G. To transform a graph $G = (V, E)$ into a cluster graph, we have to eliminate every conflict triple in G.

5.3.1 A Straightforward Algorithm

In the following, we apply the depth-bounded search tree technique to solve Weighted Cluster Editing Problem. For that purpose, we consider the *parameterized* version of the Weighted Cluster Editing Problem that is defined as following.

Definition 5.2. Given an undirected graph $G = (V, E)$ described in Definition 5.1 and an integer k as input, the *parameterized* Weighted Cluster Editing Problem asks if it is possible to transform G into a cluster graph with total modification cost of at most k.

In the following, we describe a simple search tree algorithm for the parameterized Weighted Cluster Editing Problem. To compute optimal solutions for the original problem, we use this algorithm to solve the parameterized Weighted Cluster Editing Problem with $k = 1$. If the algorithm cannot find a solution with $k = 1$, we increase k by 1 and call the algorithm again, until arriving at the smallest k, for that it is possible to transform G into a cluster graph with modification cost of at most k.

Since a conflict triple vuw in G can be eliminated by inserting the absent edge vw or deleting uv or deleting uw, our search tree algorithm simply branches into those cases to

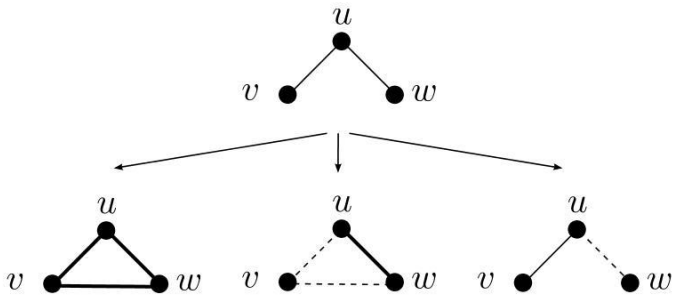


Figure 5.1: The initial branching strategy. Bold edges are permanent edges and dashed edges are forbidden edges.

eliminate the conflict triple vuw. To make sure that edge modifications are not undone later in the search tree, we fix the outcome of each edge modification, that is, whenever we insert an edge, this edge is set to "permanent" and when we delete an edge, it is set to "forbidden". In the course of the search tree algorithm, we never delete permanent edges or insert forbidden edges. Furthermore, we also set certain edges to forbidden or permanent, if we know that the corresponding edge modification does not lead to an optimal solution.

In summary, our simple search tree algorithm first identifies a conflict triple vuw and branches into three cases to eliminate the conflict triple vuw:

1. Insert vw, set uv, uw, vw to permanent.

2. Delete uv, set uw to permanent and uv and vw to forbidden.

3. Delete uw, set uw to forbidden.

In each branch, we lower k by the corresponding modification cost. Afterwards, if $k > 0$ and the resulting graph is not a cluster graph, we call our algorithm recursively with the new value of k. If $k \leq 0$ and the resulting graph is not a cluster graph, we cannot find a solution in the corresponding branch, thus we stop this branch. If the resulting graph is a cluster graph, we report the solution and stop our algorithm. Figure 5.1 illustrates this straightforward branching strategy.

This algorithm generates a search tree of size $O(3^k)$. Since identifying a conflict triple takes $O(|V|^3)$ time, the running time of this algorithm is bounded by $O(3^k\,|V|^3)$. To find the optimal solution of the Weighted Cluster Editing Problem, we call the above algorithm at most k times, if k is the cost of the optimal solution. Thus, the optimal solution of the Weighted Cluster Editing Problem can be computed in time $O(3^k\,|V|^3)$.

In [18], we introduced a set of data reduction rules that can be exhaustively executed in time $O(|V|^3 \log |V|)$ and results in a problem kernel with at most $k^2 + 3k + 2$ vertices and $\frac{1}{2}k^3 + \frac{5}{2}k^2 + 2$ edges. See [18] for more details. Here, we would like to mention that the most efficient data reduction rule in [18] is the *merge operation*: As soon as

we set an edge $uv \in V$ to permanent, we replace vertices u, v by a new vertex, say u', and set $s(wu') := s(wu) + s(wv)$ for every $w \in V \setminus \{u, v\}$. Thus, merging two vertices forces certain edge modifications associated the merged vertices, which in turn cause some extra modification cost (called *induced cost* icp(uv) in [18]). For example, if one of the edges wu or wv exists in G and $s(wu') \leq 0$, merging u and v actually implies that the existing edge is deleted. So k is lowered by the corresponding induced costs.

Using interleaving technique (see Theorem 2) with the data reduction rules in [18] together with the simple depth-bounded search tree algorithm described above, we achieve the following theorem:

Theorem 7. Weighted Cluster Editing Problem *can be solved in time* $O(3^k + |V|^3 \log |V|)$*, where* $O(|V|^3 \log |V|)$ *is the running time of the data reduction introduced in [18].*

5.3.2 Refined Branching Strategy

In the following, we present a branching strategy with branching number 2.42 for the Weighted Cluster Editing Problem, that bases on the branching strategy with branching number 2.27 of Gramm *et al.* [70] for the Unweighted Cluster Editing Problem. To show why the branching strategy in [70] cannot be directly applied to the Weighted Cluster Editing Problem, we first recapitulate the branching strategy for Unweighted Cluster Editing Problem.

Let vuw be a conflict triple in G. The branching strategy of Gramm *et al.* [70] distinguishes the following cases:

(C1) Vertices v und w do not share a common neighbor except for u.

(C2) Vertices v und w have a common neighbor $x \neq u$ and $ux \in E$.

(C3) Vertices v und w have a common neighbor $x \neq u$ and $ux \notin E$.

If case (C2) or (C3) holds, the respective branching strategy shown in Figure 5.2 or 5.3 is applied to eliminate every conflict triple consisting of v, u, w and x. In each recursion call, k is lowered by the number of edge modifications executed. The branching vector of branching strategy for case (C2) is $(1, 2, 3, 2, 3)$ corresponding to branching number 2.27, and the branching vector of branching strategy for case (C3) is $(1, 2, 3, 3, 2)$ with branching number 2.27.

Obviously, Gramm *et al.*'s branching strategies for cases (C2) and (C3) also work on Weighted Cluster Editing Problem. We only have to lower k by the corresponding modification cost in each branch, instead of by the number of edge modifications. Since we require the modification cost of each vertex pair to be at least one, the above mentioned branching vectors and branching numbers also hold for Weighted Cluster Editing Problem.

For case (C1), where v and w have no common neighbor except for u, Gramm *et al.* showed in Lemma 5 of [70] that the minimum *number* of edge modifications does not increase if vw is set to forbidden. Thus, their algorithm only branches into two subcases:

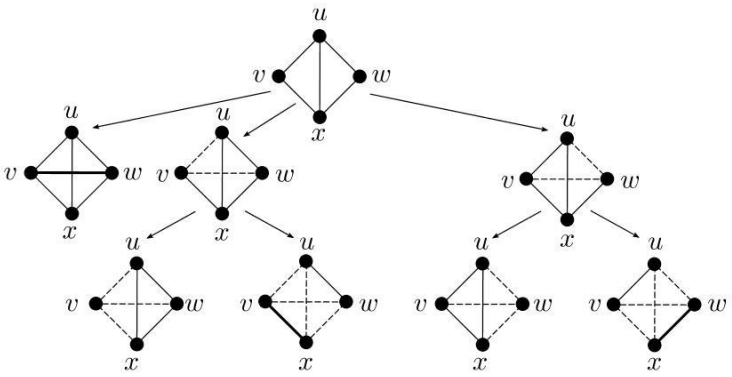


Figure 5.2: Branching strategy for case (C2): v und w have a common neighbor $x \neq u$ and $ux \in E$. Bold edges are permanent edges and dashed edges are forbidden edges. (Figure 2 in [70])

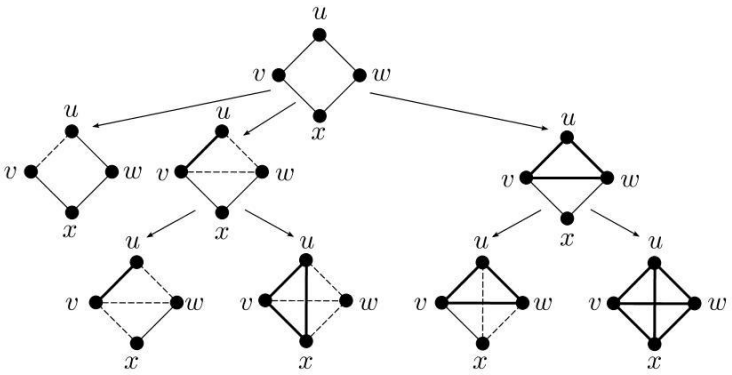


Figure 5.3: Branching strategy for case (C3): v und w have a common neighbor $x \neq u$ and $ux \notin E$. Bold edges are permanent edges and dashed edges are forbidden edges. (Figure 3 in [70])

deleting uv or deleting uw. This lead to a branching number 2 for case (C1) for the Unweighted Cluster Editing Problem. However, Lemma 5 of [70] does not hold for Weighted Cluster Editing Problem. As an example, we assume that the graph G has only three vertices v, u, w that form a conflict triple vuw with $s(uv) = s(uw) = 2$ and $s(vw) = -1$. The cost of the solution that inserts vw is one, whereas the cost of the solutions that delete uv or uw is two. Thus, we cannot use the branching strategies of Gramm *et al.* [70] for case (C1) on the Weighted Cluster Editing Problem.

In the following, we describe our own branching strategy for case (C1). First, we subdivide case (C1) into the following subcases:

(C1.1) Vertices v, w do not share common neighbors except for u, but there exits a vertex x, such that, say, $vx \in E$. We distinguish two subcases: (C1.1a) $ux \in E$ and (C1.1b) $ux \notin E$.

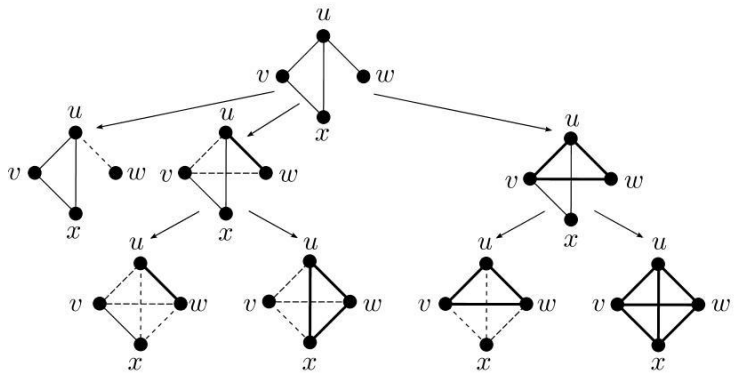


Figure 5.4: Branching strategy for case (C1.1a): v, w do not share common neighbor except for u and vx, $ux \in E$. Bold edges are permanent edges and dashed edges are forbidden edges.

(C1.2) Vertices v, w have no neighbor except for u, that is, $N(v) = N(w) = u$.

If (C1.1a) holds, we branch into the following cases to eliminate all conflict triples consisting of u, v, w, x:

1. Delete uw, set uw to forbidden.
2. Delete uv, ux, set uv, ux, vx, wx to forbidden and uw to permanent.
3. Delete uv, vx, insert wx, set uv, vw, vx to forbidden and uw, ux, wx to permanent.
4. Delete ux, vx, insert vw, set ux, vx, wx to forbidden and uv, uw, vw to permanent.
5. Insert vw, wx, set uv, ux, uw, vw, vx, wx to permanent.

Figure 5.4 illustrates our branching strategy for case (C1.1a). The branching vector of this branching strategy is $(1, 2, 3, 3, 2)$ corresponding to branching number 2.27.

Branching strategy for case (C1.1b) can be easily derived from Figure 5.5. This branching strategy has branching number 2.42.

If G is not a cluster graph, but none of the cases (C1a), (C1b), (C2) or (C3) holds for any conflict triple in G, the remaining case (C1.2) must hold for every conflict triple in G. In this case, every connected component of G that contains conflict triple, is a star graph, which is a tree where all vertices but one are leaves. In particular, the connected component of G that contains the conflict triple vuw is also a star graph. If $|N(u)| = 2$, we try all three possibilities to transform the connected component containing vuw into a cluster graph in constant time. In the following, we assume that $|N(u)| \geq 3$. Our branching strategy chooses three arbitrary neighbors of u, say v_1, v_2, v_3, and branches into the following subcases:

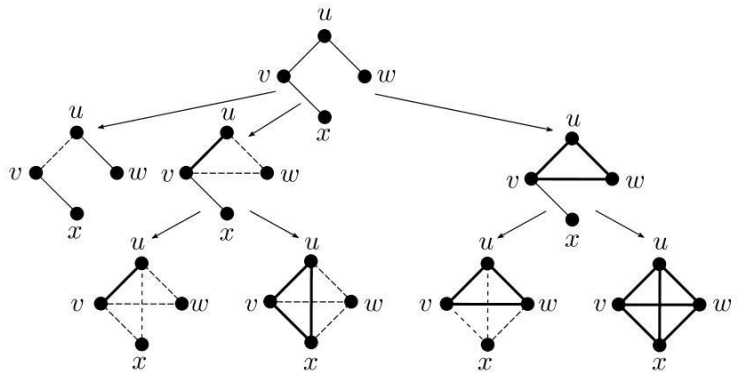


Figure 5.5: Branching strategy for case (C1.1b): v, w do not share common neighbor except for u and $ux \notin E$. Bold edges are permanent edges and dashed edges are forbidden edges.

1. Delete uv_1, set uv_1 to forbidden.
2. Delete uv_2, insert v_1v_3, set uv_2 to forbidden and uv_1, uv_3, v_1v_3 to permanent.
3. Delete uv_2, uv_3, set uv_2, uv_3 to forbidden and uv_1 to permanent.
4. Delete uv_3, insert v_1v_2, set uv_3 to forbidden and uv_1, uv_2, v_1v_2 to permanent.
5. Insert v_1v_2, v_1v_3, v_2v_3, set uv_1, uv_2, uv_3, v_1v_2, v_1v_3, v_2v_3 to permanent.

Figure 5.6 illustrates our branching strategy for case (C1.2). This branching strategy has branching vector $(1, 2, 2, 2, 3)$ and branching number 2.42.

All in all, given a weighted graph $G = (V, E)$, our algorithm identifies a conflict triple vuw. If case (C2) or (C3) holds, we apply the Gramm *et al.*'s branching strategy shown in Figure 5.2 or in Figure 5.3, respectively. If case (C1) holds, we further distinguish if it is case (C1.1a) or (C1.1b) and apply the corresponding branching strategies shown in Figure 5.4 and 5.5. If G is not a cluster graph and none of the above cases holds, every connected component of G is a star graph (case (C1.2)), we apply the branching strategy shown in Figure 5.6. The worst case branching number of our branching strategy is 2.42. Using interleaving technique with our kernelization in [18] together with the above branching strategy, we achieve the following theorem.

Theorem 8. *If the modification cost of each vertex pair of a graph $G = (V, E)$ is at least one, the running time of our algorithm with the refined branching strategy is $O(2.42^k + |V|^3 \log |V|)$.*

To evaluate the performance of our algorithms, we implemented the 3^k and the 2.42^k algorithm in C++ and evaluated their running time on artificial data, provided by Rahmann *et al.* [130], and biological data from the protein sequences dataset COG [148]. The computational results surprisingly indicate that our 3^k algorithm almost always outperforms our 2.42^k algorithm. The reason for this is the efficiency of the *merge operation*,

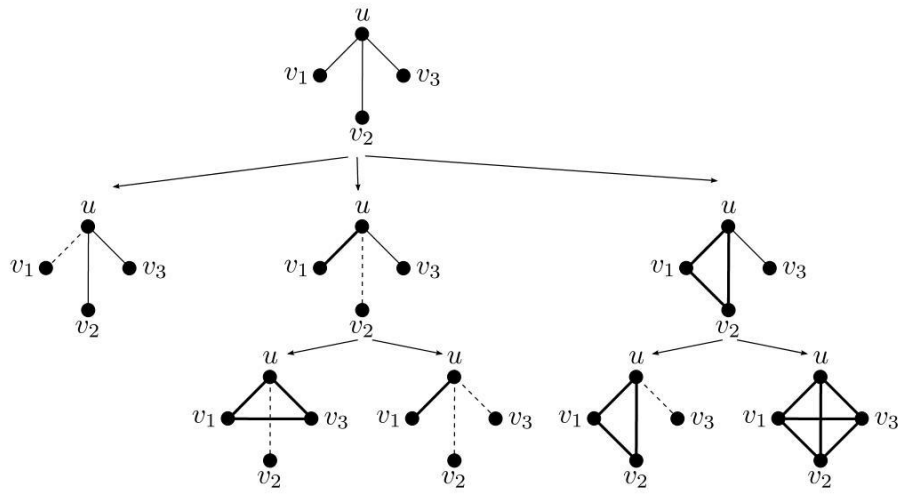


Figure 5.6: Branching strategy for case (C1.2): The conflict triple occurs in a star graph. Bold edges are permanent edges and dashed edges are forbidden edges.

which overrides advantages of the complicated branching strategy. See [16] for more details of our computational results.

5.4 Further Results

In the following, we subsume our further results on the Weighted Cluster Editing Problem published in Böcker *et al.* [18].

The main contribution of [18] consists of a novel branching strategy, called *edge branching*. This branching strategy is very simple: We just choose an arbitrary edge $uv \in E$ and branch into two subcases:

1. uv exists in a clique of the optimal solution, so we set uv to permanent, which implies that vertices u, v are merged;

2. or uv does not exists in the optimal solution, so we delete uv.

Deleting uv causes a modification cost of at least 1. As described in the previous part of this chapter, merging two vertices also forces certain edge modifications, which in turn causes modification costs (called $\mathrm{icp}(uv)$ in [18]). By an amortized running time analysis, we showed that the running time of our new algorithm is $O(2^k + |V|^3 \log |V|)$ on problem instances with positive integer cost. In [18], we showed that choosing a "good" edge to apply our edge-branching strategy results in a branching number of 1.82 and the running time of our algorithm on the basis of this branching strategy is bounded by $O(1.82^k + n^3)$. With this running time, our algorithm has the best running time for the Unweighted Cluster Editing Problem as well as the Weighted Cluster Editing Problem at time of preparing this work.

To evaluate the performance of our new algorithm, we again applied our algorithm to the artificial data provided by Rahmann *et al.* [130] and biological data from the protein sequences dataset COG [148]. Together with further data reduction rules of Böcker *et al.* [19], our 1.82^k algorithm was able to solve Weighted Cluster Editing Problem instances with several hundred of edge modifications in matter of seconds.

5.5 Summary and Outlook

In this chapter, we investigated the Weighted Cluster Editing Problem and gave a biological motivation for this problem. We then described our 2.42 branching strategy in detail. Furthermore, we mentioned our edge branching strategy, that does not only yield the best theoretical running for the Weighted Cluster Editing Problem and the Unweighted Cluster Editing Problem, but is also surprisingly simple and very efficient in our experiment. Interested readers are referred to the original articles [16–19] for more detail.

As the next step of our research regarding Weighted Cluster Editing Problem, we want to further improve the practical running time of our algorithm by applying more efficient algorithm engineering. We believe that biological instances still possess properties that are not utilized by the current implementation of our algorithm, for example, most of similarity measures are metric, but this property has not been considered in our algorithm.

From the theoretical point of view, we want to investigate closely related problems of the Weighted Cluster Editing Problem, for example, the Biclique Clustering Problem [75], which asks to transform a bipartite graph into a graph, where every connected component is a bi-clique, by a minimum number of edge modifications; or the Clique Cover Problem [71], which asks if there are at most k cliques in a graph, such that each edge of the graph belongs to at least one clique. In particular, we want to know if our merge operation can be adapted to those problems.

Similar to the situation of Flip Consensus Tree Problem and Flip Supertree Problem, there is a variant of the Unweighted Cluster Editing Problem, where it is *undecided* whether or not there is an edge between some vertex pair of the input graph, and the modification cost of those vertex pairs is zero. Whereas we recently proved that the Flip Supertree Problem is W[2]-hard [21], it is still a longstanding open question if this variant of the Unweighted Cluster Editing Problem is fixed-parameter tractable with respect to the number of edge modifications [29]. Since this problem is also biologically relevant, we want to develop an exact algorithm for this problem. However, we believe that an alternative approach, like integer linear programming with efficient data reduction rules, is more promising to solve this problem optimally in practice.

6 Bond Order Assignment

In this chapter, we investigate the Bond Order Assignment Problem that requires to assign bond orders to a given molecule graph minimizing a penalty score. In Section 6.3, we define this problem formally and show several complexity results in Section 6.4. In Section 6.5, we introduce a polynomial time algorithm for this problem when the molecule graph is a tree. Based on the tree decomposition of the molecule graph, we introduced two tree decomposition-based dynamic programming algorithm for this problem in Section 6.6.1 and 6.6.2. In Section 6.7, we discuss some heuristic improvements to speed up the practical running time of our algorithms. In Section 6.8, we report the computational results of one of our algorithms on a molecule database.

6.1 Motivation

The structural formula of a chemical compound is a representation of the molecular structure, illustrating the chemical bonds between pairs of atoms and sometimes also showing how atoms are arranged. Bond orders and atom valences are important information of a molecular structure. Knowledge of correct bond order is essential for many applications in chemistry and structural-oriented bioinformatics such as computing the molecular mechanics force field, querying structural-oriented databases, and accurate analysis of receptor-ligand interactions, which in turn play an important role in drug design [9, 51, 53, 83, 84, 142, 152]. Unfortunately, bond orders can be omitted in many data formats that represent molecular structure, such as Gaussian file formats and Mopac file formats, and even by the widely used Protein Data Bank format PDB. So, many entries in public databases omit bond order information, whereas other entries have erroneous such information. Moreover, in combinatorial chemistry, the backbone of a molecule (skeletal formula) may be drawn either manually or automatically, again omitting bond orders.

In the following, we consider the problem of (re-)assigning correct bond orders to molecular structures. We call this problem the Bond Order Assignment Problem.

6.2 Previous Work

In the last decades, many approaches have been introduced for the Bond Order Assignment Problem. Early approaches largely base on atomic coordinates. Meng and Lewis [116] used bond lengths and valence angles to derive the correct atom valence, but cannot determine bond orders. A similar approach of Baber and Hodgkin [9] can determine both atom valence as well as bond orders, but as they reported in [9], their

program errs often when dealing with conjugated ring systems. Some other geometry-based approaches [84,104,106,149,161] apply chemical function group pattern matching or use further molecular information like hybridization states and charges to achieve higher accuracy. However, the main drawback of geometry-based approaches is their strong dependence on the correctness of atomic coordinates.

Froeyen and Herdewijn [62] used an integer linear program to compute a valid Lewis structure minimizing the formal charge on each atom. In contrast to geometry-based approaches, this approach only needs bond connectivity and atom symbols as input to compute bond orders.

In [152], Wang *et al.* introduced a new approach for the Bond Order Assignment Problem, that also takes bond connectivity and atom symbols as input and computes a bond order assignment minimizing a score function that penalizes rare atom valences. The authors also provide a chemically motivated penalty score table for almost every common atom type in biomolecules. Furthermore, they introduce a heuristic algorithm to search for bond order assignment with minimum total penalty score.

Using the optimization criteria of Wang *et al.* [152], Dehof *et al.* [50] presented a branch-and-bound algorithm and an integer linear program to solve the Bond Order Assignment Problem. The authors also reported their computational results on the MMFF94 dataset [81] to confirm the practical use of their approaches.

In the following, we also investigate the Bond Order Assignment Problem using the optimization criteria of Wang *et al.* [152].

6.3 The Bond Order Assignment Problem

In this section, we formally define the Bond Order Assignment Problem using the optimization criteria of Wang *et al.* [152]. Let us start with the formal definition of a *molecule graph*.

Definition 6.1. A *molecule graph* $G = (V, E)$ is a graph, where each vertex $v \in V$ corresponds to an atom and each edge $e \in E$ corresponds to a chemical bond connecting the respective atoms. The vertices in G are labeled with the type of the corresponding atoms and the edges are labeled with the bond order of the corresponding bonds.

See Fig. 6.1 for the molecule graph of phenylalanine, an amino acid. Since each vertex corresponds to an atom, and each edge corresponds to a chemical bond, we use "atom" and "vertex", as well as "bond" and "edge" interchangeably in this chapter.

For each atom $v \in V$, let $\mathcal{A}_v$ denote the set of feasible valences of v. However, the real valence of an atom v in a specific molecule is determined by the sum of the bond orders of all bonds adjacent with that atom, and this valence is feasible if it is an element of $\mathcal{A}_v$. Wang *et al.* [152] define, for each atom $v \in V$, a penalty score function $\mathcal{S}_v : \mathcal{A}_v \to \mathbb{N}_0$ to penalize rare valences of v, and propose to assign each edge in E a bond order that minimizes the total penalty score over all atom in V. Thereby, the bond order of a bond can be a *single bond* or a *double bond* or a *triple bond*.

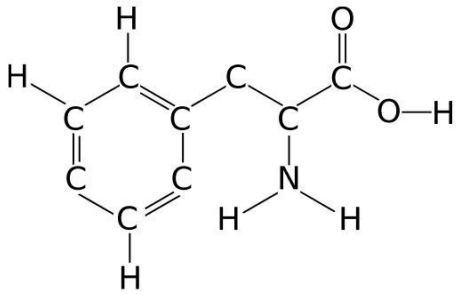


Figure 6.1: Molecule graph of phenylalanine.

Since the running time of our algorithms strongly depends on the maximum atom valence, we reduce the original valences of every atom in the input molecule graph by the number of its neighbors achieve the admissible *open valences*, that still can be consumed by the bonds adjacent to the atom.

Let $A_v := \{a - \deg(v) : a \in \mathcal{A}_v, a - \deg(v) \geq 0\}$ be the set of *open valences* of every $v \in V$. Let $A_v^* := \max A_v$ be the maximum open valence of an atom v and $\alpha := 1 + \max_v\{A_v^*\}$.

Relating to the open valence sets, we replace the score function $\mathcal{S}_v : \mathcal{A}_v \to \mathbb{N}_0$ with the *score function* $s_v : A_v \to \mathbb{N}_0$, where $s_v(a) := \mathcal{S}_v(a + \deg(v))$, for every $v \in V$.

Since using the open valences instead of the original valences already assigns every bond of G the minimum bond order of one, we define a *bond order assignment* as a function $b : E \to \{0, 1, 2\}$. For every bond $uv \in E$, setting $b(uv) = 0$ assigns uv a single bond; setting $b(uv) = 1$ assigns uv a double bond; setting $b(uv) = 2$ assigns uv a triple bond. Actually, $b(uv)$ can be considered as the number of additional chemical bonds added to the exiting bond at the edge uv.

Obviously, a bond order assignment b consumes $x_b(v) := \sum_{u \in N(v)} b(uv)$ open valences of every atom $v \in V$ and determines a valence state $x_b(v) + \deg(uv)$ for every atom $v \in V$. Moreover, a bond order assignment b is *feasible* if it holds for every $v \in V$ that $x_b(v) \in A_v$, i.e. $x_b(v) + \deg(v) \in \mathcal{A}_v$.

The *score* $S(b)$ of a bond order assignment b is defined as:

$$S(b) := \sum_{v \in V} s_v(x_b(v)).$$

To penalize infeasible bond order assignments, we set $s_v(a) := \infty$ for $a \notin A_v$, $v \in V$.

In terms of open valences, the Bond Order Assignment Problem according to Wang *et al.* is restated as follows:

Definition 6.2 (Bond Order Assignment Problem)**.** Given a molecule graph $G = (V, E)$ with open valence sets $A_v \subseteq \{0, \ldots, \alpha - 1\}$ and scoring functions s_v for every $v \in V$. Find a feasible assignment b for G with minimum score $S(b)$.

From now on, we solely work with "open valence", thus "open valence" is referred to as "valence" for simplicity.

6.4 Hardness of the Bond Order Assignment Problem

To investigate the computational complexity of the Bond Order Assignment Problem, we consider the Bond Order Assignment Decision Problem that is defined as follows.

Definition 6.3. Given an input for the Bond Order Assignment Problem, the corresponding Bond Order Assignment Decision Problem asks if there is a feasible assignment b for the input graph.

In the following, we first show that this problem is NP-hard, even on molecule graph with bounded vertex degree and maximum atom valence, and so is the Bond Order Assignment Problem.

Theorem 9. *The* Bond Order Assignment Decision Problem *is NP-complete, even on input graphs where every vertex has degree at most three and atom valences are at most four.*

From this theorem, we can quite easily infer that the Bond Order Assignment Problem cannot be approximated:

Lemma 6.1. *The* Bond Order Assignment Problem *cannot be approximated in polynomial time, unless $P = NP$, even on input graphs where every vertex has degree at most three and atom valences are at most four, and s_v is binary.*

The previous lemma might be regarded as an artifact, as asking for a solution of minimum score is somewhat arbitrary, and Wang *et al.* [152] could have formulated the Bond Order Assignment Problem as a maximization problem. This is similar to MAX-3SAT, where we do not minimize the number of unsatisfied clauses, but maximize the number of satisfied clauses. Consequently, we can use a positive score instead of a penalty score, and ask for a bond order assignment with *maximum* score. By the following theorem, finding an assignment with maximum score is a MAX SNP-hard problem, which implies the non-existence of a polynomial time approximation scheme (PTAS) unless P = NP [7].

Theorem 10. *Computing a bond order assignment with maximum score is a MAX SNP-hard problem, even on input graphs where every vertex has degree at most three and atom valences are at most four, and s_v is binary.*

We first focus on the NP-hardness of the decision problem. In the proof of Theorem 9, we will use a reduction from a variant of the 3-SAT problem:

Definition 6.4 (3-SAT)**.** Given a set X of n boolean variables $\{x_1, \ldots, x_n\}$ and a set C of m clauses $\{c_1, \ldots, c_m\}$. Each clause is a disjunction of at most three literals over X, for example $(x_1 \vee \overline{x_2} \vee x_3)$. Is there an assignment $X \to \{true, false\}$ that satisfies all clauses in C, i.e., at least one literal in every clause is *true*?

Definition 6.5 (3-SAT*)**.** The variant of 3-SAT where every variable occurs at most three times, is called the 3-SAT* problem.

In Theorem 3 of [13], Bermann *et al.* proved that 3-SAT* is NP-hard by a polynomial reduction to the NP-hard 3-SAT problem. In the following, we prove that the Bond Order Assignment Decision Problem is NP-complete by a polynomial reduction to 3-SAT*.

Proof of Theorem 9. Given a bond order assignment of a molecule graph, we can easily verify in polynomial time, if the bond order assignment is feasible. Thus, the problem is in NP. By a polynomial reduction from 3-SAT* to the Bond Order Assignment Decision Problem, we will show that the Bond Order Assignment Decision Problem is NP-hard, even if every vertex is of degree at most three and valence at most four.

Given a 3-SAT* formula, we can safely discard all clauses containing variables that only occur in either positive or negative literals. Afterwards, every variable occurs at least twice and at most three times in at least one positive and one negative literal. We then construct the SAT-graph $G = (V, E)$ as a Bond Order Assignment Decision Problem instance as follows:

The vertex set V consists of four subsets V_{var}, V_{lit}, V_{cla} and V_{aux}. For each variable x_i of the 3-SAT* instance, the vertex set V_{var} contains a *variable vertex* v_i and the vertex set V_{lit} contains two *literal vertices* u_i and u'_i corresponding to the literals x_i and $\overline{x_i}$. The set V_{cla} contains, for every clause c_j of the 3-SAT* instance, a *clause vertex* w_j. Finally, we need a couple of auxiliary vertices subsumed in V_{aux} as shown in Fig. 6.2.

The valence set of each variable vertex is $\{1\}$, of each literal vertex $\{0, 3\}$, and of a clause vertex $\{1, \ldots, d\}$, where $d \leq 3$ is the number of literals contained in the corresponding clause. The valence sets of auxiliary vertices are set as shown in Fig. 6.2. We use the trees shown in Fig. 6.2 as building blocks to connect the vertices of G.

If both literals of a variable occur once, we connect each of the literal vertices to the clause vertex that corresponds to the clause containing this literal via an auxiliary vertex with valence set $\{0, 3\}$, see Fig. 6.2 (left).

If one literal of a variable occurs once and the other twice, we connect the literal vertex that corresponds to the literal occurring in only one clause to the corresponding clause vertex via an auxiliary vertex with valence set $\{0, 3\}$. The literal vertex corresponding to the literal occurring in two clauses is connected to each of the corresponding clause vertices via a chain of three auxiliary vertices with valence sets $\{0, 3\}, \{0, 4\}, \{0, 3\}$. See Fig. 6.2 (right).

Before proving that the constructed Bond Order Assignment Decision Problem instance has a feasible assignment if and only if 3-SAT* instance is satisfiable, we consider the two building blocks of G shown in Fig. 6.2. Let a_1, a_2, b_1, b_2, c_1, c_2, c_3, d_1, d_2 denote the bond order of the corresponding edges as shown in Fig. 6.2. In a feasible assignment of G, the following facts can be easily observed:

As all variable vertices have valence set $\{1\}$, the bond orders a_1, b_1, c_1, and d_1 can either be zero or one. Bond order one can only be assigned to either a_1 or b_1, and to

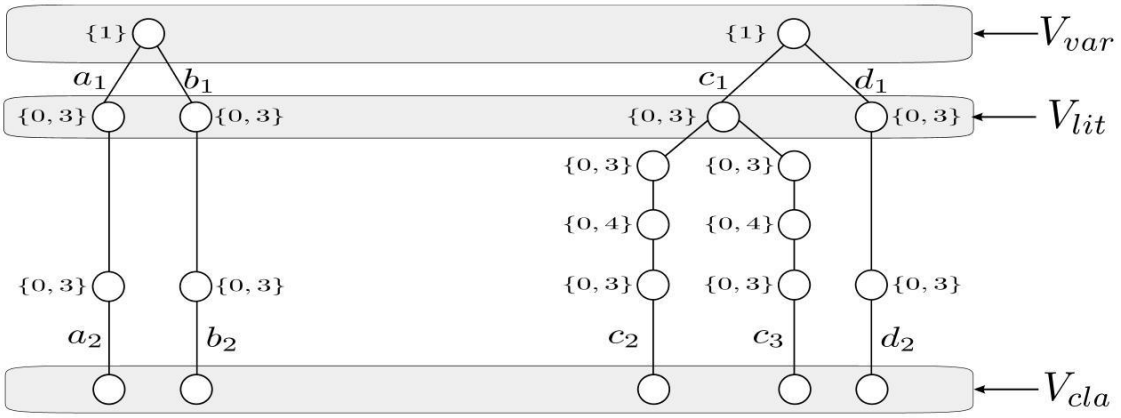


Figure 6.2: The building blocks of G. Vertices outside V_{var}, V_{lit}, and V_{cla} are auxiliary vertices. Valence sets of clause vertices are omitted.

either c_1 or d_1. The corresponding literal vertex has valence three, the other one has valence zero. Furthermore, we infer $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2 = c_3$ and $d_1 = d_2$. The fact that exactly one of two edges incident to a variable vertex has bond order one, models that exactly one of the literals $x_i, \overline{x_i}$ of a variable x_i is satisfied. The valence of a clause vertex takes a value of at least one if and only if the corresponding clause contains literals whose literal vertices have valence three. This implies that a clause is satisfied if and only if it contains a *true* literal. Furthermore, the valence set $\{1, \ldots, d(w)\}$ of a clause vertex w forces any algorithm for the Bond Order Assignment Decision Problem to assign bond order one to at least one of the edges incident to w. This implies that at least one of the literals contained in each clause has to be *true*.

Therefore, there is a feasible solution for the constructed Bond Order Assignment Decision Problem instance if and only if the 3-SAT* instance is satisfiable. Since the reduction can be done in polynomial time and the 3-SAT* problem is NP-hard, the Bond Order Assignment Decision Problem is also NP-hard. □

Since the Bond Order Assignment Decision Problem is NP-hard, so is the Bond Order Assignment Problem. Next, we prove that the Bond Order Assignment Problem is even not approximable in polynomial time, unless P = NP.

Proof of Lemma 6.1. We modify the reduction described in the proof of Theorem 9 by allowing clause vertices to take valence zero, so that the valence set of a clause vertex is $\{0, 1, \ldots, d\}$ where $d \leq 3$ is the length of the clause. Valence zero at each clause vertex is penalized with score one, whereas the scores of all other valences of all vertices are set to zero. Doing so, we ensure that the score of an assignment is the number of clause vertices with valence zero and, hence, the number of unsatisfied clauses in the 3-SAT* problem instance. Thus, the 3-SAT* problem instance is satisfiable if and only if there is a bond order assignment for the SAT-graph with score zero.

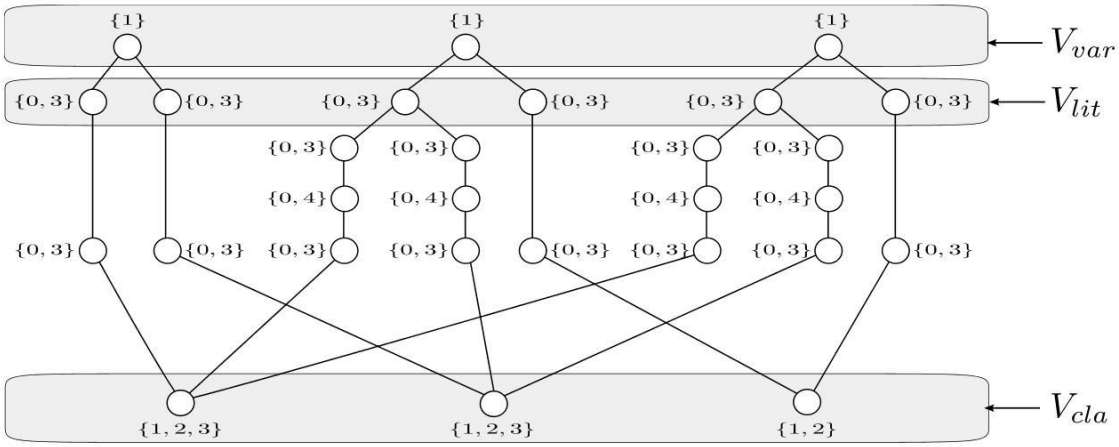


Figure 6.3: The variable vertices represents variables x_1, x_2, x_3 from left to right. The literal vertices represent literals $x_1, \overline{x_1}, x_2, \overline{x_2}, x_3, \overline{x_3}$, from left to right. The clause vertices represent clauses $(x_1 \vee x_2 \vee x_3)$, $(\overline{x_1} \vee x_2 \vee x_3)$, $(\overline{x_2} \vee \overline{x_3})$, from left to right.

Assume that there is a polynomial-time approximation algorithm for the Bond Order Assignment Problem with an arbitrary (but finite) approximation factor. This algorithm computes a bond order assignment with score zero for a Bond Order Assignment Problem instance if and only if an optimal assignment has score zero. In particular, for Bond Order Assignment Problem instances constructed from a 3-SAT* problem instance, this polynomial-time approximation algorithm computes an assignment with score zero if and only if the 3-SAT* problem instance is satisfiable. So, we can use this approximation algorithm to solve the NP-hard 3-SAT* problem in polynomial time. Thus, there is no polynomial-time approximation algorithm for the Bond Order Assignment Problem, unless P = NP. □

Finally, we focus on the MAX SNP-hardness of the maximization version of the Bond Order Assignment Problem. The MAX SNP-hardness concept was introduced by Papadimitriou *et al.* [126], who also defined the L-reduction to show MAX SNP-hardness of an optimization problem. The L-reduction is defined as follows:

Definition 6.6 (L-reduction)**.** Let Π and Π' be two optimization (maximization or minimization) problems. We say that Π L-reduces to Π' if there are two polynomial-time algorithm f, g and constants $\delta, \gamma > 0$ such that for each instance I of Π:

1. Algorithm f produces an instance $I' = f(I)$ of Π', such that the optima of I and I', $OPT(I)$ and $OPT(I')$, respectively, satisfy $OPT(I') \leq \delta OPT(I)$.

2. Given any solution of I' with cost c', algorithm g produces a solution of I with cost c such that $|c - OPT(I)| \leq \gamma |c' - OPT(I')|$.

To prove MAX SNP-hardness of computing a bond order assignment with maximum score, we introduce an L-reduction from the MAX-3SAT* problem, which is a MAX SNP-hard problem [126]. Our proof is straightforward, as we will set $\delta = \gamma = 1$.

Definition 6.7 (MAX-3SAT*)**.** Given a set of length three clauses over a set of boolean variables, where each variable occurs at most three times in the clause set, the MAX-3SAT* problem asks for an assignment of the variables that satisfies as many clauses as possible.

Proof of Theorem 10. We reduce MAX-3SAT* to the maximization version of Bond Order Assignment Problem via an L-reduction. Given a MAX-3SAT* problem instance, we construct a SAT-graph as described in the proof of Theorem 9 and use the valence sets and scoring functions defined in the proof of Lemma 6.1, but we swap the scoring functions at clause vertices, namely we assign score zero to valence zero and score one to non-zero valences. By doing this we can ensure that there is always a feasible solution for the constructed Bond Order Assignment Problem instance, and the score of a bond order assignment is the number of clause vertices that have non-zero valences.

Since a clause vertex has non-zero valence if and only if the corresponding clause is satisfied, from a solution of the MAX-3SAT* problem instance that satisfied k clauses we can construct a solution of the Bond Order Assignment Problem problem instance with score k, and vice versa. This can be done in polynomial time as described in the proof of Theorem 9. In the following, we use k to denote the number of satisfied clauses of the MAX-3SAT* problem instance, as well as the score of the corresponding Bond Order Assignment Problem instance.

Let $OPT(BOA)$ denote the score of the optimal solution of the constructed Bond Order Assignment Problem problem instance, and $OPT(M3S)$ denote the number of satisfied clauses in the optimal solution of the MAX-3SAT* problem instance. We infer that $OPT(BOA) = OPT(M3S)$.

To prove that our reduction from MAX-3SAT* problem to the maximization version of the Bond Order Assignment Problem is an L-reduction, we have to show that $OPT(BOA) \leq \delta \cdot OPT(M3S)$ and $|k - OPT(M3S)| \leq \gamma \cdot |k - OPT(BOA)|$ hold for some constant δ and γ. Since $OPT(BOA) = OPT(M3S)$, both conditions hold for $\delta = \gamma = 1$.

All in all, our reduction from MAX-3SAT* to the maximization version of the Bond Order Assignment Problem is an L-reduction. Since MAX-3SAT* is MAX SNP-hard, computing a bond order assignment with maximum score is also MAX SNP-hard, even on input graphs where every vertex has degree at most three and atom valences are at most four, and s_v is binary. □

6.5 Polynomial Algorithm on Trees

In this section, we introduce a polynomial time algorithm for a special case of the Bond Order Assignment Problem, where the input graph is a tree. Although this

case can be solved in polynomial time by the more sophisticated algorithms introduced Section 6.6.1 and 6.6.2, the algorithm introduced in this section is easier to understand and provides the basic idea of our tree decomposition-based algorithms in Section 6.6.1 and 6.6.2.

Given a tree T as input graph, we root T at an arbitrary node r and set the direction of every edge to point away from r. Let the *ordered* pair uv denote the edge connecting u and v, where v is a child of u.

In the following, we assume that T is a binary tree. Actually, our algorithm can be extended to solve the Bond Order Assignment Problem on a non-binary tree, but the main focus of this section is to provide the basic idea of our tree decomposition-based algorithm in Section 6.6.1 and 6.6.2, we omit the technical details of the extension of our algorithm on non-binary tree.

With the assumption that T is a binary tree, we introduce a polynomial time algorithm that solves Bond Order Assignment Problem bottom-up. Let uv be an edge in T and w_1, w_2 be the two children of v. Let $D_v[a_v, b(uv)]$ denote the score of the optimal solution for the subtree rooted at v under the condition that valence a_v is assigned to v and bond order $b(uv)$ is assigned to edge uv. If all values $D_{w_1}[\cdot,\cdot]$ and $D_{w_2}[\cdot,\cdot]$ are known for both children w_1, w_2 of a node v, then we can compute $D_v[\cdot,\cdot]$ using the recurrence

$$D_v[a_v, b(uv)] = s_v(a_v) + \min_{\substack{b_1+b_2 \\ +b(uv)=a_v}} \left\{ D_{w_1}[a_1, b_1] + D_{w_2}[a_2, b_2] \right\}$$

where the minimum is taken over all $a_1 \in A_{w_1}, a_2 \in A_{w_2}$ and all $b_1, b_2 \in \{0, 1, 2\}$. Note that for inner nodes with only one child, we can simplify the recurrence $D_v[a_v, b(uv)]$ by setting the missing summand to zero.

At every leaf w with parent v, we initialize the recurrence $D_w[a_w, b(vw)]$ as following:

$$D_w[a_w, b(vw)] = \begin{cases} s_w(a_w) & \text{if } a_w = b(vw), \\ \infty & \text{otherwise.} \end{cases}$$

After the bottom-up traversal, the value $\min_{a_r \in A_r} D_r[a_r, 0]$ is returned as the score of the optimal bond order assignment for T.

The correctness of our algorithm at the leaves of the input tree is obvious. Since each inner node v has two out-going edges and one in-coming edge, given a fixed valence of the atom at an inner node and a fixed bond order of the in-coming edge, it is obvious that the above recurrence computes the bond orders of the out-going edges such that bond order assignment of the subtree rooted at that inner node is optimal. Since the root of T has no in-coming edge, it holds that $\min_{a_r \in A_r} D_r[a_r, 0]$ is the score of the optimal bond order assignment for T.

To the running time of the algorithm, it is obvious that the size of each score matrix D_v is bounded by $O(|A_v|)$ and computing each entry of a matrix takes constant time (since there are only three possible bond orders for each bond). Since our algorithm visits each vertex of the tree only once, the running time of our algorithm is bounded by $O(max_{v \in V} |A_v| \cdot n)$, where n is the number of vertices.

6.6 Algorithms for Molecule Graphs of Bounded Treewidth

In the previous section, we proved that the Bond Order Assignment Problem can be solved in polynomial time if the input graph is a tree. Therefore, it is natural to extend this algorithm to molecule graphs that are "tree-like". Following this idea, we apply the tree decomposition-based dynamic programming approach (see Section 2.3.2) to the Bond Order Assignment Problem and achieve two fixed-parameter algorithms with respect to the treewidth of the molecule graph and the maximum atom valence.

Assume that an optimal tree decomposition of the molecule graph G is given. Let $\omega-1$ be the width and m be the number of bags of the optimal tree decomposition of G. This tree decomposition is then transformed into a nice tree decomposition $\langle\{X_i \mid i \in I\}, T\rangle$ of the same width with $O(m)$ bags. Above the root of T, we add additional forget nodes, such that the new root contains a single vertex. Let X_r denote the new root of the tree decomposition and v_r denote the single vertex contained in X_r. Analogously, we add additional introduce nodes below each leaf of T, such that the new leaf also contains a single vertex.

Without loss of generality, let $v_1, v_2, \ldots, v_k$ denote the atoms inside a bag X_i, where $k \leq \omega$. For simplicity of presentation, we assume that all bonds $v_1v_2, v_1v_3, \ldots, v_{k-1}v_k$ are present in each bag. Furthermore, let Y_i denote the atoms in G that are contained in the bags of the subtree rooted at bag X_i.

In our algorithms, we assign each bag X_i a score matrix D_i. In Section 6.6.1, we will specify the contents of those score matrices and describe a tree decomposition-based dynamic programming algorithm for the Bond Order Assignment Problem with running time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$, where m is the number of nodes in the tree decomposition of the molecule graph, $\alpha-1$ is the maximum *open valence* of an atom, d is the maximum degree of an atom in the molecule, $\omega - 1$ is the treewidth of the molecule graph, and $\beta := \min\{\binom{\omega}{2}, \omega\, d\}$.

In Section 6.6.2, we introduce an improved version of the algorithm in Section 6.6.1. The running time of this improved algorithm is bounded by $O(\alpha^{3\omega} \cdot \omega \cdot m)$.

6.6.1 The $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$ Algorithm

In the following, we describe the algorithm with running time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$ for the Bond Order Assignment Problem.

In the following, let $D_i[a_1, \ldots, a_k;\, e_{1,2}, \ldots, e_{k-1,k}]$ be the minimum score over all valence assignments to the atoms in $Y_i \setminus X_i$ if for every $l = 1, \ldots, k$, exactly a_l valences of atom v_l have been consumed by the bonds between v_l and atoms in $Y_i \setminus X_i$, and bond orders $e_{1,2}, \ldots, e_{k-1,k}$ are assigned to bonds $v_1v_2, v_1v_3, \ldots, v_{k-1}v_k$.

Using this definition, we delay the scoring of any atom to the forget node where it is removed from a bag. This is advantageous since every atom, except for the atom v_r at the root of the tree decomposition, is forgotten exactly once, and since the exact valence of a vertex is not known until it is forgotten in the tree decomposition. Finally, we can compute the minimum score among all assignments using the root bag $X_r = \{v_r\}$ as $\min_{a_1}\{s_{v_r}(a_1) + D_r[a_1]\}$.

Our algorithm begins at the leaves of the tree decomposition and computes the score matrix D_i for every node X_i when score matrices of its children nodes have been computed. We initialize the matrix D_j of each leaf $X_j = \{v_1\}$ with

$$D_j[a_1; \cdot] = \begin{cases} 0 & \text{if } a_1 = 0, \\ \infty & \text{otherwise.} \end{cases}$$

During the bottom-up travel, the algorithm distinguishes if X_i is a forget node, an introduce node, or a join node, and computes D_i as follows:

Introduce nodes. Let X_i be the parent node of X_j such that $X_j = \{v_1, \ldots, v_{k-1}\}$ and $X_i = \{v_1, \ldots, v_k\}$. Then,

$$D_i[a_1, \ldots, a_k; e_{1,2}, \ldots, e_{k-1,k}] = \begin{cases} D_j[a_1, \ldots, a_{k-1};\, e_{1,2}, \ldots, e_{k-2,k-1}] & \text{if } a_k = 0, \\ \infty & \text{otherwise.} \end{cases}$$

Forget nodes. Let X_i be the parent node of X_j such that $X_j = \{v_1, \ldots, v_k\}$ and $X_i = \{v_1, \ldots, v_{k-1}\}$. Then,

$$D_i[a_1, \ldots, a_{k-1};\, e_{1,2}, \ldots, e_{k-2,k-1}] = \min_{\substack{e_{1,k},\ldots,e_{k-1,k} \in \{0,1,2\} \\ a_k \in \{0,\ldots,A^*_{v_k}\}}} \Bigg\{ s_{v_k}\Big(a_k + \sum\nolimits_{l=1}^{k-1} e_{l,k}\Big) + D_j[a_1 - e_{1,k}, \ldots, a_{k-1} - e_{k-1,k}, a_k;\, e_{1,2}, \ldots, e_{k-1,k}] \Bigg\} \quad (6.1)$$

Join nodes. Let X_i be the parent node of X_j and X_h such that $X_i = X_j = X_h$. Then,

$$D_i[a_1, \ldots, a_k; e_{1,2}, \ldots, e_{k-1,k}] = \min_{\substack{a'_l = 0,\ldots,a_l \\ \text{for } l=1,\ldots,k}} \Big\{ D_j[a'_1, \ldots, a'_k;\, e_{1,2}, \ldots, e_{k-1,k}] + D_h[a_1 - a'_1, \ldots, a_k - a'_k;\, e_{1,2}, \ldots, e_{k-1,k}] \Big\} \quad (6.2)$$

For simplicity of the presentation of our algorithm, we assumed above that every two atoms in each bag of the tree decomposition are connected by a bond, but in reality, the number of bond in a bag is upper-bounded by ωd, where d is the maximum degree of an atom in the molecule graph.

Lemma 6.2. *Given a nice tree decomposition of a molecule graph G, the algorithm described above computes an optimal assignment for the* Bond Order Assignment Problem *on G in time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$, where $\alpha = 1 + \max_v A^*_v$ is the maximum (open) valence of an atom plus one, m and $\omega - 1$ are size and width of the tree decomposition, d is the maximum degree in the molecule graph, and $\beta := \min\{\binom{\omega}{2}, \omega\, d\}$.*

Proof. We first analyse the running time of the algorithm. Since the maximum valences of an atom is $\alpha - 1$, there are at most α^ω possibilities to assign consumed valences to ω atoms in each bag of T. According to the definition of β, the number of bonds in each bag of T is bounded by β, and there are at most 3^β possibilities to assign bond orders $b \in \{0, 1, 2\}$ to β bonds. This implies that the table D_i of a node X_i contains at most $\alpha^\omega \cdot 3^\beta$ entries.

We now consider the running time of computing each entry in the matrix D_i, assuming that the corresponding matrix of every child of X_i has already been calculated. We distinguish the following cases:

If X_i is a leaf of T, computing matrices D_i takes constant time, since entries with score infinity do not have to be considered, and there is only one entry with score zero. If X_i is an introduce node, calculating each entry of D_i takes constant time. Again, we do not have to consider entries with score infinity. Thus, computing the table of an introduce node take time $O(\alpha^\omega 3^\beta)$.

Assuming that X_i is a forget node and X_j is the child node of X_i. We compute the score matrix D_i on-the-fly when computing D_j: For each fixed value of $a_1, \ldots, a_{k-1}$, after computing each $D_j[a_1 - e_{1,k}, \ldots, a_{k-1} - e_{k-1,k}, a_k;\ e_{1,2}, \ldots, e_{k-1,k}]$, we also compute the minimum score $s_{v_k}(a_k + \sum_{l=1}^{k} e_{l,k}) + D_j[a_1 - e_{1,k}, \ldots, a_{k-1} - e_{k-1,k}, a_k;\ e_{1,2}, \ldots, e_{k-1,k}]$ over all possible values of a_k and $e_{1,k}, \ldots e_{k-1,k}$. This minimum score is the score needed to be computed and stored in $D_i[a_1, \ldots, a_{k-1};\ e_{1,2}, \ldots, e_{k-2,k-1}]$. By doing this, we can compute each entry of a forget node in constant time. Moreover, this only increases the running time at the child node of a forget node by a constant factor. Thus, score matrix at a forget node can be computed in time $O(\alpha^\omega 3^\beta)$.

Let X_i be a join node and X_j and X_h its children. To calculate an entry of D_i, the algorithm has to find a "partner entry" in D_h for every entry of D_j, such that the number of consumed valences of an atom in X_i is the sum of the consumed valences of the same atoms in X_j and X_h, and calculate the minimum score of all such pairs, as shown in Equation (6.2). Therefore, computing an entry of the matrix D_i can be done in time $O(\alpha^\omega \cdot \omega)$.

Now, the tree decomposition contains $O(m)$ nodes, the matrix of every node contains at most $\alpha^\omega \cdot 3^\beta$ entries. Computing each matrix entry takes time $O(\alpha^\omega \cdot \omega)$. Moreover, initializing the matrices of all leaves of the tree decomposition takes $O(m)$ time. In total, the running time of the algorithm is $O(\alpha^{2\omega} \cdot 3^\beta \cdot \omega \cdot m)$.

In the following, we prove the correctness of the algorithm. As additional forget nodes above the root of the tree decomposition are introduced until the new root contains a single atom, and every atom except the atom in the new root has to be "forgotten" once. Note that because of the third property of tree decompositions, an atom cannot be forgotten more than once. Whenever an atom is "forgotten", it gets the valence state that equals the sum of valences it used up in the subtree below the child of the forget node, plus the bond orders of its adjacent bonds inside the child node. Our algorithm chooses a valence state for the forgotten atom that minimizes the sum between scores of the chosen valence state and the corresponding entry in D_j, as described in Equation (6.1). This confirms that our algorithm correctly computes score matrices at forget nodes.

Considering every leaf of T as an introduce node, we can see that a newly introduced atom has not used up any of its valences yet. Therefore, it is correct to set the score of entries, where the newly introduced atom already uses valences, to infinity. The correctness of the algorithm at introduce nodes is also obvious.

An atom can be introduced more than once, but because of the third property of tree decompositions, there must be one join node that joins all occurrences of an atom. At each join node, only two occurrences of an atom are joined. Let v be an atom of the molecule graph that is introduced twice. On the two paths from the corresponding introduce nodes to the join node, where two occurrences of v are joined, each occurrence of v may consume different amounts of valences. Note that this can only happen if different atoms are forgotten on the two paths. Therefore, the total amount of consumed valences of v in the subtree below the join node is the sum of consumed valences in the subtrees below the children of the join node. Since we are interested only in the optimal solution, we only take the minimum score as shown in Equation (6.2). Thus, our algorithm computes score matrices at join node correctly.

When the algorithm arrives at the root r of the tree decomposition, it holds that every atom of the molecule graph has been considered, and the scores that correspond to the valence state assigned to each atom have been summed up in the corresponding entry in D_r. Except for the only atom v_r in the root r, every atom has been forgotten on some path in the tree, and it gets a feasible valence state. This means that the validity of the assignment for the subgraph $G \setminus v_r$ of G is assured. Furthermore, because of the definition of score matrix D and the correctness of our algorithm when computing score matrix, $\min_a s_{v_r}(a) + D_r[a]$ is the score of an optimal solution for the Bond Order Assignment Problem on G.

All in all, our algorithm computes an optimal bond order assignment of G using a nice tree decomposition of G in time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$. □

During the bottom-up processing, we can store where the minimum is obtained in Equation (6.1) and (6.2), afterwards we traverse the tree decomposition top-down to obtain all bond orders of the molecule. This can be done in time $O(m)$.

6.6.2 The $O(\alpha^{3\omega} \cdot \omega \cdot m)$ Algorithm

The idea for this version of the algorithm bases on the observation that the information about the bond order assigned to each bond in a bag of the tree decomposition is not really necessary, but the number of valences of an atom used up by bonds within a bag of the tree decomposition is more important. To make use of this observation, we modify our algorithm described in Section 6.6.1 as follows:

Let $v_1, \ldots, v_k$ denote the atoms in a bag X_i, and $D_i[a_1, \ldots, a_k; b_1, \ldots, b_k]$ be the minimum score over all valence assignments to the atoms in $Y_i \setminus X_i$ if, for every $l = 1, \ldots, k$, exactly a_l valences of atom v_l have been consumed by bonds between v_l and atoms in $Y_i \setminus X_i$, and b_l valences of atom v_l are consumed by bonds within the bag X_i. Recall that Y_i is the set of atoms occurring in the subtree rooted at X_i. Again, our algorithm starts at the leaves of the tree decomposition and computes the score matrix D_i for every

node X_i when score matrices of all its child nodes have been computed. The score of the optimal bond order assignment is $\min_{a_1}\{s_{v_r}(a_1) + D_r[a_1]\}$, where v_r is the only atom in the root bag X_r of the tree decomposition.

We initialize the matrix D_j of each leaf $X_j = \{v_1\}$ with

$$D_j[a_1; b_1] = \begin{cases} 0 & \text{if } a_1 = b_1 = 0, \\ \infty & \text{otherwise.} \end{cases}$$

We distinguish if a bag X_i is an introduce node, a forget node, or a join node and use the corresponding recurrence to calculate D_i:

Introduce nodes. Let X_i be the parent node of X_j such that $X_j = \{v_1, \ldots, v_{k-1}\}$ and $X_i = \{v_1, \ldots, v_k\}$. Then,

$$\begin{aligned} &D_i[a_1, \ldots, a_k;\, b_1, \ldots, b_k] \\ &\quad = \min_{\substack{e_1,\ldots,e_{k-1}\in\{0,1,2\} \\ \sum_l e_l = b_k}} \begin{cases} D_j[a_1, \ldots, a_{k-1};\, b_1 - e_1, \ldots, b_{k-1} - e_{k-1}] & \text{if } a_k = 0, \\ \infty & \text{otherwise.} \end{cases} \end{aligned} \tag{6.3}$$

State that $e_1, \ldots, e_{k-1}$ are in fact the bond orders of edges adjacent to the newly introduced atom a_k

Forget nodes. Let X_i be the parent node of X_j such that $X_j = \{v_1, \ldots, v_k\}$ and $X_i = \{v_1, \ldots, v_{k-1}\}$. Then,

$$\begin{aligned} D_i[a_1, \ldots, a_{k-1};\, b_1, \ldots, b_{k-1}] = \min\Big\{ & s_{v_k}(a_k + b_k) \\ & + D_j[a_1 - e_1, \ldots, a_{k-1} - e_{k-1}, a_k;\, b_1 + e_1, \ldots, b_{k-1} + e_{k-1}, b_k] \Big\} \end{aligned} \tag{6.4}$$

where the minimum runs over all $e_1, \ldots, e_{k-1} \in \{0, 1, 2\}$ such that $\sum_{l=1}^{k-1} e_l = b_k$, and all $a_k = 0, \ldots, A^*_{v_k}$. In this case, $e_1, \ldots, e_{k-1}$ denote the bond orders of edges incident to the forgotten atom a_k.

Join nodes. Let X_i be the parent node of X_j and X_h such that $X_i = X_j = X_h$. Then,

$$\begin{aligned} &D_i[a_1, \ldots, a_k;\, b_1, \ldots, b_k] = \\ &\quad \min_{\substack{a'_l = 0,\ldots,a_l \\ \text{for } l=1,\ldots,k}} \Big\{ D_j[a'_1, \ldots, a'_k;\, b_1, \ldots, b_k] + D_h[a_1 - a'_1, \ldots, a_k - a'_k;\, b_1, \ldots, b_k] \Big\}. \end{aligned} \tag{6.5}$$

Lemma 6.3. *Given a nice tree decomposition of a molecule graph G, the algorithm described above computes an optimal assignment for the* Bond Order Assignment Problem *on G in time $O(\alpha^{3\omega} \cdot \omega \cdot m)$, where $\alpha = 1 + \max_v A^*_v$ is the maximum (open) valence of an atom plus one, and m and $\omega - 1$ are size and width of the tree decomposition.*

Proof. We first analyze the running time of our algorithm. Since each bag that is a leaf of the tree decomposition contains only one atom, initializing the table of a leaf takes constant time, and initializing all leaves of the tree decomposition takes time $O(m)$. We now consider the running time of the algorithm at each inner node X_i of the tree decomposition. Obviously, the table D_i of each bag X_i contains at most $\alpha^{2\omega}$ entries. Let X_j denote the child node of X_i, if X_i is an introduce node or a forget node, and X_j and X_h denote the children node of X_i if X_i is a join node.

At an introduce node X_i, let v_k be the newly introduced atom and $e_l \in \{0,1,2\}$ denote the bond order of a bond $v_l v_k$. To calculate an entry $D_i[a_1,\ldots,a_k;\, b_1,\ldots,b_k]$, the algorithm has to consider every entry $D_j[a_1,\ldots,a_{k-1};\, b'_1,\ldots,b'_{k-1}]$ with fixed indices $a_1,\ldots,a_{k-1}$ and $b'_l = b_l - e_l$, for $1 \leq l \leq k-1$ and $\sum_{1\leq l\leq k-1} e_l = b_k$. Since there are at most $\alpha^{\omega-1}$ such entries in D_j and testing if $b'_l = b_l - e_l$, for $1 \leq l \leq k-1$, takes time $O(\omega)$, calculating an entry of D_i takes time $O(\alpha^\omega \cdot \omega)$. Therefore, the table of an introduce node can be calculated in time $O(\alpha^{3\omega} \cdot \omega)$.

Let X_i be a forget node where atom v_k is forgotten. Again, let $e_l \in \{0,1,2\}$ denote the bond order of a bond $v_l v_k$. To analyze the running time of the algorithm at a forget node, we describe the execution of our algorithm at a forget node in detail. To calculate an entry $D_i[a_1,\ldots,a_{k-1};\, b_1,\ldots,b_{k-1}]$, the algorithm tests for all possible valences $a'_1,\ldots,a'_{k-1},a'_k$ if $e_l = a_l - a'_l \in \{0,1,2\}$ holds, for $1 \leq l \leq k-1$. If this is true, the algorithm sets $b_k := \sum_{1\leq l\leq k-1} e_l$ and computes the score $s_{v_k}(a'_k + b_k) + D_j[a'_1,\ldots,a'_{k-1},a'_k;\, b_1+e_1,\ldots,b_{k-1}+e_{k-1},b_k]$. This can be done in time $O(\omega)$. The minimum score over all such scores is assigned to $D_i[a_1,\ldots,a_{k-1};\, b_1,\ldots,b_{k-1}]$. Since there are at most α^ω possibilities of indices $a'_1,\ldots,a'_{k-1},a'_k$, calculating an entry of a forget node can be done in time $O(\alpha^\omega \cdot \omega)$. Therefore, the running time of our algorithm at a forget node is bounded by $O(\alpha^{3\omega} \cdot \omega)$.

When calculating an entry $D_i[a_1,\ldots,a_k;\, b_1,\ldots,b_k]$ of a join node, the algorithm has to test for each entry $D_j[a'_1,\ldots,a'_k;\, b_1,\ldots,b_k]$ in D_j, if this entry and its partner $D_h[a_1-a'_1,\ldots,a_k-a'_k;\, b_1,\ldots,b_k]$ in D_h minimize score at $D_i[a_1,\ldots,a_k;\, b_1,\ldots,b_k]$. Since there are at most α^ω entries in D_j with fixed indices $b_1,\ldots,b_k$, calculating an entry of the table of a join node takes $O(\alpha^\omega \cdot \omega)$, and thus the running time for calculating the table of a join node is bounded by $O(\alpha^{3\omega} \cdot \omega)$. In total, since the tree decomposition contains m nodes, the running time of our modified algorithm is bounded by $O(\alpha^{3\omega} \cdot \omega \cdot m)$.

Next, we prove the correctness of our algorithm. The initialization at the leaves of the tree decomposition is obviously correct, since no valence of any atom is used up at this stage. At an introduce node, no valence of the newly introduced atom v_k is used up in the subtree rooted at this introduce node. Furthermore, each bond order e_l between v_k and an atom v_l in the introduce node increases the number of consumed valences of v_l from $b_l - e_l$ to b_l, and the bond orders of all edges between v_k and other vertices in this node sum up to b_k valences of v_k, that are consumed within this node. This intuition confirms the correctness of our algorithm at introduce nodes.

Let X_i be a forget node and v_k be the vertex that is forgotten at X_i. Since v_k is forgotten, we increase the number of used up valences of each vertex $v_l \in X_i$ from $a_l - e_l$ to a_l. Since v_k does not occur in X_i, we reduce the valences of vertices in X_i, which are

consumed by bonds within X_i, by the bond order of bonds between these vertices and v_k. Furthermore, the algorithm also assigns v_k the valence minimizing the corresponding entry in D_i. Therefore, our algorithm is correct at forget node.

At a join node X_i, the total number of valences consumed outside X_i of an atom results from the number of its valences that are consumed in the subtree rooted at X_j and the subtree rooted at X_h. Recall that $Y_j \setminus X_j$ and $Y_h \setminus X_h$ are disjoint. This confirms the correctness of our algorithm at join nodes.

The correctness of this algorithm at the root of the tree decomposition is analogous to the correctness of our previous algorithm introduced in Section 6.6.1.

All in all, our algorithm compute the optimal solution of the Bond Order Assignment Problem in time $O(\alpha^{3\omega} \cdot \omega \cdot m)$. □

To compute not only the optimal score but also the optimal assignment, we again store where the minimum is obtained for forget nodes and join nodes during the bottom-up processing. Afterwards, the optimal bond order assignment can be obtained by a top-down traversal in time $O(m)$.

From theoretical point of view, this algorithm drastically outperforms the algorithm in Section 6.6.1, since its running time only exponentially depends on ω, whereas the running time of the algorithm in Section 6.6.1 exponentially depends on β, which, in turn, can be as large as $O(\omega^2)$. Thus, this algorithm could be more efficient in practice if subgraphs induced by vertices in a bag of the tree decomposition are dense and the maximum valence of atom in the molecule graph is small. But this usually does not occur in practice, therefore we only implemented the algorithm with running time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$ introduced in Section 6.6.1.

6.7 Algorithm Engineering

In this section, we describe some heuristic improvements included in our implementation to reduce both running time and memory usage of our approach in applications.

Search Space Reduction Since our algorithm works with open valences, and bond orders are bounded by two, open valences of every atom v that exceed $2 \cdot \deg(v)$ cannot be fully consumed. Thus, given a Bond Order Assignment Problem instance, we first remove open valences of every vertex $v \in V$ that are larger than $2 \cdot \deg(v)$ to reduce the search space of the problem.

Avoid futile matrix entries During the course of the dynamic programming algorithm, we do not have to compute or store entries $D_i[a_1, \ldots, a_k; e_{1,2}, \ldots, e_{k-1,k}]$ with $a_l + \sum_j e_{l,j} > A^*_{v_l}$ for some l, because such entries will never lead to a feasible bond order assignment. Analogously, if the algorithm introduced in Section 6.6.2 is implemented, matrix entries $D_i[a_1, \ldots, a_k; b_1, \ldots, b_k]$ with $a_l + b_l > A^*_{v_l}$ can be ignored.

Decision versus Optimization When using the score function provided Wang *et al.* [152], score matrices D often contain many entries with scores much larger than the optimal score. Those entries never lead to the optimal solution, but make the score matrices unnecessary large. To avoid those entries, we imitate the idea of depth-bounded search tree algorithm: Instead of directly compute the optimal solution, we initialize an integer $k = 0$ and use our tree decomposition-based dynamic programming algorithm to answer the question if there is solution with score at most k. During the bottom-up traversal, we do not store matrix entries with score exceeding k. If the score of an optimal solution is at most k, an optimal solution will be found. Otherwise, we call our algorithm repeatedly with increasing k, until an optimal solution is found. Furthermore, we also remove all atom valences with scores larger than k before executing our dynamic programming algorithm. This artifice accelerates our algorithm drastically in practice.

6.8 Computational Results

To evaluate the performance of our algorithm, we implemented the algorithm introduced in Section 6.6.1 in Java and compared its running time with the heuristic algorithm of Wang *et al.* [152] on a set of real molecule graphs. All evaluations were done on an Intel Core Duo 1.83 GHz with 1 GB of memory running Linux 2.6.34.

6.8.1 Implementation and Dataset

In our implementation, we use hash maps instead of arrays to implement score matrices D to avoid allocating memory for unused entries.

To compute optimal tree decompositions of molecule graphs, we use the method QuickBB in the library LibTW implemented by van Dijk *et al.* (`http://www.treewidth.com`). The QuikBB method is an implementation of the branch-and-bound algorithm of Gogate *et al.* [67] to compute optimal tree decomposition. The computed optimal tree decomposition is transformed into nice tree decompositions as described in Section 2.3.2.

For our evaluation, we used the molecule graphs in the MMFF94 dataset[1] by Halgren *et al.* [81], which consists of 761 molecule graphs predominantly derived from the Cambridge Structural Database. This dataset has been suggested to us by experts, as it is considered to contain "hard" instances of the problem, where atoms have non-standard valences. Bond orders are given in the dataset but we ignored this information and reassigned the bond orders to all molecule graphs. We removed four molecule graphs that contain elements such as iron not covered in our scoring table (see below), or that have atom bindings such as chlorine atoms connected to four other atoms, which is also not covered in our scoring. The largest molecule graphs contains 59 atoms, the smallest 3 atoms, the average 23 atoms. We find that 20.21 % of the remaining 757 molecule graphs have treewidth one, 96.69 % have treewidth ≤ 2, and all molecule graphs have treewidth at most three. The average treewidth is 1.83.

[1] `http://www.ccl.net/cca/data/MMFF94/`, source file `MMFF94_dative.mol2`, of Feb. 5, 2009

As score function, we use the scoring table from Wang *et al.* [152]. This scoring allows atoms to have rather "exotic" valences, but gives an *atomic penalty score* (aps) to these rare valence states. As an example, carbon is allowed to take valence two (with aps 64), three (aps 32), four (aps 0), five (aps 32), or six (aps 64). In addition, different scores can be applied for the same element, depending on the local neighborhood: For example, carbon in a carboxylate group COO^- can take valence four (aps 32), five (aps 0), or six (aps 32). See Table 2 in [152] for details.

Thus, to assign a score function to an atom of a molecule graph using Wang *et al.*'s scoring table, it is necessary to identify, which molecular pattern listed in the Wang *et al.*'s scoring table is formed by the atom and its neighborhood. In general, this problem corresponds to the NP-hard Subgraph Isomorphism Problem [65], that asks for an occurrence of a pattern graph in a host graph. Fortunately, the molecular patterns listed in the Wang *et al.*'s scoring table are very small and simple. Thus, occurrences of those molecular patterns can be detected efficiently. However, to make our implementation work with any scoring table, which may contain larger and more complicated molecular patterns, we use the SMARTSQueryTool in the *Chemistry Development Kit* (CDK)[2] [146] to detect occurrences of molecular patterns in a molecule graph, instead of implementing a specific procedure to detect small and simple molecular patterns listed in the Wang *et al.*'s scoring table.

6.8.2 Comparison with other approaches

In Table 6.1, we report the running times of our algorithm and the heuristic algorithm of Wang *et al.* [152] in milliseconds. Thereby, we rounded the running times up to two significant digits. In this evaluation, we used the implementation of the Wang *et al.*' heuristic algorithm found in the Version 1.27 of the Antechamber program[3].

Total running times of our algorithm are always below one second, and 5.78 ms on average. The average running time for assigning a score function for each atom in a molecule graph using the SMARTSQueryTool is 5.22 ms. The average running times for computing optimal tree decomposition of a molecule graph and transforming the optimal tree decomposition into a nice tree decomposition is 0.14 ms. The average running times of our tree decomposition-based dynamic programming algorithm is 0.41 ms. Assigning score function to every atom of a molecule graph is the most time-consuming task, when optimal bond order assignment is computed using Wang *et al.*'s scoring table.

The average running time of the heuristic algorithm of Wang *et al.* [152] is 11.40 ms. This heuristic algorithm is slower than our exact algorithm on almost every group of molecule graphs listed in Table 6.1, except for the group of molecule graphs with 51 to 59 vertices. In this group, our algorithm is slightly slower than the heuristic algorithm. This is due to the time-consuming task of assigning score functions to atoms using the SMARTSQueryTool. The practical running time of our algorithm can be improved, if necessary, by using a better algorithm to detect the small and simple molecular patterns

[2] `http://cdk.sourceforge.net/`
[3] `http://ambermd.org/`

instance size		number of		average	running time			comparison	
$\|V\|$	$\|E\|$	instances	tw	tw	SA	TD	DP	$\sum$	AA
3–10	2–11	73	1–2	1.15	1.12	1.06	2.11	4.29	4.88
11–20	10–22	214	1–3	1.77	3.95	0.01	0.08	4.04	7.49
21–30	20–33	333	1–3	1.97	5.89	0.06	0.74	6.69	13.81
31–40	30–43	129	1–3	1.95	7.37	0.62	0.38	8.37	13.51
41–50	40–53	5	1–2	1.8	9.8	1	0.4	11.2	57.6
51–59	53–61	3	2	2.0	12.34	1.34	0.67	14.35	13.67

Table 6.1: Overview on the data used in our experiment. "tw" gives the range of treewidths in this group. "SA' is average running time for a assigning score function to every atom of a molecule graph. "TD" is average running time for computing optimal tree decomposition and transforming it into a nice tree decomposition. "DP" is average running time of our dynamic programming algorithm. "$\sum$" is total running time of our algorithm on average, and $\sum$=SA + TD + DP. "AA" is average running time of Wang *et al.*'s heuristic algorithm.

listed in Wang *et al.*'s scoring table, instead of using the SMARTSQueryTool, which is more general for any type of molecular patterns.

Recently, Dehof *et al.* [49] evaluated a previous implementation of our algorithm together with other Bond Order Assignment Problem solvers [50], an integer linear programming-based algorithm, a runtime-heuristic algorithm, and the heuristic algorithm of Wang *et al.* [152]. All three exact solvers were able to recover over 78% reference solutions in the MMFF94 dataset, whereas the heuristic algorithm of Wang *et al.* [152] was able to recover about only 37% reference solutions. Concerning the running time of the algorithms, Dehof *et al.* reported that our tree decomposition-based algorithm outperforms two other exact Bond Order Assignment Problem solvers and almost as fast as the heuristic algorithm of Wang *et al.* [152]. However, as we showed in Table 6.1, the current implementation of our algorithm outperforms the Wang *et al.*'s heuristic algorithm [152] on almost every molecule graph in the MMFF94 dataset.

Although our algorithm is the fastest algorithm from all algorithms evaluated by Dehof *et al.*, we would like to mention that the running time of our algorithms grow exponentially with the tree width of molecule graphs. Therefore, it is recommended to use other solver for input graphs with large treewidth, like spherical fullerenes (buckyball) or carbon nanotube (buckytubes).

Still, we can claim that our algorithms are efficient on biomolecule graphs, despite their (super-)exponential running times in the treewidth of biomolecule graphs. We justify our claim by the following observations:

Definition 6.8. A graph is *outerplanar* if it admits a crossing-free embedding in the plane such that all vertices are on the same face. A graph is 1-*outerplanar* if it is

outerplanar; and it is *r-outerplanar* for $r > 1$ if, after removing all vertices of the boundary face, the remaining graph is $(r-1)$-outerplanar.

Bodlaender *et al.* [28] showed the following important property of r-outerplanar graphs.

Theorem 11. *Every r-outerplanar graph has treewidth at most $3r-1$*

From this theorem, it holds that the Bond Order Assignment Problem is actually fixed-parameter tractable with respect to the maximum valence and the value r in an r-outerplanar molecule graph. Furthermore, we find that molecule graphs of biomolecules are usually r-outerplanar for some small integer r, such as $r = 2$ for proteins and DNA.

To empirically confirm our claim, we tested it on molecules from the PubChem database at `http://pubchem.ncbi.nlm.nih.gov/` [138], which contains more than 60 million entries in Jan 2010. We computed the treewidths of all molecule graphs in eight files randomly chosen from 1 782 files found at `ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/XML`. For all 135 607 connected molecule graphs in these files, we computed the exact treewidth using the QuickBB method of the LibTW library. We found that 12 004 (8.85%) molecule graphs have treewidth one, 121 267 (89.43%) have treewidth two, 2 192 (1.62%) have treewidth three, and for seven (0.01%) molecules, the QuickBB method cannot determine the treewidth after ten minutes of computation. According to the upper bound computed by the QuickBB method, the treewidth of these seven molecule graphs is at most four. The database also contains 137 (0.1%) molecules consisting of a single ion, for which the Bond Order Assignment Problem is trivial. There are no molecule graphs in the eight files with treewidth exceeding four. This confirms our claim that our algorithms can be efficiently applied to solve the Bond Order Assignment Problem on biomolecule graphs.

6.9 Summary and Outlook

In this chapter, we investigated the Bond Order Assignment Problem that requires to (re)assign bond orders to molecule graphs minimizing a given score function, and achieved the following results:

- Bond Order Assignment Problem is NP-hard even on molecule graph with vertex degree bounded by three and valence state of atoms bounded by four. However, this problem can be solved in polynomial time if the input molecule graph is a tree.
- Bond Order Assignment Problem is inapproximable even on molecule graph with vertex degree bounded by three and valence state of atoms bounded by four, unless P=NP.
- The maximization variant of Bond Order Assignment Problem is MAX SNP-hard. This result prohibits the hope for a polynomial-time approximation scheme (PTAS) for the maximization variant of Bond Order Assignment Problem.

- We introduced two tree decomposition-based dynamic programming algorithms for the Bond Order Assignment Problem, which are fixed-parameter tractable with respect to the treewidths of molecule graphs and the maximum atom valence in the molecule graph. We also discuss some heuristic improvements to accelerate our algorithm in practical use.
- We implemented and evaluated one of our algorithms on the molecule graphs of the MMFF94 dataset. It turns out that our algorithm is very efficient on this dataset.

Since the running times of our algorithms grow exponentially in the tree width of an input molecule graph, we wanted to verify if the treewidths of biomolecule graphs are small in general. Therefore, we computed the treewidth of 135 607 biomolecule graphs randomly chosen from the PubChem database, and found that most of these molecule graphs have treewidths two and none of these molecule graphs has treewidth exceeding four. This fact also points out that tree decomposition-based dynamic programming technique is a very promising approach to develop algorithms for combinatorial problems on molecule graphs.

Future Research

Regarding the Bond Order Assignment Problem, we want to reduce the dependencies of our algorithms on treewidth, which may allow us to solve more complicated structures. We also plan to do algorithm engineering towards efficient memory usage. At the moment, the Java implementation of our algorithm is available at `http://bio.informatik.uni-jena.de/software`, but we plan to re-implement our algorithm in C++ and integrate it in the *Biochemical Algorithms Library* (BALL) [98], which already contains the implementations of the Bond Order Assignment Problem exact solvers by Dehof *et al.* [50].

Since treewidths of biomolecule graphs are usually very small, we want to apply tree decomposition-based dynamic programming algorithm to further combinatorial problems on biomolecule graphs that are relevant in applications. As an example, we plan to investigate the Subgraph Isomorphism Problem on molecule graphs that asks for occurrences of a molecular pattern in a molecule graph.

As mentioned above, this Subgraph Isomorphism Problem is also encountered when computing bond orders assignment problem using a scoring table that assigns penalty function to an atom according to the neighborhood of the atom, for example the Wang *et al.*'s scoring table. A faster algorithm for the Subgraph Isomorphism Problem on molecule graph also accelerates our algorithm for the Bond Order Assignment Problem in practice.

A further application of Subgraph Isomorphism Problem on molecule graph is encountered when computing the bond order information of biopolymers. For example, to compute the bond order information of a protein molecule, we can determine the amino acids on the protein, and assign every bond of each amino acid its corresponding bond order.

In general, Subgraph Isomorphism Problem appears in many important applications in bioinformatics [114,128] and there are several tree decomposition-based dynamic programming algorithm for the Subgraph Isomorphism Problem [80,117,143]. However, we would like to investigate this problem particularly on molecule graphs, since besides small treewidths, molecule graphs also have certain properties that can be used to achieve efficient algorithm for practical use. For example, an atom on the pattern graph can be only matched with another atom of the same type in the host graph. Making use of such properties may help to shrink the search space of the Subgraph Isomorphism Problem.

Moreover, we also want to investigate a generalization of the Subgraph Isomorphism Problem, the Maximum Common Subgraph that asks for the maximum common subgraph of two input graphs. This problem also has many applications on molecule graphs and has been investigated in several publications [47, 64, 133]. We hope that tree decomposition-based dynamic programming can be applied to solve this problems optimally in practical use.

7 Conclusion

In this thesis, we investigated three NP-hard problems arising in bioinformatics: the Flip Consensus Tree Problem, the Weighted Cluster Editing Problem and the Bond Order Assignment Problem. In contrast to heuristic approaches, that are usually used in bioinformatics to estimate good solutions for NP-hard problems, we developed fixed-parameter algorithms to compute exact solutions for those problems. In the following, we briefly recapitulate our results presented in this work.

Based on the graph-theoretical model of the Flip Consensus Tree Problem, we presented two fixed-parameter algorithms with running times $O(4.83^k + poly(m, n))$ and $O(4.42^k + poly(m, n))$, that improved the previous fixed-parameter algorithm from [152] with running time $O(6^k mn)$. Besides the two new algorithms, we also introduced a set of data reduction rules and discussed several heuristic improvement to speed up our algorithms in practice. We implemented and evaluated one of our algorithms on real biological data. Computational results show that our algorithm outperforms the previous 6^k algorithm and much faster than one would expect from the theoretical running time. However, the Flip Supertree Problem, the main problem we wanted to solve with fixed-parameter algorithmic approach, turns out to be W[2]-hard with respect to the number of flips [21]. Solving the Flip Supertree Problem optimally in practice remains a challenge for future research.

From the computational point of view, the Flip Consensus Tree Problem is very similar to the Weighted Cluster Editing Problem. Whereas the Flip Consensus Tree Problem asks for a minimum set of edge modifications to transform a bipartite graph into an M-free graph, the Weighted Cluster Editing Problem asks for set of edge modifications of minimum cost to transform a graph into a cluster graph. Since the Weighted Cluster Editing Problem generalizes the Unweighted Cluster Editing Problem, algorithmic results for the Weighted Cluster Editing Problem can be easily applied to the Unweighted Cluster Editing Problem, but the other direction does not always hold. In this work, we modified the depth-bounded search tree algorithm of Gramm *et al.* [70] for the Unweighted Cluster Editing Problem and introduced an algorithm with running time $O(2.42^k + |V|^3 \log |V|)$ for the Weighted Cluster Editing Problem. We mentioned a novel branching strategy developed by our group, that leads to an algorithm with running time $O(1.82^k + |V|^3)$ for the Weighted Cluster Editing Problem. To the best of our knowledge, this is the fastest fixed-parameter algorithm for the Weighted, as well as the Unweighted Cluster Editing Problem, at time of preparing this work. Computational results in [18] showed that this algorithm can solve Weighted Cluster Editing Problem instances with several hundred edge modifications.

For the Bond Order Assignment Problem, we presented an NP-hardness proof

by a polynomial reduction from the 3-SAT* problem, from which we also infer the inapproximability of the Bond Order Assignment Problem. We then proved that the maximization variant of the Bond Order Assignment Problem is MAX SNP-hard. Thus, there is no hope for a PTAS for the maximization Bond Order Assignment Problem. However, we showed that the Bond Order Assignment Problem on trees can be solved in polynomial time by a dynamic programming algorithm. Based on the idea of this dynamic programming algorithm on trees, we applied the tree decomposition-based dynamic programming to the Bond Order Assignment Problem and presented two algorithms for the Bond Order Assignment Problem with running time $O(\alpha^{2\omega} \cdot 3^{\beta} \cdot \omega \cdot m)$ and $O(\alpha^{3\omega} \cdot \omega \cdot m)$. Those algorithms are in fact fixed-parameter algorithms with respect to the maximum atom valence and the treewidth of molecule graphs. We implemented one of our algorithms and evaluated the performance of our algorithm on a molecule dataset. Due to the small treewidth of molecule graphs, our algorithm turn out to be very efficient on this dataset. In particular, our algorithm also outperforms the previous heuristic algorithm of Wang *et al.* [152] for the Bond Order Assignment Problem. To further confirm the practical use of our algorithm, we measured the treewidths of over hundred thousands molecule graphs randomly chosen from the Pubchem database, and found that the treewidths of those molecule graphs are mostly two and never exceed four. As a direction of our future research regarding tree-decomposition based dynamic programming, we plan to investigate the Subgraph Isomorphism Problem and the Largest Common Subgraph Problem on molecule graphs.

In the following, we conclude this work by recapitulating some general techniques of fixed-parameter algorithmics that are applied to achieve the results presented in this work, and can be applied to the open problems discussed along this work.

Parameterizing. The first task to develop fixed-parameter algorithm is to find an appropriate parameter to investigate. The choice of a parameter depends on whether the chosen parameter will be small in practical problem instances, and whether the problem is fixed-parameter tractable with respect to that parameter.

Data Reduction. Data reduction helps to cut down the size of problem instance without changing the solution of the original problem instance. Thus data reduction is highly recommended in combination with any type of algorithmic approach, no matter if it is a fixed-parameter algorithm, an approximation algorithm, a heuristic algorithm, or an integer linear programming algorithm. Furthermore, data reduction can lead to a problem kernel of a parameterized problem, that in turn confirms the fixed-parameter tractability of the problem.

Algorithm Design. Although there is no recipe of designing algorithms, some general algorithm design techniques do exist and have been successfully applied in many cases. One of the most frequently used technique to design fixed-parameter algorithms is the depth-bounded search tree approach. While an initial branching strategy is obvious in

many cases, a good branching strategy requires a careful design and mostly goes hand in hand with complicated case analysis. However, the careful designing of a branching strategy often leads to a smaller search tree. For example, the trivial branching strategy for the Flip Consensus Tree Problem generates a search tree of size $O(6^k)$, whereas our carefully designed branching strategies lead to search trees of size bounded by $O(4.83^k)$ and $O(4.42^k)$. The 4.83^k algorithm is also much faster than the 6^k algorithm, as shown by our computational results. Unfortunately, in some cases, a complicated branching strategy also increases the practical running time of a depth-bounded search tree algorithm. For example our 2.42^k-algorithm for the Weighted Cluster Editing Problem is less efficient than the trivial 3^k algorithm in practice (see [18] for more details).

Dynamic programming is also a classical algorithmic approach for combinatorial problems, where solutions of a problem instance can be computed from solutions of its subproblem instances. For combinatorial problems on graphs that can be solved efficiently with dynamic programming on trees, it is worthwhile to consider tree decomposition-based dynamic programming algorithm for input graph with bounded treewidths. In many cases, tree decomposition-based dynamic programming algorithms are very efficient in practice, despite their unimpressive theoretical running time, especially when the treewidths of the input graphs are small.

Besides depth-bounded search tree and dynamic programming, several other advanced techniques such as color-coding, iterative compression, greedy localization are also frequently applied to design fixed-parameter algorithms. See [121] for a detailed introduction to those techniques.

Implementation and Algorithm Engineering. An algorithm should be implemented and evaluated on real problem instances to confirm its practical use. Although algorithm-engineering may worsen the theoretical running of an algorithm in some cases, it usually improves the practical running time of the algorithm. In general, algorithm engineering is an important part of algorithm development.

Besides fixed-parameter algorithmic approach, integer linear programming is a classical approach, that is usually used in practice to solve NP-hard problem exactly. Thus, if a parameterized problem is proven to be fixed-parameter intractable, integer linear programming algorithm in combination with efficient data reduction should be considered as an alternative approach. Although integer linear programming algorithms may not be so efficient as fixed-parameter algorithms for certain problems, their appeal is due to the inexpense in development compared to the development of a fixed-parameter algorithm.

Bibliography

[1] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, 1981. 31

[2] T. Akutsu, H. Arumura, and S. Shimozono. On approximation algorithms for local multiple alignment. In *Proc. of Research in Computational Molecular Biology (RECOMB 2000)*, pages 1–7. ACM Press, 2000. 3

[3] J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proc. of Latin American Theoretical Informatics Symposium (LATIN 2002)*, 2002. 18

[4] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell.* Taylor & Francis, 2007. 23

[5] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embedding in a k-tree. *SIAM J. Algebra. Discr.*, 8:277–284, 1987. 19

[6] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Appl. Math.*, 23:11–24, 1989. 13

[7] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998. 72

[8] G. Ausiello, P. Crescenzi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 1999. 12

[9] J. C. Baber and E. E. Hodgkin. Automatic assignment of chemical connectivity to organic molecules in the cambridge structural database. *J. Chem. Inf. Model.*, 32:401–406, 1992. 69

[10] B. R. Baum. Combining trees as a way of combining data sets for phylogenetic inference, and the desirability of combining gene trees. *Taxon*, 41(1):3–10, 1992. 31

[11] B. R. Baum and M. A. Ragan. The MRP method. In O. R. Bininda-Emonds, editor, *Phylogenetic supertrees: Combining information to reveal the Tree of Life*, volume 4 of *Computational Biology Book Series*, chapter 1, pages 17–34. Kluwer Academic, 2004. 31

[12] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *J. Comput. Biol.*, 6(3-4):281–297, 1999. 57, 58

[13] P. Berman, M. Karpinski, and A. D. Scott. Computational complexity of some restricted instances of 3-sat. *Discrete Appl. Math.*, 155:649–653, 2007. 73

[14] M. W. Bern, E. L. Lawler, and A. L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *J. Algorithms*, 8:216–235, 1987. 13

[15] G. E. Blelloch, K. Dhamdhere, E. Halperin, R. Ravi, R. Schwartz, and S. Sridhar. Fixed parameter tractability of binary near-perfect phylogenetic tree reconstruction. In *Proc. of International Colloquium on Automata, Languages and Programming (ICALP 2006)*, pages 667–678, 2006. 4

[16] S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truss. A fixed-parameter approach for weighted cluster editing. In *Proc. of Asia-Pacific Bioinformatics Conference (APBC 2008)*, volume 5 of *Series on Advances in Bioinformatics and Computational Biology*, pages 211–220. Imperial College Press, 2008. iii, v, 1, 22, 57, 66, 67

[17] S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truss. Going weighted: Parameterized algorithms for cluster editing. In *Proc. of Conference on Combinatorial Optimization and Applications (COCOA 2008)*, volume 5165 of *Lect. Notes Comput. Sc.*, pages 1–12. Springer, 2008. 1, 22, 67

[18] S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truss. Going weighted: Parameterized algorithms for cluster editing. *Theor. Comput. Sci.*, 410(52):5467–5480, 2009. iii, v, 1, 4, 5, 22, 57, 59, 60, 61, 62, 65, 66, 67, 91, 93

[19] S. Böcker, S. Briesemeister, and G. W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. In *Proc. of Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *Lect. Notes Comput. Sc.*, pages 289–302. Springer, 2008. 67

[20] S. Böcker, S. Briesemeister, and G. W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 2009. In press, doi:10.1007/s00453-009-9339-7. 4, 12, 22

[21] S. Böcker, Q. B. A. Bui, S. Guillemot, and A. Truss. Fixed-parameter complexity of minimum-flip supertrees. In preparation. 1, 5, 15, 34, 55, 67, 91

[22] S. Böcker, Q. B. A. Bui, P. Seeber, and A. Truss. Computing bond types in molecule graphs. In *Proc. of Computing and Combinatorics Conference (COCOON 2009)*, volume 5609 of *Lect. Notes Comput. Sc.*, pages 297–306. Springer, 2009. 1

[23] S. Böcker, Q. B. A. Bui, and A. Truss. An improved fixed-parameter algorithm for minimum-flip consensus trees. In *Proc. of International Workshop on Parameterized and Exact Computation (IWPEC 2008)*, volume 5018 of *Lect. Notes Comput. Sc.*, pages 43–54. Springer, 2008. 1

[24] S. Böcker, Q. B. A. Bui, and A. Truss. Improved fixed-parameter algorithms for minimum-flip consensus trees. *ACM T. Algorithms*, Jun 2009. Accepted for publication. 1, 46

[25] S. Böcker, Q. B. A. Bui, and A. Truss. Computing bond orders in molecule graph. *Theor. Comput. Sci.*, Dec 2010. Accepted for publication. 1

[26] H. L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP '88: Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, 1988. 13

[27] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. 19

[28] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209:1–45, 1998. 88

[29] H. L. Bodlaender, M. R. Fellows, P. Heggernes, F. Mancini, C. Papadopoulos, and F. Rosamond. Clustering with partial information. *Theor. Comput. Sci.*, 411:1202–1211, 2010. 67

[30] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, D. M. Thilikos, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. In *Proc. of Conference on Annual European Symposium (ESA 2006)*, pages 672–683, 2006. 19

[31] T. A. Brown. *Genomes*. John Wiley & Sons, 2002. 23

[32] S. Bruckner, F. Hüffner, R. M. Karp, R. Shamir, and R. Sharan. Topology-free querying of protein interaction networks. In *Proc. of Research in Computational Molecular Biology (RECOMB 2009)*, volume 5541 of *Lect. Notes Comput. Sc.*, pages 74–89. Springer, 2009. 4

[33] D. Bryant and V. Moulton. Neighbor-net: an agglomerative method for the construction of phylogenetic networks. *Mol. Biol. Evol.*, 21(2):255–265, Feb 2004. 28

[34] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Inf. Process. Lett.*, 58(4):171–176, 1996. 17

[35] D. Chen, L. Diao, O. Eulenstein, D. Fernández-Baca, and M. J. Sanderson. Flipping: A supertree construction method. In *Bioconsensus*, volume 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 135–160. American Mathematical Society, 2003. 34

[36] D. Chen, O. Eulenstein, D. Fernández-Baca, and J. G. Burleigh. Improved heuristics for minimum-flip supertree construction. *Evol. Bioinform. Online*, 2:391–400, 2006. 34

[37] D. Chen, O. Eulenstein, D. Fernández-Baca, and M. Sanderson. Supertrees by flipping. In *Proc. of Conference on Computing and Combinatorics (COCOON 2002)*, volume 2387 of *Lect. Notes Comput. Sc.*, pages 391–400. Springer, 2002. 34

[38] D. Chen, O. Eulenstein, D. Fernández-Baca, and M. Sanderson. Minimum-flip supertrees: complexity and algorithms. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 3(2):165–173, 2006. iii, 3, 5, 29, 30, 31, 33, 34, 35, 36, 53, 55

[39] J. Chen, I. A. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In *Proc. of Mathematical Foundations of Computer Science (MFCS 2006)*, volume 4162 of *Lect. Notes Comput. Sc.*, pages 238–249. Springer, 2006. 15

[40] J. Chen and J. Meng. A $2k$ kernel for the cluster editing problem. In *Proc. of Computing and Combinatorics Conference (COCOON 2010)*, volume 6196 of *Lect. Notes Comput. Sc.*, pages 459–468. Springer, 2010. 59

[41] M. Chimani, S. Rahmann, and S. Böcker. Exact ILP solutions for phylogenetic minimum flip problems. In *Proc. of ACM Conf. on Bioinformatics and Computational Biology (ACM-BCB 2010)*, pages 147–153, 2010. 12, 34, 56

[42] B. Chor and T. Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21 Suppl 1:i97–106, Jun 2005. 3, 29

[43] B. Chor and T. Tuller. Finding a maximum likelihood tree is hard. *J. ACM*, 53(5):722–744, 2006. 3, 29

[44] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971. 10

[45] S. A. Cook. An overview of computational complexity. *ACM*, 26:400–408, 1983. 7

[46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, first edition, 1990. 8, 9

[47] B. Cuissart, F. Touffet, B. Crémilleux, R. Bureau, and S. Rault. The maximum common substructure as a molecular depiction in a supervised classification context: experiments in quantitative structure/biodegradability relationships. *J. Chem. Inf. Comput. Sci.*, 42(5):1043–1052, 2002. 90

[48] R. Dakin. A tree-search algorithm for mixed integer programming problems. *Computer Journal*, 8:250–255, 1965. 11

[49] A. K. Dehof, A. Rurainski, Q. B. A. Bui, S. Böcker, H.-P. Lenhof, and A. Hildebrandt. Automated bond order assignment as an optimization problem. *Bioinformatics*, Dec 2010. Accepted for publication. 1, 87

[50] A. K. Dehof, A. Rurainski, H.-P. Lenhof, and A. Hildebrandt. Automated bond order assignment as an optimization problem. In *Proc. of German Conference in Bioinformatics (GCB 2009)*, Lecture Notes in Informatics, pages 201–210. Gesellschaft für Informatik, 2009. 12, 70, 87, 89

[51] R. L. DesJarlais, G. L. Seibel, I. D. Kuntz, P. S. Furth, J. C. Alvarez, P. R. O. de Montellano, D. L. DeCamp, L. M. Babé, and C. S. Craik. Structure-based design of nonpeptide inhibitors specific for the human immunodeficiency virus 1 protease. *Proc. Natl. Acad. Sci. USA*, 87(17):6644–6648, Sep 1990. 69

[52] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 3, 7, 13, 15, 22

[53] J. H. V. Drie, D. Weininger, and Y. C. Martin. Aladdin: an integrated tool for computer-assisted molecular design and pharmacophore recognition from geometric, steric, and substructure searching of three-dimensional molecular structures. *J. Comput. Aided Mol. Des.*, 3(3):225–251, Sep 1989. 69

[54] I. Elias. Settling the intractability of multiple alignment. *J. Comput. Biol.*, 13(7):1323–1339, Sep 2006. 3

[55] A. Enright, V. Kunin, and C. Ouzounis. Protein families and tribes in genome sequence space. *Nucleic Acids Res*, 31(15):4632–4638, 2003. 57, 58

[56] O. Eulenstein, D. Chen, J. G. Burleigh, D. Fernández-Baca, and M. J. Sanderson. Performance of flip supertree construction with a heuristic algorithm. *Syst. Biol.*, 53(2):299–308, 2004. 32, 33, 34, 56

[57] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Sunderland, Massachusetts, 2004. 28, 29

[58] W. M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155:279–84, 1967. 29

[59] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006. 7

[60] L. Foulds and R. L. Graham. The Steiner problem in phylogeny is NP-complete. *Adv. Appl. Math.*, 3:43–49, 1982. 3, 29, 32

[61] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, Feb 2007. 57, 58

[62] M. Froeyen and P. Herdewijn. Correct bond order assignment in a molecular framework using integer linear programming with application to molecules where only non-hydrogen atom coordinates are available. *J. Chem. Inf. Model.*, 5:1267–1274, 2005. 70

[63] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA 1990)*, pages 434–443. Society for Industrial and Applied Mathematics, 1990. 45, 46

[64] E. J. Gardiner, P. J. Artymiuk, and P. Willett. Clique-detection algorithms for matching three-dimensional molecular structures. *J. Mol. Graph. Model.*, 15(4):245–253, Aug 1997. 90

[65] M. R. Garey and D. S. Johnson. *Computers and Intractability (A Guide to Theory of NP-Completeness)*. Freeman, New York, 1979. 3, 7, 8, 9, 10, 11, 13, 86

[66] F. W. Glover and G. A. Kochenberger. *Handbook of Metaheuristics*. Springer, 2003. 11

[67] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proc. of Conference on Uncertainty in artificial intelligence (UAI 2004)*, pages 201–20, 2004. 19, 85

[68] P. A. Goloboff. Minority rule supertrees? MRP, compatibility, and minimum flip may display the least frequent groups. *Cladistics*, 21(3):282–294, 2005. 34

[69] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004. 51, 59

[70] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. *Theor. Comput. Syst.*, 38(4):373–392, 2005. 59, 62, 63, 91

[71] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction, exact, and heuristic algorithms for clique cover. In *Proc. of Workshop on Algorithm Engineering and Experiments (ALENEX 2006)*, pages 86–94, 2006. To appear in *ACM Journal of Experimental Algorithmics*. 67

[72] I. Gronau and S. Moran. Neighbor joining algorithms for inferring phylogenies via lca distances. *J. Comput. Biol.*, 14(1):1–15, 2007. 29

[73] M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Math. Program.*, 45:52–96, 1989. 59

[74] H. Guo, R. Renaut, K. Chen, and E. Reiman. Clustering huge data sets for parametric pet imaging. *Biosystems*, 71(1-2):81–92, Sep 2003.

[75] J. Guo, F. Hüffner, C. Komusiewicz, and Y. Zhang. Improved algorithms for bicluster editing. In *Proc. of Conference on Theory and Applications of Models of Computation (TAMC 2008)*, volume 4978 of *Lect. Notes Comput. Sc.*, pages 445–456. Springer, 2008. 67

[76] J. Guo, C. Komusiewicz, R. Niedermeier, and J. Uhlmann. A more relaxed model for graph-based data clustering: s-plex editing. In *International Conference on Algorithmic Aspects in Information and Management (AAIM 2009)*, 2009. 59

[77] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19–28, 1991. 32, 33, 36

[78] D. Gusfield and V. Bansal. *A Fundamental Decomposition Theory for Phylogenetic Networks and Incompatible Characters*, volume 3500. Jan. 2005. 28

[79] P. Haiwei, J. Li, and Z. Wei. Medical image clustering for intelligent decision support. *Conf. Proc. IEEE Eng. Med. Biol. Soc.*, 3:3308–3311, 2005. 57

[80] M. T. Hajiaghayi and N. Nishimura. Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. *J. Comput. Syst. Sci.*, 73(5):755–768, 2007. 90

[81] T. A. Halgren. MMFF VI. MMFF94s option for energy minimization studies. *J. Comp. Chem.*, 17(5-6):490–519, 1996. 70, 85

[82] L. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Springer, 2002. 10

[83] K. Hemm, K. Aberer, and M. Hendlich. Constituting a receptor-ligand information base from quality-enriched data. *Proc. Int. Conf. Intell. Syst. Mol. Biol.*, 3:170–178, 1995. 69

[84] M. Hendlich, F. Rippmann, and G. Barnickel. BALI: automatic assignment of bond and atom types for protein ligands in the brookhaven protein databank. *J. Chem. Inf. Model.*, 37:774–778, 1997. 69, 70

[85] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Longman, 2001. 10

[86] W.-L. Hsu and T.-H. Ma. Substitution decomposition on chordal graphs and applications. In *Proc. of International Symposium on Algorithms (ISA 1991)*, volume 557 of *Lect. Notes Comput. Sc.*, pages 52–60. Springer, 1991. 39

[87] F. Hüffner, N. Betzler, and R. Niedermeier. Separator-based data reduction for signed graph balancing. *J. Comb. Optim.*, 20(4):335–360, 2009. 4

[88] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding to facilitate signaling pathway detection. In *Proc. of Asia-Pacific Bioinformatics Conference (APBC 2007)*, number 5 in Advances in Bioinformatics and Computational Biology, pages 277–286. Imperial College Press, January 2007. 4

[89] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 2007. To appear. 4

[90] D. H. Huson and D. Bryant. Application of phylogenetic networks in evolutionary studies. *Mol. Biol. Evol.*, 23(2):254–267, Feb 2006. 28

[91] D. H. Huson and T. H. Kloepper. Computing recombination networks from binary sequences. *Bioinformatics*, 21 Suppl 2:ii159–ii165, Sep 2005. 28

[92] D. H. Huson, R. Rupp, and C. Scornavacca. *Phylogenetic Networks*. Cambridge University Press, 2011. 28

[93] T. Jiang, L. Wang, and K. Zhang. Alignment of trees: an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995. 13

[94] H. Joe and J. Ward. Hierarchical grouping to optimize an objective function. *J Amer Statist Assoc*, 58:236–244, 1963. 57, 58

[95] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972. 12

[96] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2005. 8, 9

[97] T. Kloks. *Treewidth, Computation and Approximation*. Springer, 1994. 19

[98] O. Kohlbacher and H. P. Lenhof. Ball–rapid software prototyping in computational molecular biology. biochemicals algorithms library. *Bioinformatics*, 16(9):815–824, Sep 2000. 89

[99] C. Komusiewicz and J. Uhlmann. A cubic-vertex kernel for flip consensus tree. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, Leibniz International Proceedings in Informatics, pages 280–291, 2008. 22, 34, 38, 40, 46, 47, 53

[100] A. Krause, J. Stoye, and M. Vingron. Large scale hierarchical clustering of protein sequences. *BMC Bioinformatics*, 6:15, 2005. 57

[101] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1998. 9

[102] O. Kullmann. New methods for 3-sat decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999. 16

[103] M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Inform.*, 23(3):311–323, 1986. 58

[104] P. Labute. On the perception of molecules from 3D atomic coordinates. *J. Chem. Inf. Model.*, 45(2):215–221, 2005. 70

[105] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960. 11

[106] E. Lang, C.-W. von der Lieth, and T. Förster. Automatic assignment of bond orders based on the analysis of the internal coordinates of molecular structures. *Anal. Chim. Acta*, 265:283–289, 1992. 70

[107] M. Leone, Sumedha, and M. Weigt. Clustering by soft-constraint affinity propagation: applications to gene-expression data. *Bioinformatics*, 23(20):2708–2715, Oct 2007. 57, 58

[108] B. Lewin. *Genes VII*. Oxford University Press, 1999. 23

[109] W.-H. Li. *Molecular Evolution*. Sinauer Associates, 1997. 27

[110] M. Liptrot, K. H. Adams, L. Martiny, L. H. Pinborg, M. N. Lonsdale, N. V. Olsen, S. Holm, C. Svarer, and G. M. Knudsen. Cluster analysis in kinetic modelling of the brain: a noninvasive alternative to arterial sampling. *Neuroimage*, 21(2):483–493, Feb 2004. 57

[111] D. J. Lockhart and E. A. Winzeler. Genomics, gene expression and dna arrays. *Nature*, 405(6788):827–836, Jun 2000. 26

[112] B. Ma and X. Sun. More efficient algorithms for closest string and substring problems. In *Proc. of Research in Computational Molecular Biology (RECOMB 2008)*, volume 4955 of *Lect. Notes Comput. Sc.*, pages 396–409. Springer, 2008. 4

[113] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of Berkeley Symposium on Math. Statist. and Prob.*, pages 281–297, 1967. 29, 57

[114] J. Marialke, R. Körner, S. Tietze, and J. Apostolakis. Graph-based molecular alignment (GMA). *J. Chem. Inf. Model*, 47(2):591–601, 2007. 90

[115] A. Marshall and J. Hodgson. DNA chips: an array of possibilities. *Nat Biotechnol*, 16(1):27–31, Jan 1998. 26

[116] E. C. Meng and R. A. Lewis. Determination of molecular topology and atomic hybridization states from heavy atom coordinates. *J. Comp. Chem.*, 12(7):891–898, 1991. 69

[117] B. T. Messmer and H. Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE T. Knowl. Data En.*, 12(2):307–323, 2000. 90

[118] M. Möhl, S. Will, and R. Backofen. Fixed parameter tractable alignment of RNA structrures including arbitrary pseudoknots. In *Proc. of Combinatorial Pattern Matching (CPM 2008)*, volume 5029 of *Lect. Notes Comput. Sc.*, pages 69–81, 2008. 4

[119] S. Monti, P. Tamayo, J. Mesirov, and T. Golub. Consensus clustering: A resampling-based method for class discovery and visualization of gene expression microarray data. *Mach. Learn.*, 52(1-2):91–118, 2003. 57

[120] R. Motwani. Lecture notes on approximation algorithms. Technical Report STAN-CS-92-1435, Department of Computer Science, Stanford University, 1992. 11

[121] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006. 7, 15, 93

[122] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Inform. Process. Lett.*, 73:125–129, 2000. 22, 46, 47

[123] R. D. M. Page. Modified mincut supertrees. In *Proc. of Workshop on Algorithms in Bioinformatics (WABI 2002)*, volume 2452 of *Lect. Notes Comput. Sc.*, pages 537–552. Springer, 2002. 31

[124] R. D. M. Page and E. C. Holmes. *Molecular Evolution: A Phylogenic Approach: A Phylogenetic Approach.* Wiley Blackwell, 1998. 27

[125] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994. 10

[126] C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991. 75, 76

[127] I. Pe'er, T. Pupko, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. *SIAM J. Comput.*, 33(3):590–607, 2004. 32, 33

[128] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinformatics*, 22(8):974–980, Apr 2006. 90

[129] M. A. Ragan. Phylogenetic inference based on matrix representation of trees. *Mol. Phylogenet. Evol.*, 1(1):53–58, 1992.

[130] S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truss, and S. Böcker. Exact and heuristic algorithms for weighted cluster editing. In *Proc. of Computational Systems Bioinformatics (CSB 2007)*, volume 6, pages 391–401, 2007. 57, 65, 67

[131] S. Rajasekaran, S. Balla, C.-H. Huang, V. Thapar, M. Gryk, M. Maciejewski, and M. Schiller. High-performance exact algorithms for motif search. *J. Clin. Monit. Comput.*, 19(4-5):319–328, Oct 2005. 3

[132] G. Ramsay. DNA chips: state-of-the art. *Nat Biotechnol*, 16(1):40–44, Jan 1998. 26

[133] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *J. Comput. Aided Mol. Des.*, 16(7):521–533, Jul 2002. 90

[134] N. Robertson and P. Seymour. Graph minors: algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986. 17

[135] S. Roch. A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 3(1):92–94, 2006. 3, 29

[136] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, 1987. 29

[137] M. Salemi and A.-M. Vandamme. *The Phylogenetic Handbook*. Cambridge University Press, 2003. 29

[138] E. W. Sayers, T. Barrett, D. A. Benson, E. Bolton, S. H. Bryant, K. Canese, V. Chetvernin, D. M. Church, M. Dicuccio, S. Federhen, M. Feolo, L. Y. Geer, W. Helmberg, Y. Kapustin, D. Landsman, D. J. Lipman, Z. Lu, T. L. Madden, T. Madej, D. R. Maglott, A. Marchler-Bauer, V. Miller, I. Mizrachi, J. Ostell, A. Panchenko, K. D. Pruitt, G. D. Schuler, E. Sequeira, S. T. Sherry, M. Shumway, K. Sirotkin, D. Slotta, A. Souvorov, G. Starchenko, T. A. Tatusova, L. Wagner, Y. Wang, W. J. Wilbur, E. Yaschenko, and J. Ye. Database resources of the national center for biotechnology information. *Nucleic Acids Res.*, 38(Database issue):D5–16, 2010. 88

[139] C. Semple and M. Steel. A supertree method for rooted trees. *Discrete Appl. Math.*, 105(1-3):147–158, 2000. 31

[140] R. Shamir and R. Sharan. Algorithmic approaches to clustering gene expression data. In T. Jiang, T. Smith, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 269–300. MIT Press, 2002. 57, 59

[141] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *J. ACM*, 10:217–255, 1963. 8, 9

[142] R. P. Sheridan, A. Rusinko, R. Nilakantan, and R. Venkataraghavan. Searching for pharmacophores in large coordinate data bases and its use in drug design. *Proc. Natl. Acad. Sci. U S A*, 86(20):8165–8169, Oct 1989. 69

[143] Y. Song, C. Liu, X. Huang, R. L. Malmberg, Y. Xu, and L. Cai. Efficient parameterized algorithms for biopolymer structure-sequence alignment. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 3(4):423–432, 2006. 90

[144] S. Sridhar, K. Dhamdhere, G. Blelloch, E. Halperin, R. Ravi, and R. Schwartz. Algorithms for efficient near-perfect phylogenetic tree reconstruction in theory and practice. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 4(4):561–571, 2007. 29

[145] S. C. Stearns and J. C. Koella. *Evolution in Health and Disease*. Oxford University Press, 2008. 28

[146] C. Steinbeck, Y. Han, S. Kuhn, O. Horlacher, E. Luttmann, and E. Willighagen. The Chemistry Development Kit (CDK): An open-source java library for chemo- and bioinformatics. *J. Chem. Inf. Comp. Sci.*, 43:493–500, 2003. 86

[147] E.-G. Talbi. *Metaheuristics: from design to implementation.* Wiley, 2009. 11

[148] R. L. Tatusov, N. D. Fedorova, J. D. Jackson, A. R. Jacobs, B. Kiryutin, E. V. Koonin, D. M. Krylov, R. Mazumder, S. L. Mekhedov, A. N. Nikolskaya, B. S. Rao, S. Smirnov, A. V. Sverdlov, S. Vasudevan, Y. I. Wolf, J. J. Yin, and D. A. Natale. The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4:41, 2003. 65, 67

[149] D. M. van Aalten, R. Bywater, J. B. Findlay, M. Hendlich, R. W. Hooft, and G. Vriend. PRODRG, a program for generating molecular topologies and unique molecular descriptors from coordinates of small molecules. *J. Comput. Aided Mol. Des.*, 10(3):255–262, Jun 1996. 70

[150] A. van Zuylen and D. P. Williamson. Deterministic algorithms for rank aggregation and other ranking and clustering problems. In *Proc. of Workshop on Approximation and Online Algorithms (WAOA 2007)*, volume 4927 of *Lect. Notes Comput. Sc.*, pages 260–273. Springer, 2008. 58

[151] V. V. Vazirani. *Approximation Algorithms.* Springer, 2001. 12

[152] J. Wang, W. Wang, P. A. Kollmann, and D. A. Case. Automatic atom type and bond type perception in molecular mechanical calculations. *J. Mol. Graph. Model.*, 25:247–260, 2006. iv, vi, 69, 70, 72, 85, 86, 87, 91, 92

[153] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *J. Comput. Biol.*, 1(4):337–348, 1994. 3

[154] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms.* Springer, 2004. 7, 10

[155] T. Wittkop, J. Baumbach, F. Lobo, and S. Rahmann. Large scale clustering of protein sequences with FORCE – a layout based heuristic for weighted cluster editing. *BMC Bioinformatics*, 8(1):396, 2007. 57, 59

[156] T. Wittkop, D. Emig, S. Lange, S. Rahmann, M. Albrecht, J. H. Morris, S. Böcker, J. Stoye, and J. Baumbach. Partitioning biological data with transitivity clustering. *Nat. Methods*, 7(6):419–420, 2010. 57, 58

[157] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization*, pages 185–208, 2001. 12

[158] J. Xu. Rapid protein side-chain packing via tree decomposition. In *Proc. of Research in Computational Molecular Biology (RECOMB 2005)*, volume 3500 of *Lect. Notes Comput. Sc.*, pages 423–439. Springer, 2005. 4

[159] J. Xu, F. Jiao, and B. Berger. A tree-decomposition approach to protein structure prediction. In *Proc. of IEEE Computational Systems Bioinformatics Conference (CSB 2005)*, pages 247–256, 2005. 4, 19

[160] J. Xu, F. Jiao, and B. Berger. A parameterized algorithm for protein structure alignment. In *Proc. of Research in Computational Molecular Biology (RECOMB 2006)*, volume 3909 of *Lect. Notes Comput. Sc.*, pages 488–499. Springer, 2006. 4

[161] Y. Zhao, T. Cheng, and R. Wang. Automatic perception of organic molecules based on essential structural information. *J. Chem. Inf. Model*, 47(4):1379–1385, 2007. 70