View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digitale Bibliothek Thüringen

Algorithm Design Techniques for Parameterized Graph Modification Problems

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.-Inform. Jiong Guo

geboren am 28.11.1970 in Chengdu, V. R. China

Gutachter

1. Prof. Rolf Niedermeier (Friedrich-Schiller-Universität Jena)
2. Prof. Michael R. Fellows (The University of Newcastle, Australien)
3. Prof. Michael A. Langston (University of Tennessee, USA)

Tag der letzten Prüfung des Rigorosums: 9. Feb. 2006

Tag der öffentlichen Verteidigung: 16. Feb. 2006

Erkärung

Hiermit erkläre ich, dass ich die Arbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe.

Jena, April 2005 *Jiong Guo*

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Entwurf parametrisierter Algorithmen für Graphmodifikationsprobleme wie Feedback Vertex Set, Multicut in Trees, Cluster Editing, Closest 3-Leaf Power und Multicommodity Demand Flow in Trees.

Graphen sind heutzutage ein weit verbreitetes und oft verwendetes Werkzeug zur Modellierung komplexer Beziehungen zwischen einzelnen Objekten. Basierend auf solchen Graphmodellierungen können viele in der Praxis auftretende Probleme, die andernfalls nur sehr schwer zugänglich sind, in einer anschaulichen und mathematisch präzisen Form dargestellt werden. Da es sich bei vielen praxisbezogenen Problemen um Fehlerkorrektur, Konfliktvermeidung und Umstrukturierung handelt, sind Graphmodifikationsprobleme ein fundamentaler Bestandteil der algorithmischen Graphentheorie. Grob gesagt haben solche Probleme als Eingabe einen Graphen, für den eine minimale Änderung durch Löschung von Knoten und Kanten und Einfügung von Kanten gesucht ist, damit der geänderte Graph eine bestimmte Eigenschaft hat.

Aufgrund der NP-Vollständigkeit der meisten Graphmodifikationsprobleme stellt die Approximation einer optimalen Lösung den am meisten gebrauchten Lösungsansatz dar. Seit Anfang der 90er Jahre des letzten Jahrhunderts entwickelten sich parametrisierte Algorithmen zu einer sinnvollen und ernstzunehmenden Alternative zu Approximationsalgorithmen beim Lösen NP-harter Graphmodifikationsprobleme. Die Grundidee dieses algorithmischen Ansatzes ist die Identifizierung impliziter und expliziter Problemparameter, auf welche die „kombinatorische Explosion“, die bei der algorithmische Behandlung harter Probleme auftritt, beschränkt werden kann. Sind die Parameter klein, was in vielen praktischen Anwendungen der Fall ist, kann mit einer beweisbar guten Laufzeit des Lösungsalgorithmus gerechnet werden. Damit lassen sich optimale Lösungen harter Probleme oftmals in einem praktisch erträglichen Zeitraum berechnen.

Diese Arbeit untersucht die Anwendbarkeit von vier Techniken zur Entwicklung parametrisierter Algorithmen im Bereich von Graphmodifikationsproblemen. Darunter befinden sich zwei klassische Techniken, nämlich Datenreduktion und tiefenbeschränkte Suchbäume, und zwei neue Techniken, nämlich iterative Kompression und Parametrisierung durch die „Distanz von einer Trivialität“. Jeder der Techniken ist ein eigener Teil dieser Arbeit gewidmet. Im Folgenden, ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge Kanten E. Die Anzahl der Knoten bezeichnen wir mit n ($|V| = n$), die Anzahl der Kanten mit m ($|E| = m$).

Teil 1 enthält eine kurze Einführung in Graphmodifikationsprobleme und eine Übersicht der verwendeten Notationen und Grundbegriffe aus der Graphentheorie. Dieser Teil wird durch eine ausführliche Zusammenfassung von

Ergebnissen dieser Arbeit abgeschlossen.

Teil 2 befasst sich mit der im Jahr 2004 eingeführten Technik der sogenannten iterativen Kompression, die mit einem sehr einfachen Teilgraphen eines gegebenen Graphen anfängt und induktiv immer größerere Teilgraphen aufbaut bis der Eingabegraph erreicht wird. Im Laufe des induktiven Aufbaus wird zunächst eine größere Lösung berechnet und dann durch eine Kompressionsprozedur wieder zu einer kleineren „komprimiert". Kapitel 2 stellt die Grundidee von iterativer Kompression vor und analysiert deren Anwendbarkeit für Graphmodifikationsprobleme. Das eingeführte Entwurfschema für die Anwendung dieser Technik, bestehend aus Iterationsprozedur und Kompressionsprozedur, könnte für künftige Forschungen im Bereich dieser Technik eine gute Basis bilden.

Kapitel 3 befasst sich mit einem klassischen Graphmodifikationsproblem, Feedback Vertex Set (FVS). Dieses fragt für einen gegebenen Graphen nach einer Menge von höchstens k Knoten, deren Entfernung aus dem Graphen alle seine Kreise zerstört. Dieses Problem findet Anwendung in vielen Bereichen wie etwa Bioinformatik und künstlicher Intelligenz. Basierend auf iterativer Kompression zeigen wir hier einen parametrisierten Algorithmus für FVS in Bezug auf den Parameter k, dessen Laufzeit $O(c^k \cdot m^2)$ für eine Konstante c beträgt. Dies erlaubt es, eine lange Zeit offene Frage aus der parametrisierten Komplexitätstheorie zu beantworten.[1]

In Kapitel 4 führen wir zwei allgemeine Beschleunigungsmethoden für iterative Kompression ein. Als ein Beispiel für die Anwendung der ersten Beschleunigungsmethode betrachten wir das Edge Bipartization Problem: für einen gegebenen Graphen wird eine Menge von höchstens k Kanten gesucht, deren Löschung den Graphen in einen bipartiten Graphen transformiert. Wir verbessern die bisher beste Laufzeitschranke für dieses Problem von $O(3^k \cdot k^3 \cdot m^2 \cdot n)$ auf $O(2^k \cdot m^2)$. Mit der zweiten Beschleunigungsmethode erreichen wir einen zweiten Algorithmus für FVS mit einer Laufzeit von $O(c^k \cdot m)$. Dies bedeutet, dass FVS für konstanten Parameter in Linearzeit lösbar ist, eine Eigenschaft die bislang nur für wenige parametrisierte Probleme gezeigt werden konnte.

Das letzte Kapitel von Teil 2 erweitert das Anwendungsgebiet der iterativen Kompression um *alle* minimalen Lösungen für ein Problem berechnen zu können. Das Hauptergebnis ist ein Aufzählungsalgorithmus, der in $O(c^k \cdot m)$ Zeit alle minimalen Lösungen für FVS in einem gegebenen Graphen aufzählt, welche nicht mehr als k Knoten enthalten. Der Algorithmus ist zudem der erste Aufzählungsalgorithmus der auf iterativer Kompression basiert.

Datenreduktion und Problemkerne sind das Thema von Teil 3. Dabei wird versucht, die Eingabeinstanz auf eine neue, kleinere Instanz zu „reduzieren". Die kleinere Instanz wird als Problemkern bezeichnet. Um die Grundidee dieser Technik zu erklären, rekapitulieren wir die klassische „Buss-Datenreduktion", die zu einem Problemkern für Vertex Cover führt. Außerdem zeigen wir in einer Fallstudie einen Problemkern für das Minimum Clique Cover Problem,

[1] Das Ergebnis wurde auch unabhängig von Dehne et al. gezeigt [52].

welcher die parametrisierte Handhabbarkeit dieses Problems impliziert.[2]

Kapitel 7 präsentiert einen Problemkern für das CLUSTER EDITING Problem, das für einen gegebenen Graphen nach einer Menge von höchsten k Kantenmodifikationen sucht, die den Eingabegraph in eine Menge von disjunkten vollständigen Teilgraphen transformieren. Der Problemkern besteht aus $O(k^2)$ Knoten, wobei k die Anzahl der benötigten Kantenmodifikationen ist, und basiert auf zwei Datenreduktionsregeln. Angeregt durch unser Ergebnis untersuchte Damaschke [50] einen „full problem kernel" für die Aufzählungsvariante von CLUSTER EDITING.

Das Hauptresultat von Kapitel 8 ist der Nachweis der Existenz eines Problemkerns für das MULTICUT IN TREES Problem. MULTICUT IN TREES wurde sehr ausführlich im Kontext der Approximationsalgorithmen in der Literatur untersucht. Das Problem kommt im Bereich des Netzwerkentwurfs vor und sucht nach einer Menge von Kanten in einem Baumnetzwerk, deren Löschung vorgegebene Knotenpaare voneinander trennt. Acht einfache und leicht zu implementierende Reduktionsregeln werden gegeben; der Beweis für den Problemkern erweist sich allerdings als technisch aufwändig.

Teil 4 beschäftigt sich mit tiefenbeschränkten Suchbäumen, der wohl am häufigsten angewandten Methode für den Entwurf von parametrisierten Algorithmen. Die entscheidende Idee hierbei ist ein vollständiges Absuchen des Lösungsraumes, das nach einer Baumstruktur systematisch organisiert wird. Bezüglich Graphmodifikationsproblemen konzentrieren wir uns in Kapitel 9 auf ein allgemeines Schema zum Entwurf parametrisierter Algorithmen für die Probleme, die auf eine sogenannte Charakterisierung durch verbotene Teilgraphen zurückzuführen sind. Der Schwerpunkt der Untersuchung liegt auf solchen Charakterisierungen, die unendlich viele verbotene Teilgraphen enthalten. Um dieses Schema für den Entwurf von Suchbaumalgorithmen mit Hilfe verbotener Teilgraphen zu erläutern, geben wir einen einfachen Suchbaumalgorithmus für MULTICUT IN TREES mit einer Laufzeit von $O(2^k \cdot n^2)$ an.

Zwei weitere Suchbaumalgorithmen werden in Kapitel 10 und 11 vorgestellt. Der erste liefert exakte Lösungen für CLUSTER EDITING in $O(2.27^k + n^3)$ Zeit. Dies ist zugleich der erste parametrisierte Algorithmus im Bereich des „Data Clustering". Der zweite Algorithmus löst das aus einer bioinformatischen Anwendung hervorgehende CLOSEST 3-LEAF POWER Problem, welches einen phylogenetischen Baum mit Distanzinformationen aus einem Netzwerk von Spezies zu rekonstruieren versucht. Der Lösungsansatz ist ein tiefenbeschränkter Suchbaum, der auf einer Charakterisierung mittels verbotener Teilgraphen von Graphen, die *3-leaf powers* sind, basiert. Diese Charakterisierung enthält unendlich viele verbotene Teilgraphen und wird in Kapitel 11 gezeigt. Um diese unendlich vielen verbotenen Teilgraphen zu behandeln wird ein *critical clique graph* eingeführt, auf dem der Suchbaumalgorithmus arbeitet. Der Beweis einer Eins-zu-Eins-Korrespondenz zwischen der Lösung für den critical clique graph und der Lösung für den Eingabegraphen ist der technisch anspruchsvollste Teil dieses Kapitels.

[2]Die parametrisierte Komplexität von MINIMUM CLIQUE COVER war bislang offen.

Die letzte Technik, die Parametrisierung durch die „Distanz von einer Trivialität“, wird in Teil 5 diskutiert. Dabei wird zuerst für ein Problem ein „trivial“ zu lösender Fall von Instanzen ausgewählt und dann wird ein Parameter identifiziert, der die „Distanz“ allgemeiner Instanzen zu den speziellen Instanzen misst. Bezüglich dieser Distanz wird ein parametrisierter Algorithmus für allgemeine Instanzen entworfen. Diese neue Art der Parametrisierung ist besonders hilfreich wenn wir mit einem Problem zu tun haben, das unter anderen Parametrisierungen – wie beispielsweise mit der Größe der Lösung als Parameter – als W[1]-hart erwiesen ist.

In Kapitel 13 betrachten wir das Multicommodity Demand Flow in Trees (MDFT) Problem das versucht, in einem Kommunikationsnetz möglichst viele profitable peer-to-peer Kommunikationen zu erlauben, ohne dabei die Kapazität der Kommunikationsverbindungen zu überschreiten. Kapitel 13 liefert ein neues Ergebnis bezüglich der exakten Lösbarkeit von MDFT. Wir identifizieren einen neuen Parameter k, der die maximale Anzahl von peer-to-peer Kommunikationen über einen Knoten darstellt. Damit erzielen wir einen $O(2^k \cdot n^2)$-Zeit Algorithmus, welcher der erste exakte Algorithmus von praktischer Relevanz für MDFT sein könnte.

Der zweite Algorithmus in Teil 3 resultiert aus der Betrachtung des Tree-Like Weighted Set Cover (TWSC) Problems, das aus Anwendungen bei der Berechnung von Baumzerlegungen [22] und in der Phylogenomics [148] stammt. Das Problem wird formal in Kapitel 14 definiert und stellt einen Spezialfall des klassischen Set Cover Problems dar, wobei sich die Teilmengen in der Sammlung von Mengen in einem Baum organisieren lassen können und zudem eine Konsistenzeigenschaft erfüllen müssen. Bezeichnen wir die maximale Größe der Teilmengen mit k, so läuft unser Algorithmus für TWSC in $O(3^k \cdot n^2)$ Zeit. Zusammen mit einer parametrisierten Reduktion von TWSC auf Multicut in Trees zeigen wir, dass, parametrisiert durch eine „vertex cutwidth“ k, Multicut in Trees in $O(3^k \cdot n^2)$ Zeit lösbar ist.

Am Ende der Arbeit fassen wir die erzielten Resultate nochmals zusammen und es werden einige Fragestellungen für zukünftige Forschung aufgeworfen.

Preface

This thesis summarizes a significant part of my study and research on parameterized complexity, particularly focusing on the design of efficient fixed-parameter algorithms for graph modification problems. From March 2002 to February 2003 I was supported by the Zentrum für Bioinformatik in Tübingen and, since March 2003, my research is supported by the Deutsche Forschungsgemeinschaft (DFG) under the project "Fixed-Parameter Algorithms (PIAF)", NI 369/4. I owe sincere thanks to this support and to Klaus-Jörn Lange and Rolf Niedermeier for giving me the research opportunities. In particular, I deeply thank my supervisor Rolf Niedermeier who initiated the PIAF-project and has been my guide and mentor for the past four years. I am in great debt to all members in our working group: Jens Gramm who shared with me these four years from the beginning and with whom I have worked closely, Jochen Alber who gave me advice concerning graph theory; Michael Dom, Falk Hüffner, and Sebastian Wernicke with whom I have had various stimulating discussions.

This thesis is based on my various research collaborations together with one or another above mentioned members of our working group. The most important partner in my research has been Rolf Niedermeier. In this thesis I present only results which are closely related to fixed-parameter algorithms for graph modification problems and to whose achievement I made significant contributions. Further results concern sequence analysis [11, 90, 89], matrix property and matrix modification [62, 178], NP-completeness of graph problems [59, 61], parameterized complexity of graph problems [60, 58, 94, 99, 100], graph theory [100], and algorithm engineering [85, 87, 88].

There are six parts in this thesis. After a brief "Introduction" (Part I), four design techniques for fixed-parameter algorithms are presented, namely "Iterative Compression" (Part II), "Data Reduction and Problem Kernels" (Part III), "Search Trees Based on Forbidden Subgraphs" (Part IV), and "Parameterization by Structure" (Part V). Finally, a short "Conclusion" (Part VI) summarizes the results. The first chapter of each of Parts II to V is an introduction to the technique considered in the respective part. In each of these parts the further chapters following the introductory chapter contain the new results. In the following, I also briefly sketch my contributions in this collaborative studies.

Part II is mainly based on [93]. As case studies for the iterative compression technique, fixed-parameter algorithms for Feedback Vertex Set and Edge Bipartization are given. I have initiated the study of these two problems and the major parts of the technically involved proofs of the correctness and runtimes of the algorithms were carried out by myself. The enumeration algorithm for Feedback Vertex Set resulted from a discussion with Jens Gramm.

Part III contains two main new results, the problem kernels for Cluster Editing and Multicut in Trees. Concerning Cluster Editing my contri-

bution was to develop the data reduction rules. Moreover the analysis of the problem kernel given in this thesis is slightly different from the one given in [86]. The problem kernel for Multicut in Trees is based on [97]. I provided the mathematically involved analysis of the problem kernel.

In Chapter 10 of Part IV the key difficulty for achieving the improved branching strategy for Cluster Editing lies in proving a better branching for one of the three cases. I came up with the proof of a 2-branching for this case that finally led to the improved search tree algorithm which is based on [86]. I have initiated the study of Closest 3-Leaf Power (Chapter 11 in Part IV) which is based on [59]. The basic idea for the search tree algorithm, that is, considering forbidden subgraphs, was due to me. My further achievements here include the proof of the correctness of working on critical clique graphs instead of working on the original graphs.

The first result of Part V is a theoretical framework for uncovering structural parameters which is based on [95]. My main contribution here was to develop several case studies for this framework, among others, the two fixed-parameter algorithms given in Chapters 13 and 14. These two algorithms were published in [96, 98]. Here I want to thank our students Natja Betzler and Johannes Uhlmann for initiating the research on the Tree-like Weighted Set Cover problem.

Contents

I Introduction 1

1 Introduction 3

1.1 Graph Modification Problems . . . 3
1.2 Fixed-Parameter Algorithms . . . 5
1.3 Preliminaries . . . 6
1.4 Summary of Results . . . 8

II Iterative Compression 11

2 Basic Concepts and Ideas 13

2.1 Iteration . . . 14
2.1.1 Case Study 1: Vertex Cover . . . 15
2.1.2 Case Study 2: Multicut in Trees . . . 16
2.1.3 Case Study 3: Cluster Deletion . . . 17
2.2 Compression . . . 17
2.2.1 Case Study 1: Vertex Cover . . . 18
2.2.2 Case Study 2: Cluster Deletion . . . 19
2.3 Concluding Remarks . . . 19

3 Feedback Vertex Set 21

3.1 Problem Definition and Previous Results . . . 21
3.2 The Algorithm . . . 22
3.2.1 Iteration . . . 22
3.2.2 Compression . . . 22
3.3 Concluding Remarks . . . 25

4 Speed-up Methods 27

4.1 Compression without Partition . . . 27
4.2 Constant-Factor Approximation Instead of Iteration . . . 31
4.3 Concluding Remarks . . . 32

5 Compression-Based Enumeration 33
5.1 Feedback Vertex Set 34
5.2 Concluding Remarks 36

III Data Reduction and Problem Kernels 37

6 Basic Concepts and Ideas 39
6.1 Data Reduction 39
6.1.1 Case Study 1: Feedback Vertex Set 41
6.1.2 Case Study 2: Vertex Cover 42
6.1.3 Case Study 3: Minimum Clique Cover 42
6.2 Problem Kernel 43
6.2.1 Case Study 1: Vertex Cover 44
6.2.2 Case Study 2: Minimum Clique Cover 44
6.3 Concluding Remarks 45

7 Cluster Editing 47
7.1 Problem Definition and Previous Results 47
7.2 Data Reduction Rules 48
7.3 Problem Kernel 50
7.4 Concluding Remarks 51

8 Multicut in Trees 53
8.1 Problem Definition and Previous Results 53
8.2 Parameter-Independent Reduction Rules 54
8.3 Parameter-Dependent Reduction Rules 55
8.3.1 Some Notation and Definitions 55
8.3.2 Parameter-Dependent Data Reduction Rules 57
8.4 Some Observations on Reduced Instances 58
8.5 Problem Kernel 61
8.5.1 Problem Kernel for Caterpillars 61
8.5.2 Problem Kernel for Spiders of Caterpillars 66
8.5.3 Problem Kernel for General Trees 68
8.6 Concluding Remarks 70

IV Search Trees Based on Forbidden Subgraphs 73

9 Basic Concepts and Ideas 75
9.1 Forbidden Subgraph Characterizations 76
9.2 Depth-Bounded Search Trees 77
9.3 Search Trees Based on Forbidden Subgraphs 79
9.4 Two Case Studies 80
9.4.1 Case Study 1: Vertex Cover 81
9.4.2 Case Study 2: Multicut in Trees 82

9.5 Concluding Remarks 83

10 Cluster Editing 85
10.1 Basic Branching Strategy 85
10.2 Refined Branching Strategy 87
10.3 Concluding Remarks 92

11 Closest 3-Leaf Power 93
11.1 Problem Definition and Previous Results 93
11.2 Forbidden Subgraph Characterization for 3-Leaf Powers 96
11.3 Algorithms 99
11.3.1 Edge Modifications (Overview) 100
11.3.2 Vertex Deletion 105
11.4 Concluding Remarks 106

V Parameterization by Structure 107

12 Basic Concepts and Ideas 109
12.1 Distance From Triviality 109
12.2 Case Study 1: Clique 112
12.3 Case Study 2: Power Dominating Set 113
12.4 Concluding Remarks 116

13 Multicommodity Demand Flow in Trees 119
13.1 Problem Definition and Previous Results 119
13.2 The Algorithm 121
13.2.1 Agreements and Basic Tools 121
13.2.2 Dynamic Programming Algorithm 122
13.2.3 Main Result 124
13.3 Concluding Remarks 125

14 Weighted Multicut in Trees 127
14.1 Multicut in Trees and Tree-Like Set Cover 128
14.1.1 Tree-Like Weighted Set Cover (TWSC) 128
14.1.2 Weighted Multicut in Trees and TWSC 132
14.2 Algorithm for TWSC 133
14.2.1 TWSC with Binary Subset Tree 133
14.2.2 TWSC with Arbitrary Subset Tree 136
14.3 Concluding Remarks 136

VI Conclusion 139

15 Conclusion 141

Part I

Introduction

Chapter 1

Introduction

In the first section we give an introduction to graph modification problems and motivate the study of such problems. The second section is a brief introduction to fixed-parameter algorithms. Some basic notation used throughout this thesis will be given in the third section. Finally, we close this chapter with a summary of the results of this thesis.

1.1 Graph Modification Problems

In the last decades, graph models have played a key role in the development of computer science [4, 103, 111], biology [29, 156], social sciences [34, 156, 157], economics [16, 136], and many other scientific disciplines. Using vertices and edges representing entities and the interrelation between the entities, respectively, real-life problems can be formulated in a simple and precise way. From the beginning of the introduction of graph models, researchers have formulated many *graph modification problems*, inspired by practical applications dealing with error correction, conflict resolution, and reconfiguration. In general, graph modification problems can be defined as follows:

> **Input**: A graph and a desired property Π.
> **Task**: Make a minimum number of modifications to the given graph such that Π is fulfilled.

Possible modifications include deletions of vertices and edges and insertions of edges. In the above definition, the optimization goal is to minimize the number of modifications to be made since modifications in real-world networks are cost-intensive. Thus, graph modification problems belong to the class of minimization problems.

Motivation. Graph modification problems have applications in many fields such as molecular biology, machine learning, and operations research. Let us describe two examples for these applications.

The CLUSTER EDITING problem (see Chapter 7 for a formal definition) seeks for a minimum number of edge modifications which transform a given graph into a union of disjoint *complete* graphs. A graph is called complete if it has an edge between each pair of vertices. This problem is partly motivated by applications in machine learning. One of these applications deals with clustering entity names where, given some entries which are taken from multiple databases (e.g., names/affiliations of researchers), the goal is to collect together the entries that correspond to the same entity (person). For instance, in the case of researchers, the same person might appear multiple times with different affiliations. By representing the entries as vertices, a classifier specifies an edge between two entries if it believes that the entries represent the same person. After making a minimum number of edge modifications, each of the disjoint complete graphs in the resulting graph gives a collection of entries that might correspond to the same entity. More details can be found in [17].

One problem studied in this thesis seeks to break all cycles in a graph by removing a minimum number of vertices, the so-called FEEDBACK VERTEX SET problem (see Chapter 13). This problem finds—among others—applications in *genetic linkage analysis*. Genetic linkage is the phenomenon where alleles of different genes appear to be genetically coupled. Analysis of linkage is important for, e.g., genetic counseling and estimating changes in population genetics. A good treatment of this subject can be found in [146]. Applying Bayesian networks to genetic linkage analysis, "pedigree cycles" pose a difficult computational challenge. Finding a minimum size set of vertices whose removal destroys the pedigree cycles is crucial for the probabilistic inference in the Bayesian networks. An approximation algorithm and a randomized algorithm for FEEDBACK VERTEX SET were implemented in a genetic linkage analysis software package, FASTLINK [19, 20]. For more details on this application we refer to [18, 71].

Previous Complexity Results. The study of the complexity of graph modification problems can be dated back to 1970's. Garey and Johnson [80] list 18 NP-complete graph modification problems. Lewis and Yannakakis [128] showed that, for any non-trivial hereditary graph property Π, the corresponding graph modification problem with only vertex deletion allowed is NP-complete. Yannakakis [181] gave a characterization of those graph properties for which the vertex deletion problems are polynomial-time solvable in bipartite graphs and a characterization of those for which the vertex deletion problems remain NP-complete.

Concerning edge modification problems, there is no such general characterization of graph properties for which the corresponding edge modification problems are NP-complete as in the case of vertex deletion problems.[1] Yannakakis [180] showed the NP-completeness of edge modification problems with respect to seven graph properties including bipartite and outerplanar. We refer to [137] for an overview of the complexity status of edge modification problems for some graph classes.

[1] Two relatively general characterizations can be found in [67, 13].

1.2 Fixed-Parameter Algorithms

Since most graph modification problems are NP-complete, it seems hopeless to optimally solve them in polynomial time. However, as mentioned above, graph modification problems belong to the class of minimization problems and the sizes of the optimal solutions for these problems are expected to be small: a big solution size means a high cost and, thus, often has to be considered as an infeasible strategy in practical applications. Therefore, a natural approach concerning the exact solvability of these problems is to try to restrict the seemingly unavoidable combinatorial explosion to a—hopefully small—parameter, which results in efficient algorithms in case of small parameter values. Such algorithms are called *fixed-parameter algorithms*.

We begin with some basic definitions of parameterized complexity theory pioneered by Downey and Fellows [65].

Definition 1.1. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the* parameter *of the problem.*

Throughout this thesis the parameter is a nonnegative integer. Hence we will usually write $L \subseteq \Sigma^* \times \mathbb{N}$ instead of $L \subseteq \Sigma^* \times \Sigma^*$. For $(x, k) \in L$, the two dimensions of parameterized complexity analysis are constituted by the input size n, that is, $n := |(x, k)|$, and the parameter value k (usually a nonnegative integer). We give now a parameterized definition of graph modification problems:

> **Input**: A graph and a desired property Π and an integer $k \geq 0$.
> **Task**: Make at most k modifications to the given graph such that Π is fulfilled.

Fixed-parameter tractability as defined below is the key notion in this thesis.

Definition 1.2. *A parameterized problem L is* fixed-parameter tractable *if there exists an algorithm that decides in $f(k) \cdot n^{O(1)}$ time whether $(x, k) \in L$, where f is a computable function only depending on k. This algorithm is called a* fixed-parameter algorithm. *The complexity class containing all fixed-parameter tractable problems is called* FPT.

Note that the runtime of a fixed-parameter algorithm is polynomial in the instance size and the exponential part of the runtime is only dependent on parameter k. Thus, for small value of k (as expected in many practical applications), $f(k)$ is reasonably small, which means an efficient algorithm for the given problem.

In Definition 1.2, one may assume any standard model of sequential deterministic computation such as deterministic Turing machines or RAMs. For the sake of convenience, if not stated otherwise, we will always take the parameter, denoted by k, as a nonnegative integer encoded with a unary alphabet.

Not every parameterized problem is fixed-parameter tractable. Analogously to classical complexity theory, Downey and Fellows developed a reducibility and

completeness program. The completeness theory of parameterized intractability involves significantly more technical effort than the classical one. We very briefly sketch some integral parts of this theory in the following.

We first need a reducibility concept:

Definition 1.3. *Let $L_1, L_2 \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. We say that L_1* reduces to L_2 by a standard parameterized reduction *if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathbb{N}$ to $\mathbb{N}$ and a function $(x,k) \mapsto x'$ from $\Sigma^* \times \mathbb{N}$ to Σ^* such that*

1. *$(x,k) \mapsto x'$ is computable in $k'' \cdot |(x,k)|^c$ time for some constant c and*
2. *$(x,k) \in L_1$ iff $(x',k') \in L_2$.*

Notably, most reductions from classical complexity turn out *not* to be parameterized ones. The "lowest class of parameterized intractability" can be defined as the class of parameterized languages that are equivalent to the so-called SHORT TURING MACHINE ACCEPTANCE problem (also known as the k-STEP HALTING problem). Here, we want to determine whether a given Turing machine accepts a given word in a computation of at most k steps. Together with the above introduced reducibility concept, SHORT TURING MACHINE ACCEPTANCE can now be used to define the lowest class of parameterized intractability, that is, W[1].

The fundamental conjecture that FPT≠W[1] is very much analogous (but clearly "weaker") to the conjecture that P≠NP. From a more algorithmic point of view, it is usually sufficient to distinguish between W[1]-hardness and membership in FPT. Thus, for an algorithm designer not being able to show fixed-parameter tractability of a problem, it may be "sufficient" to give a parameterized reduction from a W[1]-hard problem to the given problem. This then proves that, unless FPT=W[1], the problem does not allow for an $f(k) \cdot n^{O(1)}$ time algorithm. One piece of circumstantial evidence for this unlikeliness is the result showing that FPT=W[1] would imply a $2^{o(n)}$ time algorithm for the NP-complete 3-SATISFIABILITY problem (where n denotes the number of variables of the given Boolean formula) [1], which would mean a major (and so far considered unlikely) breakthrough in computational complexity theory.

For further details on the field of parameterized complexity, the reader is referred to [63, 65, 75, 140, 141].

1.3 Preliminaries

We use the following notion throughout this thesis.

Graph theory. An *undirected* graph G with n vertices and m edges is a pair (V, E) where V is a finite set of *vertices* and E is a finite set of *edges*. Edges are defined as *unordered* pairs of vertices called *endpoints*.[2] We denote

[2] All graphs considered in this thesis are undirected. Ordered pairs of vertices yield *directed edges* and, thus, *directed* graphs.

an edge e with vertices u and v as endpoints as $e = \{u, v\}$. The edge e is *incident* to u and to v and vertices u and v are *adjacent*.

A *loop* is an edge whose endpoints are equal. *Parallel edges* or *multiple edges* are edges that have the same pair of endpoints. We consider here only graphs without loops or multiple edges.

The *degree* of a vertex is the number of edges incident to it. The (*open*) *neighborhood* of a vertex v in graph $G = (V, E)$ is defined as

$$N(v) := \{\, u \mid \{u, v\} \in E \,\}.$$

The *closed neighborhood* $N[v]$ of v then is $N(v) \cup \{v\}$. Finally, for a vertex set $U \subseteq V$ we define $N(U) := \bigcup_{v \in U} N(v)$ and $N[U] := \bigcup_{v \in U} N[v]$.

A *path* in a graph is a sequence of pairwise distinct vertices so that subsequent vertices are adjacent in the graph. The first and last vertices of a path are call the *endpoints* of the path. If the two endpoints of a path containing at least three vertices are adjacent, then we have a *cycle*. The *length* of a path (or a cycle) is the number of edges in the sequence. The *distance* between two vertices u and v in a graph G, denoted by $d_G(u, v)$ (or simply $d(u, v)$) is the least length of a path with u and v as endpoints. An edge between two vertices of a cycle that is not part of the cycle is called *chord*. An induced, chordless cycle of length at least four is called *hole*.

A graph is *connected* if every pair of vertices is connected by a path. If a graph is not connected then it naturally decomposes into its *connected components*. A graph having no cycle is *acyclic*. A *forest* is an acyclic graph; a *tree* is a connected acyclic graph. A *leaf* is a vertex of degree one. Sometimes we deal with *rooted* trees. They arise when choosing one distinguished vertex (the *root*) and directing all edges from the root to its neighbors—then called its *children*—analogously directing all edges between the root's children and their neighbors different from the root, and so on.

For a graph $G = (V, E)$ and a set $V' \subseteq V$, the *subgraph* of G *induced* by V' is denoted by $G[V'] = (V', E')$, where $E' := \{\, \{u, v\} \in E \mid (u \in V') \wedge (v \in V') \,\}$. By way of contrast, a *subgraph* $H = (V'', E'')$ of $G = (V, E)$ simply fulfills that $V'' \subseteq V$, $E'' \subseteq E$, and the endpoints of edges in E'' have to be contained in V''. A *spanning subgraph* of G is a subgraph with vertex set V. A *spanning tree* is a spanning subgraph that is a tree. The *complement* of a graph $G = (V, E)$ is the graph with the same vertex set V and an edge set containing $\{u, v\}$ iff $\{u, v\} \notin E$.

We occasionally use $G \setminus V'$ for $V' \subseteq V$ to denote the graph resulting by deleting the vertices in V' and their incident edges from G, i.e., $G \setminus V' := G[V \setminus V']$. Similarly, $G \setminus E'$ for $E' \subseteq E$ denotes the graph resulting by deleting the edges in E' from G, i.e. $G \setminus E' := (V, E \setminus E')$. Moreover, when both edge deletion and insertion allowed, $G' = (V, E \ominus E')$ denotes the graph resulting by deleting the edges in $E \cap E'$ from G and inserting the edges in $E' \setminus E$ into G, where $\ominus$ denotes the *symmetric difference operation* between two sets.

In this work we deal with some special classes of graphs. A *complete graph* or *clique* is a graph where every two vertices have an edge between them. An

independent set in a graph G is a vertex subset I with $G[I]$ containing no edge of G. A *chordal graph* is a graph that contains no hole. A graph $G = (V, E)$ is *bipartite* if V is the union of two disjoint sets such that each edge consists of one vertex from each set.

For a comprehensive overview on the general graph theory we refer to [55, 111, 179]. An excellent overview on graph classes can be found in [30].

Approximation. *Polynomial-time approximation algorithms* strive for approximate instead of optimal solutions for optimization problems. Here we restrict to minimization problems. The quality of the approximation (with respect to the given optimization criterion) is measured by the *approximation factor*. An approximation algorithm for a minimization problem has approximation factor r if for any input size n, the cost C of the solution produced by the approximation algorithm is within a factor of $r \geq 1$ of the cost C_{opt} of an optimal solution: $C/C_{\text{opt}} \leq r$. A *polynomial-time approximation scheme (PTAS)* for a minimization problem is an algorithm which, for each $\epsilon > 0$ and each problem instance, returns a solution with approximation factor $1 + \epsilon$. The polynomial runtime of such an algorithm, as a rule, crucially depends on $1/\epsilon$. We mention in passing that a class MaxSNP of optimization problems can be "syntactically" defined together with a reducibility concept [149]. The point is that *MaxSNP-hard* problems are unlikely to have polynomial-time approximation schemes.

More details on approximation theory can be found, e.g., in [14, 172].

Big O notation. We use the familiar "*big O notation*" to upperbound the runtimes of our algorithms. Thus, we ignore constant factors but, if appropriate, we point to cases where the involved constant factors of the algorithms are large and, thus, might threaten or destroy the practical usefulness of the algorithms. For functions $f(n)$ and $g(n)$, $f(n)$ is in $O(g(n))$ iff there are a constant c and an integer n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

1.4 Summary of Results

This thesis aims at a systematical study of the applicability of several techniques for designing fixed-parameter algorithms for graph modification problems. The main results with respect to six graph modification problems, namely FEEDBACK VERTEX SET, EDGE BIPARTIZATION, CLUSTER EDITING, (WEIGHTED) MULTICUT IN TREES, CLOSEST 3-LEAF POWER, and MULTICOMMODITY DEMAND FLOW IN TREES, are summarized in the following. Here we always use n and m to denote the number of the graph vertices and the graph edges, respectively.

Iterative compression. In Chapter 2 we study the iterative compression technique [155] and provide a thorough analysis of its two components, namely, iteration and compression. With three easily understandable case studies, we

hope to convince the reader that this new technique has the potential to be an important tool for designing efficient fixed-parameter algorithms for graph modification problems

The main result of Chapter 3 is a fixed-parameter algorithm based on the iterative compression technique which solves Feedback Vertex Set in $O(c^k \cdot m \cdot n)$ time. Here c is a constant ($c \approx 38$) and k denotes the size of the desired feedback vertex set.[3] This result affirmly answers an open question from previous work and demonstrates that iterative compression seems useful for attacking unsettled fixed-parameter complexity questions.

Two speed-up methods for iterative compression, compression without partition and polynomial-time constant-factor approximation instead of iteration, are suggested in Chapter 4. As two example applications of these methods, we present two fixed-parameter algorithms for Edge Bipartization and Feedback Vertex Set. The runtime of the algorithm for Edge Bipartization is $O(2^k \cdot m^2)$ which improves an $O(3^k \cdot k^3 \cdot m^2 \cdot n)$ time algorithm by Reed et al. [155]. Here k denotes the size of the edge bipartization set. Applying a factor-four linear-time approximation algorithm, a second fixed-parameter algorithm for Feedback Vertex Set runs in $O(c^k \cdot m)$ time where c is a constant (much greater than 38) and k is the size of the feedback vertex set. This implies that Feedback Vertex Set is "linear-time fixed-parameter tractable" in the sense that the input size is measured by the number of graph edges.

In Chapter 5 we present a parameterized enumeration algorithm for Feedback Vertex Set running in the same time as the algorithm given in Chapter 4. This algorithm seems to be the first parameterized enumeration algorithm based on iterative compression and opens a new application field for this technique.

Data reduction and problem kernels. Data reduction and the notion of problem kernels perhaps are the most important contribution of parameterized complexity to practical computing. In Chapter 6, after a brief introduction, we revisit the classical data reduction rules for Feedback Vertex Set and Vertex Cover. As a case study, we show a problem kernel for the Minimum Clique Cover problem which is the first result with respect to the parameterized complexity of this problem.

In Chapter 7 a problem kernel for Cluster Editing which consists of $O(k^2)$ vertices is achieved where k denotes the required number of edge modifications. This seems to be the first result with respect to kernelization in the field of clustering problems. Inspired by our kernelization, Damaschke [50] investigated so-called *full* kernels for the enumeration version of Cluster Editing.

The second main result in this part is a problem kernel for Multicut in Trees, a problem well-studied in terms of polynomial-time approximabilities. The proof of the existence of a kernel involving complicated combinatorial arguments is provided in Chapter 8.

[3] Independently, this result was also shown by Dehne et al. [52], improving the constant to 10.6.

Search trees based on forbidden subgraphs. The construction of depth-bounded search trees based on induced forbidden subgraphs is probably the most commonly used technique in the design of fixed-parameter algorithms for graph modification problems [32]. In Chapter 9 we discuss the applicability of this technique with the emphasis on the distinction between finite and infinite forbidden subgraph characterizations. As a case study, we give a simple algorithm solving MULTICUT IN TREES in $(2^k \cdot n^2)$ time with k denoting the number of the edge deletions.

Following the algorithm design schemes suggested in Chapter 9, we present two search tree algorithms for CLUSTER EDITING (Chapter 10) and CLOSEST 3-LEAF POWER (Chapter 11). The former problem has a finite forbidden subgraph characterization and the latter problem an infinite one. Based on the known forbidden subgraph P_3 of cluster graphs, the basic branching strategy leads to a size-3^k search tree for CLUSTER EDITING with the number of edge modifications as parameter k. In Chapter 10 we demonstrate how to shrink the search tree size by a careful case distinction, resulting in an $O(2.27^k + n^3)$ time algorithm. In Chapter 11, to construct a search tree algorithm for CLOSEST 3-LEAF POWER, we first derive an infinite forbidden subgraph characterization of graphs being 3-leaf powers.[4] Then, by combining two depth-bounded search tree algorithms for two subtasks, each subtask getting rid of a subset of the forbidden subgraphs, we establish a fixed-parameter tractability result of CLOSEST 3-LEAF POWER where the parameter is the number of edge modifications.

Parameterization by structure. Finally, we propose the "distance from triviality" measurement as a useful structural problem parameterization and discuss a framework for uncovering such parameters in analyzing computationally hard problems. The case studies given in Chapter 12 shall exhibit the versatility of this approach to gain important new views for computational complexity analysis.

Two main problems are considered in this framework, MULTICOMMODITY DEMAND FLOW IN TREES and WEIGHTED MULTICUT IN TREES. For both problems we propose a "vertex cutwidth parameter" whose value can be assumed to be small in several applications. With k denoting this vertex cutwidth parameter we show that MULTICOMMODITY DEMAND FLOW IN TREES and WEIGHTED MULTICUT IN TREES can be solved in $O(2^k \cdot n^2)$ and $O(3^k \cdot n^2)$ time, respectively. While both problems have been intensively studied with respect to approximation algorithms, these results seem to be the first results concerning their parameterized complexity. Moreover, both algorithms are conceptually simple enough to allow easy implementation and may be profitable alternatives to existing approximation algorithms. The achieved algorithm for WEIGHTED MULTICUT IN TREES solves actually a more general problem, TREE-LIKE WEIGHTED SET COVER, which is motivated by applications in tree decomposition based computing [22] and phylogenomics [148].

[4] This characterization is crucial for the NP-completeness proof of a variant of CLOSEST ℓ-LEAF POWER for $\ell \geq 3$ [59].

Part II

Iterative Compression

Chapter 2

Basic Concepts and Ideas

In 2004, Reed et al. [155] introduced a simple but elegant technique, the so-called "iterative compression" technique, which appears to have the potential to be a general tool for designing efficient fixed-parameter algorithms for minimization problems with the solution size as the parameter. In their ground-breaking paper [155] they demonstrated how powerful this technique is by giving an efficient fixed-parameter algorithm for the Graph Bipartization problem. This has been a central open problem in parameterized complexity for several years. Graph Bipartization is defined as follows:

> **Input**: A undirected graph $G = (V, E)$ and an integer $k \geq 0$.
>
> **Task**: Find a *vertex bipartization set* V' with at most k vertices whose removal transforms the input graph into a bipartite graph.

The central part of their algorithm is a *compression procedure* which, given an n-vertex graph G and a size-$(k+1)$ solution of Graph Bipartization on G, either constructs a size-k solution or proves that there is no size-k solution. This compression procedure runs in $O(3^k \cdot k \cdot m)$ with $m := |E|$. Based on this compression procedure, the algorithm starts with the trivial subinstance consisting of the subgraph $G[\{v_1\}]$ and parameter k. The optimal vertex bipartization set for this subinstance is clearly the empty set. Then, the algorithm iteratively considers for $i = 2, \ldots, n$ the subinstances consisting of the subgraphs $B_i := G[\{v_1, \ldots, v_i\}]$ and parameter k. Let V'_{i-1} denote the vertex bipartization set of G_{i-1} with $|V'_{i-1}| \leq k$. Obviously, $V'_i := V'_{i-1} \cup \{v_i\}$ is a vertex bipartization set of G_i. If $|V_i| \leq k$, then the algorithm proceeds with G_{i+1}; otherwise, it calls the compression procedure with G_i and the size-$(k+1)$ solution V'_i as inputs to "compress" V'_i. Note that, if the compression procedure fails to compress V'_i, then G_i has no size-k vertex bipartization set and the input graph G has no size-k vertex bipartization set as well. In summary, the overall runtime of this algorithm is $O(3^k \cdot k \cdot m \cdot n)$.

Roughly speaking, the basic idea of the iterative compression technique is as follows: To show that a minimization problem is fixed-parameter tractable

with respect to the solution size, it suffices to give a fixed-parameter algorithm which, given a size-$(k+1)$ solution, shows that either there is no size-k solution or constructs one. As shown in the Graph bipartization example, the iterative compression technique employs two procedures. One procedure is the compression procedure which constructs a size-k solution from the given size-$(k+1)$ solution and the other is called the "iteration" procedure. It provides the compression procedure with the desired size-$(k+1)$ solution. The iteration procedure is the main procedure which calls the compression procedure as a subroutine. In this chapter we discuss both procedures in detail. Then, in Chapter 3, we present a fixed-parameter algorithm based on this technique solving the Feedback Vertex Set problem. In Chapter 4, we present two general methods which can improve the runtimes of iterative compression algorithms in several cases. Finally, we show in Chapter 5 how to use the iterative compression technique to design an enumeration algorithm for Feedback Vertex Set.

2.1 Iteration

As already demonstrated in the Graph Bipartization example, in order to feed the compression procedure with a size-$(k+1)$ solution, the iteration procedure starts with a trivially solvable, small subinstance of the given instance—a one-vertex subgraph—and inductively applies the compression procedure a linear number of rounds to larger and larger subinstances. We now extend this scheme of the iteration procedure to general minimization problems. Three case studies will be given later.

For a given instance I of a minimization problem consisting of a graph G and parameter k, the iteration procedure works as follows:

1. Set I' to a trivially solvable subinstance of I and compute a size at most k solution S for I';
2. While $I' \neq I$ do
   (a) Move one step forward to I by "augmenting" I' with an additional "element" which is in I but not in I';
   (b) Based on S, compute a solution S' for the new I';
   (c) If $|S'| > k$,
      i. then call the compression procedure with I' and S' as inputs to compress—if possible— S' to a size-k solution. Set S equal to the size-k solution returned by the compression procedure; (For more details on the compression procedure see Section 2.2.)
      ii. Otherwise, $S := S'$;
3. Return S as the solution of the minimization problem on the input instance I.

Observe that, although the iteration procedure is very simple, its applicability and correctness heavily depend on the choice of the element added to I' to get the new instance. We call the element the *augmentation element*. For many problems, the choice of the augmentation element is not uniquely determined. We can add a vertex (as in the iteration procedure for Graph Bipartization) or an edge (as we do for Edge Bipartization in Section 4.1) in each augmentation step, which seem to be the most natural and simple ways to augment the subinstances dealing with graph problems. Some graph problems carry additional inputs which can also be promising candidates for the augmentation element. For example, many network connectivity problems examine the reliability of the communication between some given pairs of network vertices. These given vertex pairs can also be used as augmentation element, as we will demonstrate in the Multicut in Trees case study (Section 2.1.2).

For determining the augmentation element, there are two important criteria:

- **(C1)** The computation of a solution of size at most $k+1$ for the new, augmented subinstance, based on the size-(k) solution of the old, smaller subinstance should be "easy": In the following case studies, this computation works in polynomial time for most problems with proper augmentation elements.
- **(C2)** The optimal solutions of the subinstances should exhibit a *monotonicity* property on their sizes: For two subinstances I and I' with $|I| \leq |I'|$, the size of the optimal solutions of I should not exceed the size of the optimal solutions of I'.

The criterion **(C1)** guarantees the efficiency of the iteration procedure. With an augmentation element violating **(C2)**, we cannot say that the original instance is a "No"-instance, even when the compression fails to compress the size-$(k+1)$ solution for a subinstance.

In the following, we will investigate the applicability of the iteration procedure by doing some case studies. In particular, these cases studies demonstrate the usefulness of the two criteria given above for the choice of the augmentation element.

2.1.1 Case Study 1: Vertex Cover

Vertex Cover asks for a set C of at most $k \geq 0$ vertices in a given graph $G = (V, E)$ *covering* all edges in G, i.e., each edge in E has at least one endpoint in C. Since the optimization measure is the size of a vertex subset, adding a single vertex at each augmentation step seems to be most reasonable. Let $V := \{v_1, v_2, \ldots, v_n\}$.

Vertex Cover is known to be NP-complete [115]. The problem is approximable within a factor slightly better than 2. Unless P=NP an approximation factor of 1.36 cannot be achieved [56]. The currently best known fixed-parameter algorithm has runtime of $O(1.2745^k k^4 + kn)$ with $k := |C|$ [36]. For approximability and fixed-parameter tractability of several generations of Vertex Cover we refer to [24, 31, 44, 79, 92, 100, 122].

The initial subinstance is the subgraph G_1 of G containing only the vertex v_1 for which the vertex cover set is empty. Suppose that, for $1 < i \leq n$, we have computed a vertex cover C_{i-1} with $|C_{i-1}| \leq k$ for $G_{i-1} = G[V_{i-1}]$ where $V_{i-1} := \{v_1, v_2, \ldots, v_{i-1}\}$. We augment G_{i-1} to G_i by adding vertex v_i to G_{i-1}. The edges in E between v_i and the vertices in V_{i-1} are also added to G_{i-1}. Note that $G_i = G[V_i]$ where $V_i := \{v_1, v_2, \ldots, v_i\}$. If the vertices in C_{i-1} cover all edges in G_i, then $C_i := C_{i-1}$ is a vertex cover for G_i with at most k vertices; otherwise, $C_i := C_{i-1} \cup \{v_i\}$ is a vertex cover for G_i with no more than $k+1$ vertices, since the newly added edges in G_i are all incident to v_i. If $|C_i| = k+1$, then we invoke the compression procedure (see Section 2.2). The augmentation process is repeated until we reach the original input instance G.
Remark: Note that the computation of C_i for G_i based on C_{i-1} is simple: To determine the covering status of the edges incident to v_i can be clearly done in $O(|V_i|)$ time. Criterion **C1** for the choice of the augmentation element is fulfilled. Moreover, each vertex cover of G_i is clearly a vertex cover of G_{i-1} but not conversely. This implies the monotonicity property indicated in Criterion **C2**. Therefore, the above choice of the augmentation element results in a fairly simple iteration procedure for VERTEX COVER.

2.1.2 Case Study 2: Multicut in Trees

For a given input tree $T = (V, E)$, MULTICUT IN TREES asks for a set of at most k edges whose removal disconnects all vertex pairs given in an input set $H := \{(u_j, v_j) \mid u_j, v_j \in V \text{ and } 1 \leq j \leq h\}$. The following iteration procedure augments the subinstances by adding a vertex pair from H in each step.

The initial instance consists of tree T and the set H_1 which only contains the vertex pair $(u_1, v_1) \in H$. The solution M_1 for this initial subinstance is easy to compute: We determine the unique path between u_1 and v_1 in T and add an arbitrary edge of this path to M_1. Suppose that we have a solution M_{i-1} with $|M_{i-1}| \leq k$ for the subinstance consisting of T and $H_{i-1} := \{(u_j, v_j) \mid u_j, v_j \in V \text{ and } 1 \leq j \leq i-1\} \subseteq H$. The subinstance with H_{i-1} is then augmented by adding the vertex pair $(u_i, v_i) \in H$ to H_{i-1}. Consider the unique path between u_i and v_i in T. If at least one of the edges in this path is contained in M_{i-1}, then the solution M_i for the new, augmented subinstance is set equal to M_{i-1}. Otherwise, we get M_i by adding an arbitrary edge of this path to M_{i-1}. It is clear that $|M_i| \leq k+1$ and the compression procedure is invoked if $|M_i| = k+1$. This completes the description of the iteration procedure.
Remark: In this problem it seems to be more complicated to augment the subinstances by adding vertices or edges. The linear-time computation of M_i from M_{i-1} is simple, and disconnecting the vertex pairs in H_i needs clearly at least as many edge removals as disconnecting the vertex pairs in H_{i-1}. Therefore, the two criteria **C1** and **C2** are fulfilled by the chosen augmentation element, that is the vertex pairs in H.

Unfortunately, we are unable to give a compression procedure for this problem. One possible direction for deriving a compression procedure could be to combine the data reduction rules introduced in Chapter 8 and the branching

Figure 2.1: Graph G_i results from graph G_{i-1} by adding edge e_i.

into two subcases in Chapter 9.

2.1.3 Case Study 3: Cluster Deletion

In this case study we give an example for which there is no obvious augmentation element satisfying both criteria. This problem is called CLUSTER DELETION: Find a set of at most k edges whose removal transforms a given graph into a vertex-disjoint union of cliques. Let $G = (V, E)$ denote the input graph. By Lemma 10.1, a graph is a vertex-disjoint union of cliques iff it contains no induced P_3, a path with three vertices. So, we can formulate CLUSTER DELETION as the problem to get rid of all induced P_3 by deleting at most k edges.

At first sight, it seems very natural to use the graph edge as augmentation element when minimizing the number of edge deletions. However, consider the two subgraphs of a graph G in Figure 2.1: G_i is the new, augmented subgraph constructed by adding edge e_i to G_{i-1}. It is easy to see that the solutions of G_{i-1} and G_i do not have the monotonicity property of **C2**: G_i is already a clique and, hence, needs no edge deletion, while G_{i-1} needs at least two edge deletions. Therefore, the graph edges are not the proper augmentation elements.

With the graph vertices as the augmentation element, it is not clear how to compute the solution with at most $k + 1$ edge deletions of the new, augmented instance.

Remark: The described difficulty also occurs with other edge modification problems with a finite set of forbidden subgraphs. It remains as an open problem to uncover other augmentation elements which lead to feasible iteration procedures for such edge modification problems.

2.2 Compression

Now, we turn our attention to the compression procedure which is invoked by the iteration procedure. It has an instance and a solution of size $k+1$ as inputs. The compression procedure returns "no" if the size-$(k+1)$ solution is an optimal solution for the input instance; otherwise, it computes a solution with a size at most k.

Compared to the original problem formulation, this procedure solves actually a slightly easier problem since it has more information about the input instance,

i.e., the instance has a size-$(k+1)$ solution. However, for most problems, this seemingly easier problem is still NP-hard: The iteration procedures for VERTEX COVER and MULTICUT IN TREES have a polynomial runtime as shown in Section 2.1. If the compression procedures for these problems could be done in polynomial time as well, then we would have polynomial-time algorithms solving these problems, which would imply P=NP.

In contrast to the iteration procedure, it is hard to give a general scheme for the compression procedure which turns out to be highly problem-specific as demonstrated in the following case studies. However, observe that there exists a common property of the input instances of the compression procedures for different problems: They have all a solution of size $k+1$. A possible approach to do the compression could be to explore the relation between the size-$(k+1)$ solution and the possibly existing size-k solution. The following simple observation provides a start point for designing the compression procedure as in the compression procedure for GRAPH BIPARTIZATION by Reed et al. [155].

> Given an instance together with a size-$(k+1)$ solution S, if there exists a size-k solution S', then S can be partitioned into two disjoint subsets X and Y such that $X \cup Y = S$, $Y \neq \emptyset$, $X \subseteq S'$, and $Y \cap S' = \emptyset$.

Based on this observation, we can always begin with partitioning the size-$(k+1)$ solution S into two disjoint subsets X and Y in all possible ways and, for each partition, we try to find a size-k solution S' containing X but not Y. If we have a size-k solution for one partition, then this solution is returned as output of the compression procedure. If there is no size-k for all partitions, then the size-$(k+1)$ solution is optimal and we answer "No." Note that, for each partition, $X \subseteq S'$ *and* $Y \cap S' = \emptyset$ have to be fulfilled by the possibly existing size-k solution S'. This means that, if there exists a size-k solution S' for this partition, then S' is obtained by replacing the elements in Y by at most $|Y|-1$ elements which are not in S, a significantly restricted version of the original problem. For some problems as shown in the case studies, we can determine in polynomial time the elements which are not in S and which are needed to replace the elements of Y. Note that, since there are 2^{k+1} such partitions of S, the compression procedure based on this partitioning and replacing scheme requires at least $O(2^k)$ steps.

The following two case studies give some concrete examples for the compression procedure.

2.2.1 Case Study 1: Vertex Cover

If a size-$(k+1)$ vertex cover C is known for a VERTEX COVER instance $G = (V, E)$, then we try all possible ways to partition C into two disjoint sets X and Y as described above. Note that $Y \neq \emptyset$ because we seek for a smaller solution. There are 2^{k+1} such partitions. For each partition, we assume that the possibly existing size-k vertex cover C' has to contain X but not Y. Then, we delete the vertices in X and their incident edges since these vertices have to be in C'

and they cover the deleted edges. Observe that, if such a vertex cover C' exists, then the induced subgraph $G[V \setminus X]$ needs to be bipartite with edges between Y and $V \setminus C$. Because $Y \cap C' = \emptyset$, we have to take all neighbors of the vertices in Y into C' to cover the remaining edges. If $|N(Y)| < |Y|$, then $C' := N(Y) \cup X$; otherwise, there is no size-k vertex cover for this partition. Combining with the iteration procedure given in Section 2.1.1, we have an iterative compression algorithm for VERTEX COVER with a runtime of $O(2^k \cdot |V|^2)$.
Remark: We can do a more clever partitioning by examining the degrees of the vertices in C or by making case distinctions based on whether there is an edge in $G[C]$ or not. For example, in order to cover the edges incident to a vertex v, we have to take v or all its neighbors into a vertex cover. Thus, if there is a vertex v in C with more than k neighbors in G, then we do not consider the partitions of C with $v \in Y$, since these partitions cannot lead to a vertex cover with at most k vertices. However, it seems to be difficult to beat the algorithm based on a depth-bounded search tree approach for VERTEX COVER which has a runtime of $O(1.2745^k k^4 + k|V|)$ [36].

2.2.2 Case Study 2: Cluster Deletion

As shown in Section 2.1.3 it seems to be difficult to give an iteration procedure for CLUSTER DELETION, the compression procedure is fairly simple: Given a CLUSTER DELETION instance with a size-$(k+1)$ solution S, we partition S into X and Y. For each partition we delete the edges in X due to the assumption of this partition that they have to be in the possibly existing size-k solution S', and we initialize $S' := X$. In the resulting graph, we search for the remaining P_3's, paths induced by three vertices. If there is a P_3, then one edge of this P_3 has to be in Y; otherwise, S would not be a solution. Since $Y \cap S' = \emptyset$, we have to add the other edge of this P_3 to S' and, then, we delete it from the graph. This process is repeated until there is no more P_3. Finally, if $|S'| \leq k$, then S' is the output of the compression procedure; otherwise, for this partition, there is no size-k solution. Clearly, the compression procedure needs $O(2^k \cdot |V|^3)$ time where $O(|V|^3)$ is due to the search for P_3's.
Remark: For most problems, the compression procedure seems to be more complicated than the iteration procedure. In particular, the W[2]-complete DOMINATING SET has an iteration procedure running in polynomial time by using the graph vertices as the augmentation elements, while compressing a size-$(k+1)$ dominating set is fixed-parameter intractable with respect to k; otherwise, it would imply FPT=W[2] which is generally believed not to be true. CLUSTER DELETION seems to be an exception.

2.3 Concluding Remarks

Iterative compression is a very new technique for designing fixed-parameter algorithms solving minimization problems. As already demonstrated by the example GRAPH BIPARTIZATION [155], this technique seems to be useful for

attacking unsettled questions in fixed-parameter complexity. In Chapter 3, we give another successful application of the iterative compression technique to the FEEDBACK VERTEX COVER problem, achieving an $O(c^k \cdot m)$-time algorithm where c is a constant, k denotes the size of the solution, and m is the number of edges in the given graph. This result answers a more than ten years open question.[1] Compared with other fixed-parameter design techniques, so far little research has been done on iterative compression and there remain many challenging open questions concerning this new technique such as whether it can also be used to derive polynomial-time approximation algorithms.

By incorporating additional techniques into the iterative compression based algorithm by Reed et al. [155] and evaluating its performance on real-world data from computational biology and synthetic data, Hüffner [108] demonstrated that iterative compression may lead to practically useful algorithms. Experiments with other iterative compression based algorithms should shed more light on the potential of iterative compression.

[1] Independently, this result was also shown by Dehne et al. [52].

Chapter 3

Feedback Vertex Set

In Chapter 2 we have demonstrated the usefulness of the iterative compression technique for some concrete graph problems. However, none of the algorithms given there could improve the runtimes of the already known fixed-parameter algorithms for these problems which are based on other algorithmic techniques. In this chapter we show a successful application of the iterative compression technique to the FEEDBACK VERTEX SET problem with the until now best fixed-parameter time complexity. Independently, this result was also shown by Dehne et al. [52]. We follow partly [93].

3.1 Problem Definition and Previous Results

The FEEDBACK VERTEX SET (FVS) problem is defined as follows:

> **Input**: An undirected graph $G = (V, E)$ and an integer $k \geq 0$;
> **Task**: Find a set F with $|F| \leq k$ vertices such that each cycle in G contains at least one vertex from F. (The removal of all vertices in F from G therefore results in a forest.)

The vertex set F is called a *feedback vertex set (fvs)*.

FVS is one of the classical NP-complete problems [115] and it is known that an optimal solution can be approximated to a factor of two in polynomial time [15]. FVS is MaxSNP-hard [131] and, hence, there is no hope for polynomial-time approximation schemes. A question of similar importance as approximability is to ask how fast one can find an *optimal* feedback vertex set. There is a very simple randomized algorithm due to Becker et al. [19] which solves the FVS problem in $O(c \cdot 4^k \cdot kn)$ time by finding a feedback vertex set of size k with probability at least $1 - (1 - 4^{-k})^{c4^k}$ for an arbitrary constant c. Note that this means that by choosing an appropriate value c, one can achieve any constant error probability independent of k.

Bodlaender [26] and Downey and Fellows [64] were the first to show that the problem is fixed-parameter tractable. A fixed-parameter algorithm with

runtime $O((2k+1)^k \cdot n^2)$ was described by Downey and Fellows [65]. In 2002, Raman et al. [152] made a significant step forward by proving the improved upper bound $O(\max\{12^k, (4\log k)^k\} \cdot n^\omega)$ (where n^ω denotes the time to multiply two $n \times n$ integer matrices). Using results from extremal graph theory, Kanj et al. [113] have slightly improved this bound to $O((2\log k + 2\log\log k + 18)^k \cdot n^2)$. Finally, Raman et al. [153] published an algorithm with a further improved upper bound $O((12\log k/\log\log k + 6)^k \cdot n^\omega)$.

For an overview on FVS and its variants, we refer to Festa, Pardalos, and Resende [77].

3.2 The Algorithm

Now we show that FEEDBACK VERTEX SET can be solved in $O(c^k \cdot n \cdot m)$ time for $c \approx 37.7$ by presenting an algorithm based on iterative compression, where $n := |V|$ and $m := |E|$ for an input graph $G = (V, E)$.[1] Without changing the asymptotic runtime, this algorithm can solve a more general problem (introduced in [18]) where some graph vertices are marked "blackout" and may not be part of the feedback vertex set.

3.2.1 Iteration

The iteration procedure of this algorithm is quite simple. We use the graph vertices as augmentation elements.

Given as input a graph G with vertex set $\{v_1, \ldots, v_n\}$, the initial subinstance is $G_1 := G[\{v_1\}]$. The optimal fvs F_1 of G_1 is empty. For $i > 1$, suppose that an fvs F_{i-1} with $|F_{i-1}| \leq k$ for $G_{i-1} = G[\{v_1, \ldots, v_{i-1}\}]$ is known. Obviously, $F_i := F_{i-1} \cup \{v_i\}$ is an fvs for G_i. If $|F_i| = k + 1$, then the compression procedure is invoked which will be given in the next subsection; otherwise, we proceed with G_{i+1}. For $i = n$, we thus have computed, if existing, an fvs F with $|F| \leq k$ for G in $T_C \cdot O(n)$ time, where T_C denotes the runtime of the compression procedure. The correctness of the iteration procedure is obvious.

3.2.2 Compression

The following lemma provides the compression procedure.

Lemma 3.1. *Given a graph G and a size-$(k+1)$ feedback vertex set (fvs) F for G, we can decide in $O(c^k \cdot m)$ time for some constant c whether there exists a size-k fvs F' for G and, if so, provide one.*

Proof. The compression procedure consists of two steps. The first step, as described in Chapter 2, is to try by brute force all 2^{k+1} partitions of F into two sets X and Y. In the second step, for each partition, we assume that a possible smaller fvs F contains X but not Y. The key idea for replacing the

[1] By using almost the same technique but refined analysis, Dehne et al. [52] improved the constant c to 10.6.

vertices in Y by at most $|Y| - 1$ vertices from $V \setminus F$ is to show that there is only a "small" set V' of candidate vertices to draw from in order to complete X to F'. As later shown in Lemma 3.2, we apply two simple data reductions and compute V' in $O(m)$ time. Moreover, Lemma 3.2 shows that the size of V' is bounded from above by $14 \cdot |Y|$. Since $|Y| \leq k+1$, $|V'|$ thus only depends on the problem parameter k and not on the input size. We again use brute force and test each of the at most $\binom{14 \cdot |Y|}{|Y|-1}$ possible choices of vertices from V' whether it is an fvs of $G[V']$. If one of the choices is an fvs, then we add it to X to form F' and return F' as the output of the compression procedure. Finally, if there is no such fvs F' as desired for all partitions, then the compression procedure outputs "NO." Note that the second step is the problem-specific part of the compression procedure.

Since the test whether a choice of vertices from V' together with X forms an fvs can easily be done in $O(m)$ time, we can now bound the overall runtime T_C of the compression procedure, where the index i corresponds to a partition of F into X and Y with $|X| = i$ and $|Y| = |F| - i$:

$$\begin{aligned} T_C &= O\left(\sum_{i=0}^{k} \binom{|F|}{i} \cdot \left(O(m) + \binom{14 \cdot (|F| - i)}{|F| - i - 1} \cdot O(m)\right)\right) \\ &= O\left(2^k \cdot m + \sum_{i=0}^{k} \binom{k+1}{i} \cdot \binom{14 \cdot (k+1-i)}{k-i} \cdot m\right) \end{aligned}$$

and with Stirling's inequality to evaluate the second binomial coefficient,

$$= O\left(2^k \cdot m + \sum_{i=0}^{k} \binom{k+1}{i} (36.7)^{k+1-i} \cdot m\right) = O((1+36.7)^k \cdot m),$$

which gives the lemma's claim with $c \approx 37.7$. □

It remains to show the size bound of the "candidate vertices set" V' for fixed partition X and Y of a size-$(k+1)$ fvs F. To this end, we make use of two simple data reduction rules.

Lemma 3.2. *Given a graph $G = (V, E)$, a size-$(k+1)$ fvs F for G, and a partition of F into two sets X and Y. Let F' denote a size-k fvs for G with $F \cap F' = X$ and $F' \cap Y = \emptyset$. In $O(m)$ time, we can either decide that no such F' exists or compute a subset V' of $V \setminus F$ with $|V'| \leq 14 \cdot |Y|$ such that there exists an F' as desired consisting of $|Y| - 1$ vertices from V' and all vertices from X.*

Proof. The idea of the proof is to use an obvious data reduction technique for FVS to get rid of degree-1 and degree-2 vertices and to show that if the resulting instance is too large as compared to the part Y (whose vertices we are not allowed to add to F'), then there exists no set F' as desired.

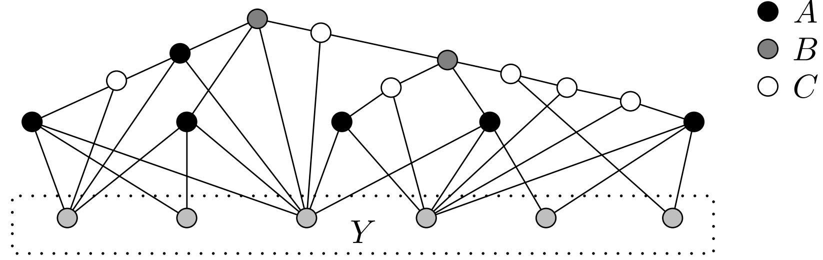

Figure 3.1: Partition of the vertices in V' into three disjoint subsets A, B, and C.

First, check that Y does not induce a cycle; otherwise, no F' with $F' \cap Y = \emptyset$ can be an fvs for G. Then, remove in G all vertices from X as they are determined to be in F'. Finally, apply a standard data reduction to the vertices in $V \setminus F$ (the vertices in Y remain unmodified): remove degree-1 vertices and successively bypass any degree-2 vertex by adding a new edge between its neighbors (thereby removing the bypassed degree-2 vertex together with its incident edges). There are two exceptions to note: One exception is that we do not bypass a degree-2 vertex which has two neighbors in Y. The other exception is the way to deal with parallel edges. If we create two parallel edges between two vertices during the data reduction process—these two edges form a length-two cycle—, then exactly one of the two endpoints of these edges has to be in Y since Y is an fvs of $G[V \setminus X]$ and $G[Y]$ contains no cycle. Thus, we have to delete the other endpoint and add it to F' since we are not allowed to add vertices from Y to F'. Proofs for the correctness and the time bound of the data reduction technique are basically straightforward, see Section 6.1.1.

In the following we use $G' = (V' \cup Y, E')$ with $V' \subseteq V \setminus F'$ to denote the graph resulting after exhaustive application of the data reduction described above; note that none of the vertices in Y has been removed during the data reduction process. In order to prove that $|V'| < 14 \cdot |Y|$, we partition V' into three subsets, each of which will have a provable size bound linearly depending on $|Y|$ (the partition is illustrated in Figure 3.1):

$$A := \{v \in V' \mid |N(v) \cap Y| \geq 2\},$$
$$B := \{v \in V' \setminus A \mid |N(v) \cap V'| \geq 3\},$$
$$C := V' \setminus (A \cup B).$$

To bound the number of vertices in A, consider the bipartite subgraph $G_A = (A \cup Y, E_A)$ of $G' = (V' \cup Y, E')$ with $E_A := (A \times Y) \cap E'$. Observe that if there are more than $|Y| - 1$ vertices in A, then there is a cycle in G_A: If G_A is acyclic, then G_A is a forest, and, thus, $|E_A| \leq |Y| + |A| - 1$. Moreover, since each vertex in A has at least two incident edges in G_A, $|E_A| \geq 2|A|$, which implies that $|A| \leq |Y| - 1$ if G_A is acyclic. It follows directly that if $|A| \geq 2|Y|$, it is impossible to delete at most $|Y|$ vertices from A such that $G'[A \cup Y]$ is acyclic.

To bound the number of vertices in B, observe that $G'[V']$ is a forest. Furthermore, all leaves of the trees in $G'[V']$ are from A since G' is reduced with respect to the above data reduction rules. By the definition of B, each vertex in B has at least three vertices in V' as neighbors. Thus, there cannot be more vertices in B than in A, and therefore $|B| < 2|Y|$.

Finally, consider the vertices in C. By the definitions of A and B, and since G is reduced, each vertex in C has degree two in $G'[V']$ and exactly one neighbor in Y. Hence, graph $G'[C]$ is a forest consisting of paths and isolated vertices. We now separately upperbound the number of isolated vertices and those participating in paths.

Each of the isolated vertices in $G'[C]$ connects two vertices from $A \cup B$ in $G'[V']$. Since $G'[V']$ is acyclic, the number of isolated vertices in $G'[C]$ cannot exceed $|A \cup B| - 1 < 4|Y|$. The total number of vertices participating in paths in $G'[C]$ can be upperbounded as follows: Consider the subgraph $G'[C \cup Y]$. Each edge in $G'[C]$ creates a path between two vertices in Y, that is, if $|E(G'[C])| \geq |Y|$, then there exists a cycle in $G'[C \cup Y]$. By an analogous argument to the one that upperbounded the size of A (and considering that removing a vertex from $G'[C]$ destroys at most two edges), the total number of edges in $G'[C]$ may thus not exceed $|Y| + 2|Y|$, bounding the total number of vertices participating in paths in $G'[C]$ from above by $6|Y|$.

Altogether,

$$|V'| = |A| + |B| + |C| < 2|Y| + 2|Y| + (4 + 6)|Y| = 14|Y|.$$

□

Including the runtime of the compression procedure $T_C = O(c^k \cdot m)$ into the runtime of the iteration procedure $O(n) \cdot T_C$, we have the main result of this chapter.

Theorem 3.1. Feedback Vertex Set *can be solved in* $O(c^k \cdot mn)$ *time for a constant* c.

3.3 Concluding Remarks

The successful application of the iterative compression technique to Feedback Vertex Set demonstrates that this technique is an important tool not only in classifying the fixed-parameter tractability but also in the design of efficient fixed-parameter algorithms for minimization problems.

In the next two chapters we will show that this algorithm can be improved to a *linear-time* fixed-parameter algorithm, i.e., an algorithm running in linear time for constant value of parameter k, and we will also extend the compression procedure in order to enumerate all optimal solutions of FVS asymptotically within the same runtime.

The practical performance of the algorithm is an issue remaining to be explored. Using algorithm engineering methods, Hüffner [108] demonstrated the

efficiency of the iterative compression algorithm by Reed et al. [155] for the closely related GRAPH BIPARTIZATION problem with real-world data sets. To this end, it also would be useful to develop data reduction rules and kernelizations (see Part III) for FVS.

Finally, it remains a long-standing open problem whether FEEDBACK VERTEX SET in *directed* graphs is fixed-parameter tractable. The iterative compression technique seems to be a promising approach for attacking this question.

Chapter 4

Speed-up Methods

In this chapter, we introduce two speed-up methods for the iteration compression technique. They are not problem-specific and, in general, their application can simplify the fixed-parameter algorithms based on iterative compression.

4.1 Compression without Partition

In Section 2.2, we give a general compression scheme consisting of partitioning the size-$(k+1)$ solution S into two disjoint subsets and analyzing each single partition. The scheme is based on the observation that, if there exists a size-k solution S', then S' keeps some elements of S and replaces the other elements by some elements not in S. However, if there is a size-k solution which has no common element with the size-$(k+1)$ solution, then we do not have to do the partition and only have to consider the case that $S \cap S' = \emptyset$, i.e., the partition of S into X and Y with $X = \emptyset$. This implies that we can reduce the run time by a factor of $O(2^k)$. We call this method "compression without partition." Note that this compression without partition method is not generally applicable. For some problems, there is no size-k solution having no common element with a size-$(k+1)$ solution.

In the following, we show an example of the application of this method. The considered problem is called EDGE BIPARTIZATION:

> **Input**: An undirected graph $G = (V, E)$ and an integer $k > 0$.
> **Task**: Find a subset $B \subseteq E$ of edges with $|B| \leq k$ whose removal transforms G into a bipartite graph.

A necessary and sufficient condition for a graph to be a bipartite graph is that it contains no cycle of odd length. The task of EDGE BIPARTIZATION can also be formulated as seeking for a set B of at most k edges such that each odd-length cycle in G contains at least one edge from B. The set B is called an *edge bipartization* set. We call a set of edges E' a *edge cut* set if the removal of the edges in E' decomposes the graph into at least two connected components. Given a set $E' \subseteq E$ of edges, $V(E')$ denotes the set $\bigcup_{\{u,v\} \in E'} \{u, v\}$ of endpoints.

EDGE BIPARTIZATION is known to be MaxSNP-hard [149] and can be approximated to a factor of $O(\log n)$ in polynomial time [81]. It has applications in genome sequence assembly [150] and VLSI chip design [112]. Since there is a parameter-preserving reduction from EDGE BIPARTIZATION to GRAPH BIPARTIZATION [177], the algorithm by Reed et al. [155] for GRAPH BIPARTIZATION directly implies a run time of $O(3^k \cdot k^3 m^2 n)$ for EDGE BIPARTIZATION, k denoting the number of edges to be deleted. Here, we significantly reduce the run time from $O(3^k \cdot k^3 m^2 n)$ to $O(2^k \cdot m^2)$ by using the compression without partition method.

The following lemma provides some central insights into the structure of a minimal edge bipartization set.

Lemma 4.1. *Given a graph $G = (V, E)$ with a minimal edge bipartization set B for G, the following two properties hold:*

1. *For every odd-length cycle C in G, $|E(C) \cap B|$ is odd.*
2. *For every even-length cycle C in G, $|E(C) \cap B|$ is even.*

Proof. For each edge $e = \{u, v\} \in B$, note that u and v are on the same side of the bipartite graph $G \setminus B$, since otherwise we do not need e to be in B and B would not be minimal. Consider a cycle C in G. The edges in $E(C) \setminus B$ are all between the two sides of $G \setminus B$, while the edges in $E(C) \cap B$ are between vertices of the same side as argued above. In order for C to be a cycle, however, this implies that $|E(C) \setminus B|$ is even. Since $|E(C)| = |E(C) \setminus B| + |E(C) \cap B|$, we conclude that $|E(C)|$ and $|E(C) \cap B|$ have the same parity. □

As mentioned above, the applicability of the compression without partition depends on the assumption that there exists a size-k solution disjoint from the given size-$(k+1)$ solution. In order to show that the assumption holds for EDGE BIPARTIZATION, we apply a simple transformation to the input graph. For a graph G with a size-$(k+1)$ edge bipartization set B, we subdivide all edges in X by two new vertices. This transformation is formalized in the following definition.

Definition 4.1. *For a graph $G = (V, e)$ with a minimal edge bipartization set B, the vertex set and the edge set of the corresponding* edge-extension graph $\tilde{G} := (\tilde{V}, \tilde{E})$ *are defined as*

$$\begin{aligned} \tilde{V} &:= V \cup \{u_e, v_e \mid e \in B\} \text{ and} \\ \tilde{E} &:= (E \setminus B) \cup \{\{u, u_e\}, \{u_e, v_e\}, \{v_e, v\} \mid e = \{u, v\} \in B\}. \end{aligned}$$

See Figure 4.1 for an example of this transformation. Since this transformation preserves the parity of the length of cycles, it is easy to see that the thus transformed graph has an edge bipartization set with k edges iff the original graph has an edge bipartization set with k edges. Moreover, for each edge bipartization set B for the transformed graph there is an edge bipartization set B'

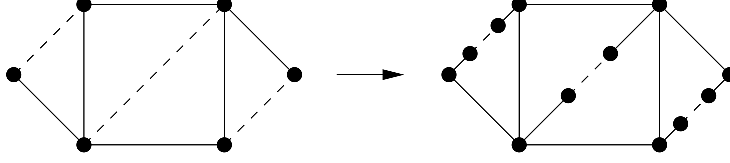

Figure 4.1: The left graph has an edge bipartization set B (dashed lines). To be able to assume without loss of generality that a bipartization set smaller than B is disjoint from B, we subdivide each edge in B by two vertices and choose the middle edge from each thus generated path as new edge bipartization set (right).

of the same size that is disjoint from B, which can be obtained by replacing every edge in $B' \cap B$ by one of its two adjacent edges. In the following, for all considered graphs, it is implicitly assumed that they are transformed.

The following simple definition is the only remaining prerequisite for the central lemma of this section.

Definition 4.2. *Let $G = (V, E)$ be a graph and $B \subseteq E$. A mapping $\Phi : V(B) \to \{0, 1\}$ is called* valid partition of $V(B)$ *if for each $\{u, v\} \in B$, we have $\Phi(u) \neq \Phi(v)$.*

Lemma 4.2. *Consider a graph $G = (V, E)$ and a minimal edge bipartization set B for G. For a set of edges $B' \subseteq E$ with $B \cap B' = \emptyset$, the following are equivalent:*

(1) B' is an edge bipartization set for G.

(2) There is a valid partition Φ of $V(B)$ such that B' is an edge cut between $0_\Phi := \Phi^{-1}(0)$ and $1_\Phi := \Phi^{-1}(1)$ in $G \setminus B := (V, E \setminus B)$.

Proof. (2) $\Rightarrow$ (1): Consider any odd-length cycle C in G. We show that $E(C) \cap B' \neq \emptyset$. Let $s := |E(C) \cap B|$. By Property 1 in Lemma 4.1, s is odd. Without loss of generality, we assume that $E(C) \cap B = \{\{u_0, v_0\}, \ldots, \{u_{s-1}, v_{s-1}\}\}$ with vertices v_i and $u_{(i+1) \bmod s}$ being connected by a path in $C \setminus B$. Since Φ is a valid partition, we have $\Phi(u_i) \neq \Phi(v_i)$ for all $0 \leq i < s$. With s being odd, this implies that there is a pair $v_i, u_{(i+1) \bmod s}$ such that $\Phi(v_i) \neq \Phi(u_{(i+1) \bmod s})$. Since the removal of B' destroys all paths in $G \setminus B$ between 0_Φ and 1_Φ, we obtain that $E(C) \cap B' \neq \emptyset$.

(1) $\Rightarrow$ (2): See Figure 4.2 for an illustration of a valid partition Φ of $V(B)$ and graph $G \setminus B' := (V, E \setminus B')$. Then graph $G \setminus B'$ can be partitioned into two subgraphs: G_1 contains $\Phi^{-1}(0)$ and the vertices which are connected to $\Phi^{-1}(0)$ in graph $(V, E \setminus (B \cup B'))$, and G_2 contains $\Phi^{-1}(1)$ and the vertices which are connected to $\Phi^{-1}(1)$ in graph $(V, E \setminus (B \cup B'))$. For the purpose of contradiction, suppose that $G \setminus B'$ is not bipartite. Because B is a minimal edge bipartization set, all odd-length cycles in $G \setminus B'$ have to contain an odd number of edges in B (Lemma 4.1). It is easy to verify that such an odd-length cycle, passing odd

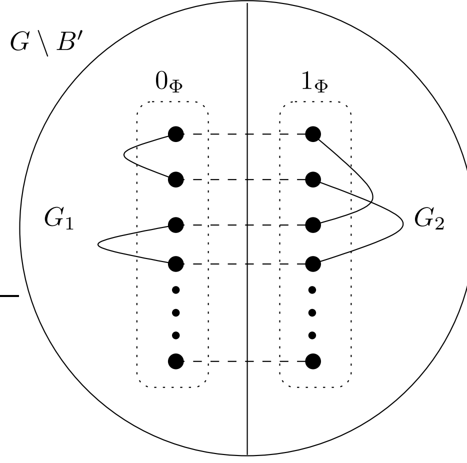

Figure 4.2: A valid partition Φ of $V(B)$ and graph $G \setminus B'$. The edges in B are drawn as dashed lines. Subgraph G_1 of $G \setminus B'$ contains $\Phi^{-1}(0)$ and the vertices connected to $\Phi^{-1}(0)$ in graph $(V, E \setminus (B \cup B'))$, and subgraph G_2 contains $\Phi^{-1}(1)$ and the vertices connected to $\Phi^{-1}(1)$ in graph $(V, E \setminus (B \cup B'))$. By Lemma 4.1 the cycle passing four edges in B shown here is an even-length cycle.

number of edges in B, needs at least one path from G_1 to G_2 (or from G_2 to G_1) which contains no edge from B. This implies that the edges in B are not the only connections between G_1 and G_2. Thus, we can infer that B' is not an edge cut between 0_Φ and 1_Φ in $G \setminus B := (V, E \setminus B)$. □

Lemma 4.2 shows that EDGE BIPARTIZATION fulfills the assumption that, if the given EDGE BIPARTIZATION instance with an edge-extension graph has a size-k solution, then there exists a size-k solution B' disjoint from the given bigger solution B of size $k+1$. Then, our compression procedure only considers the partition X and Y of B with $X = \emptyset$.

Theorem 4.1. EDGE BIPARTIZATION *can be solved in* $O(2^k \cdot m^2)$ *time.*

Proof. The iteration procedure uses the graph edges as the augmentation element: Given as input a graph G with edge set $\{e_1, \ldots, e_m\}$, we iteratively consider the graphs G_i induced by the edge set $\{e_1, \ldots, e_i\}$ for $i = 1, \ldots, m$. For $i = 1$, the optimal edge bipartization set is empty. For $i > 1$, assume that a minimal edge bipartization set B_{i-1} with $|B_{i-1}| \leq k$ for G_{i-1} is known. If B_{i-1} is not an edge bipartization set for G_i, then we consider the set $B_{i-1} \cup \{e_i\}$, which obviously is a minimal edge bipartization set for G_i. Using the compression procedure which is described in the following either to determine that $B_{i-1} \cup \{e_i\}$ is an optimal edge bipartization set for G_i or, if not, to compute an size-k edge bipartization set B_i for G_i. Let T_C denote the run time of the compression procedure. This process outputs "NO" if $|B_i| > k$, since then no solution exists.

Summing over all iterations, we have an algorithm that computes an size-k edge bipartization set for G in $T_C \cdot O(m)$ time.

It remains to describe the compression step that, given a graph and a minimal edge bipartization set B of size $k+1$, either computes a smaller edge bipartization set B' in $O(2^k \cdot km)$ time or proves that no such B' exists. For this, we first apply the input transformation from Figure 4.1 which allows us to assume the prerequisite of Lemma 4.2 that $B' \cap B = \emptyset$. We then enumerate all 2^k valid partitions Φ of $V(B)$ and determine a minimum edge cut between 0_Φ and 1_Φ until we find an edge cut B' of size k. Each of the minimum cut problems can individually be solved in $O(km)$ time with the Edmonds-Karp algorithm that goes through k' rounds, each time finding and augmenting a flow augmenting path [45]. By Lemma 4.2, B' is an edge bipartization set; furthermore, if no such B' is found, we know that B is optimal. Since there are at most 2^k many valid partitions of $V(B)$, we get $T_C = O(2^k \cdot km)$. With the same technique as used by Hüffner [108] to improve the run time of the iterative compression algorithm for Graph Bipartization, T_C here can also be improved from $O(2^k \cdot km)$ to $O(2^k \cdot m)$.

Altogether, we have the overall run time of $O(2^k \cdot m^2)$. □

4.2 Constant-Factor Approximation Instead of Iteration

In this and the previous chapter we have presented several algorithms based on the iterative compression technique. None of these algorithms is a *linear-time* fixed-parameter algorithm, i.e., an algorithm running in linear time for constant value of parameter k. Note that a linear-time algorithm is the best one usually can hope for a non-trivial problem and, therefore, attracts strong interest from the field of algorithmic research. For example, Fiorini et al. [78] very recently showed, by significant technical expenditure, a linear-time fixed-parameter algorithm for the Graph Bipartization problem restricted to planar graphs. However, algorithms relying on both iteration and compression cannot be linear-time fixed-parameter algorithms since, on the one hand, we have to do $O(|I|)$ augmentations where I denotes the input instance and, on the other hand, the compression procedure needs at least $O(|I|)$ time. This means that, if we want to use the iterative compression technique to design linear-time fixed-parameter algorithms, then we have to save one of the two procedures. In the following, we show by an example that for a class of problems we can save the iteration procedure.

The example used here is the Feedback Vertex Set problem introduced in Chapter 3: One asks for a set of at most k vertices whose removal transforms an input graph into a forest. The key observation leading to saving the iteration procedure is that the most important task of the iteration procedure is to feed the compression procedure with a size-$(k+1)$ solution. However, a constant-factor approximation algorithm for a minimization problem provides a size-ck

solution for the compression procedure for a constant $c > 0$. Together with the observation that the exponential term of the run time of the compression procedure depends only on the size of the given large solution, we have still a fixed-parameter algorithm for the compression procedure. For FEEDBACK VERTEX SET, by Lemma 3.1, if the compression procedure has a size-ck solution instead of a size-$(k+1)$ solution as a part of the input, then the compression procedure has a combinatorial explosion still upperbounded by c_1^k, however for a larger constant c_1:

Theorem 4.2. FEEDBACK VERTEX SET *can be solved in* $O(c^k \cdot m)$ *time for a constant* c*.*

Proof. We first determine in $O(m)$ time a factor-4 approximation as described by Bar-Yehuda et al. [18].[1] This gives us the precondition for Lemma 3.1 with $|F| = 4k$ instead of $|F| = k+1$. Now, we can employ the same technique as in the proof of Lemma 3.1 to obtain the desired run time: we examine 2^{4k} partitions $X \dot\cup Y$ of F, and—by applying the arguments from Lemma 3.2—for each partition there is some constant c' such that the number of candidate vertices is bounded from above by $c' \cdot |Y|$. In summary, there is some constant c such that the run time of the compression step is bounded from above by $O(c^k \cdot m)$. Since one of the 2^{4k} partitions must lead to a solution of size k, we need only one compression step to obtain an optimal solution, which proves the claimed run time bound. □

Note that any improvement of the approximation factor of a linear-time approximation algorithm for FEEDBACK VERTEX SET below four will immediately improve the run time of the linear-time fixed-parameter algorithm described in Theorem 4.2.

This speed-up method can be applied to all problems which have a linear-time constant-factor approximation.

4.3 Concluding Remarks

There is another speed-up method which does not improve the asymptotic runtime of the iterative compression based algorithm, but can be relevant for practical performance. For example, EDGE BIPARTIZATION has no constant-factor approximation and, thus, the iteration procedure seems to be unavoidable. We can, however, apply a more efficient iteration procedure: Instead of starting with a subgraph containing only one edge, we can start the iteration procedure with a spanning tree of the input graph. Obviously, a tree is a bipartite graph and its optimal edge bipartization set is an empty set. Since, for a tree $T = (V, E)$, $|E| = |V| - 1$, this new iteration procedure needs at most $m - n$ iterations. The overall runtime of the algorithm given in Section 4.1 can be improved to $O(2^k \cdot m \cdot (m - n))$.

[1] Note that the factor-2 approximation algorithm by Bafna et al. [15] is not a linear-time algorithm.

Chapter 5

Compression-Based Enumeration

Parameterized enumeration, i.e., the question whether or not it is fixed-parameter tractable to find *all* minimal solutions of size at most k, has lately attracted some interest [51, 50, 76, 141]. Here we show that the iterative compression technique can be used not only for computing one minimal solution with size at most k but also for enumerating all minimal solutions with size at most k.

The basic scheme of the compression procedure described in Chapter 2 has actually a "brute-force property" required for enumeration algorithms: partitioning the size-$(k+1)$ solution S in all possible ways into X and Y and, for each partition, trying to compress S into a size-k solution S' with $X \subseteq S'$ and $Y \cap S' = \emptyset$. Using the same idea, we can easily construct an enumeration procedure with a given minimal solution S with size at most k.[1] We consider all possible partitions of S into X and Y and, for each partition, we enumerate all minimal solutions with size at most k containing X but not Y. This enumeration procedure is added at the end of the iterative compression algorithm and the output of the iterative compression algorithm, a minimal solution with size at most k, is further given to the enumeration procedure as a part of its input. If the iterative compression algorithm outputs "No", then we have no size-k solution and the output of the enumeration procedure is the empty set; otherwise, the enumeration procedure outputs all minimal solutions with size at most k enumerated for all possible partitions of the input size-k solution.

For example, all minimal solutions with size at most k of Vertex Cover can be easily enumerated. The enumeration procedure has the input graph $G = (V, E)$ and the output of the iterative compression algorithm, a minimal vertex cover C with size at most k, as inputs. For a partition of C into two sets X

[1] For the purpose of a clearer presentation, we divide the compression based enumeration into two phases: First, determine, given a size-$(k+1)$ solution, whether there is size-k solution for given instance and, if yes, construct one; then, using the size-k solution output by the first phase, enumerate all size-k solutions. For a more efficient implementation, one could combine both phases, i.e., directly enumerate size-k solutions with a given size-$(k+1)$ solution.

and Y, there is only one minimal vertex cover C satisfying $|C'| \leq k$, $X \subseteq C'$, and $Y \cap C' = \emptyset$: the union of X and the set of all neighbors of Y in $V \setminus C$. If the size of the union is at most k, then we have a minimal vertex cover C' with $|C'| \leq k$. The enumeration procedure terminates after repeating this operation for all partitions. It is obvious that this enumeration procedure is correct and can be done in $O(2^k \cdot |V|)$ time.

A much more complicated modification is required for the enumeration procedure for FEEDBACK VERTEX SET as shown below.

5.1 Feedback Vertex Set

Schwikowski and Speckenmeyer [162] studied "classical algorithms" for enumerating minimal solutions of FEEDBACK VERTEX SET. Extending the algorithm given in Chapter 3, we can enumerate in $O(c^k \cdot m)$ time for a constant c—in compact form—*all* minimal feedback vertex sets of size at most k. Since, in general, there may be more than $O(c^k \cdot m)$ many such vertex sets, we list *compact representations* of all minimal feedback vertex sets.

For a graph $G = (V, E)$ a compact representation of some of its minimal feedback vertex sets is a set $\mathcal{C}$ of pairwise disjoint vertex subsets from V such that choosing exactly one vertex from every set in $\mathcal{C}$ results in a minimal feedback vertex set. Naturally, a set in $\mathcal{C}$ may also contain exactly one vertex; then this vertex is in every minimal feedback vertex set represented by $\mathcal{C}$. This notion of compact representations allows us to easily expand a compact representation to the set of minimal feedback vertex sets it represents and to enumerate the compact representations of all minimal feedback vertex sets within the claimed time bound.

Recall that in order to compress a size-$(k+1)$ feedback vertex set F to a size-k feedback vertex set F', the algorithm in Chapter 3 first tries all partitions of F into X and Y under the assumption that $X \subseteq F'$ and $F' \cap Y = \emptyset$. After deleting the vertices in X, data reduction rules are applied to reduce the instance with respect to its degree-1 and degree-2 vertices. This data reduction is based on the observation that there is always an optimal solution for FEEDBACK VERTEX SET without degree-1 and degree-2 vertices. Here, in contrast, in order to enumerate all minimal feedback vertex sets, some of which can contain degree-2 vertices, the degree-2 vertices cannot be reduced any more. Observe that the number of the vertices with degree higher than two in the graph after deleting the vertices in X is bounded by $14 \cdot |Y|$ as shown in Lemma 3.2. Moreover, since degree-1 vertices cannot contribute to a minimal feedback vertex set we can still eliminate all degree-1 vertices as in Chapter 3. Then, compared to finding one (minimal) feedback vertex set with at most k vertices, the only problem with enumeration is how to deal with degree-2 vertices. The solution to this problem is given in the proof of the following lemma.

Lemma 5.1. *Given a graph G and a feedback vertex set (fvs) F for G of size k, we can enumerate compact representations of all minimal feedback vertex sets for G having size at most k in $O(c^k \cdot m)$ time for a constant c.*

Proof. We show this lemma by giving a description of the enumeration procedure which constructs all compact representations in the claimed time bound.

Consider a minimal fvs F' with $F \neq F'$ and $|F'| \leq k$ which retains, in comparison to F, some vertices $X \subseteq F$ and replaces the vertices in $Y := F \setminus X$ by at most $|Y|$ new vertices from $V \setminus F$. Therefore, we begin with a branching into 2^k cases corresponding to all such partitions of F.

In each case the compact representation is initialized as $\mathcal{C} := \{\{v\} \mid v \in X\}$. As in the proof of Lemma 3.1, we delete the vertices in X and degree-1 vertices from G. Let $G' = (V', E')$ denote the resulting graph. We partition

$$V' = V'_{\geq 3} \cup V'_{=2} \cup Y$$

where $V'_{\geq 3}$ contains the vertices with degree at least 3 in $V' \setminus Y$ and $V'_{=2}$ the degree-2 vertices in $V' \setminus Y$. Observe that $|V'_{\geq 3}| \leq 14 \cdot |Y|$ due to Lemma 3.2. We then make a further branching into at most $\sum_{l=0}^{|Y|} \binom{14 \cdot |Y|}{l}$ cases; in each case, $\mathcal{C}$ is extended by l one-element sets $\{\{v\} \mid v \in V'_{\geq 3}\}$ for $0 \leq l \leq |Y|$.

In each of these cases, we delete the vertices in $V'_{\geq 3}$ which are added to $\mathcal{C}$ from G', and we reduce successively the degree-1 vertices as they cannot participate in a minimal fvs. Let $G'' = (V'', E'')$ denote the resulting graph and let $V''_{=2}$ denote the set of degree-2 vertices in $V'' \setminus Y$. If G'' is cycle-free, then we have a compact representation $\mathcal{C}$. Otherwise, the cycles in G'' can only be destroyed by deleting degree-2 vertices.

In G'', we identify every "maximal path" of vertices $v_1, v_2, \ldots, v_r$ where $v_i \in V''_{=2}$ for $i = 1, \ldots, r$, v_i is adjacent to v_{i+1} for $i = 1, \ldots, r-1$, and both v_1 and v_r are adjacent to vertices in $V'' \setminus V''_{=2}$. These maximal paths can be identified in $O(m)$ time by considering $G''[V''_{=2}]$. Clearly, a minimal feedback vertex set may contain at most *one* vertex from such a path. If it does contain one vertex from a path, then it does not matter which vertex is chosen. Therefore, since we are aiming for a compact representation of a minimal feedback vertex set, we save all these maximal paths in a set $\mathcal{P}$, i.e., $\mathcal{P} := \{\{v_1, v_2, \ldots, v_r\} \mid v_1, v_2, \ldots, v_r \text{ form a maximal path}\}$.

Having obtained $\mathcal{P}$ in this way, we now show $|\mathcal{P}| \leq 16 \cdot |Y|$. To this end, we define a bipartite graph B which has as vertices on one side the elements of $\mathcal{P}$, and on the other side the vertices in $V'' \setminus V''_{=2}$. An element of $\mathcal{P}$ has an edge to a vertex v in $V'' \setminus V''_{=2}$ iff one endpoint of its corresponding maximal path has an edge to v in G''. Note that there can be multiple edges between an element of $\mathcal{P}$ and a vertex in $V'' \setminus V''_{=2}$. Completing $\mathcal{C}$ to a compact representation of minimal feedback vertex sets having size at most k is now equivalent to selecting at most $|Y| - l$ many elements of $\mathcal{P}$ to eliminate all cycles in B.

Observe that, if $|\mathcal{P}| \geq |V'' \setminus V''_{=2}|$, then there exists at least one cycle in B. Thus, we can now infer that $|\mathcal{P}| \leq |V'' \setminus V''_{=2}| + (|Y| - l)$; otherwise, it would not be possible to remove all cycles in B by deleting $|Y| - l$ elements of $\mathcal{P}$. Therefore,

$$|\mathcal{P}| \leq |V'' \setminus V''_{=2}| + (|Y| - l) \leq |V'_{\geq 3}| + |Y| + |Y| = 16 \cdot |Y|.$$

Now, we make the last branching into $\sum_{j=0}^{|Y|-l} \binom{16|Y|}{j}$ cases; each case represents

a choice of at most $|Y| - l$ elements from $\mathcal{P}$. For a case where the resulting graph by deleting these chosen elements from B is cycle-free, we extend $\mathcal{C}$ by the chosen elements.

Altogether, we have at most 2^k partitions of F, and for each partition at most $\sum_{l=0}^{|Y|} \binom{14|Y|}{l}$ cases corresponding to the choices of vertices with degree more than two, and then for each possible choice of vertices in $V'_{\geq 3}$, we have further $\sum_{j=0}^{|Y|-l} \binom{16|Y|}{j}$ choices for degree-2 vertices in compact form. With $|F| \leq k$ and $Y \subseteq F$, the enumeration procedure can be done in time

$$O\left(\sum_{i=0}^{k} \binom{k}{i} \cdot \left(O(m) + \sum_{l=0}^{i} \binom{14 \cdot i}{l} \cdot \left(O(m) + \sum_{j=0}^{i-l} \binom{16 \cdot i}{j} \cdot O(m)\right)\right)\right)$$
$$= O\left(c^k \cdot m\right),$$

where c is a constant. □

Together with Theorem 4.2, we obtain the following result:

Theorem 5.1. *All minimal feedback vertex sets of size at most k can be enumerated in $O(c^k \cdot m)$ time for a constant c.*

5.2 Concluding Remarks

Fernau [76] has demonstrated that kernelizations (see Chapter 6) as well as search trees (see Chapter 9) are very useful techniques for parameterized enumeration. A special case of kernels with respect to parameterized enumeration, the so-called "full kernels", was introduced in [51]. Continuing and complementing their work, we showed that the iterative compression technique can also be used to derive parameterized enumeration algorithms. We provided an example by enumerating all size-at-most-k minimal feedback vertex sets. However, parameterized enumeration is still a relatively new research topic and provides promising research opportunities.

Note that the algorithm given here might return the same minimal feedback vertex set more than once. It would be interesting to consider the problem of enumerating all minimal solutions *without repetitions*. One obvious way is to compare all output minimal feedback vertex sets with each other in order to get rid of repetitions. However, this postprocessing would square the runtime. A better approach is desirable.

Part III

Data Reduction and Problem Kernels

Chapter 6

Basic Concepts and Ideas

Quoting Fellows [74, Page 9] from one of his surveys, "data reduction and kernelization rules are one of the primary outcomes of research on parameterized complexity." The usefulness of problem kernelization is tied to the concrete development of effective data reduction rules that work in polynomial time and, for instance, can be used in a preprocessing phase to shrink the given problem instance[1]. As a matter of practical experience, we are far from being allowed to expect that showing fixed-parameter tractability "automatically" brings along data reduction rules for a problem kernelization. In fact, many fixed-parameter tractable problems still await the development of effective data reduction rules.

In this chapter, we give a formal description of the data reduction method together with the concept of kernelization and present, with the help of several case studies, some general ideas for developing data reduction rules and proving problem kernels. Two more complicated examples are given in the next two chapters.

6.1 Data Reduction

Roughly speaking, the basic idea behind the data reduction method is, in polynomial time, to prune "trivially" solvable parts of a given NP-complete problem instance by applying some data reduction rules such that the given instance shrinks to a small "hard" part. A part of a given instance can be a subgraph of a given graph or a submatrix of a given matrix. For some NP-complete problems, after exhaustively applying the data reduction rules, the remaining "hard" parts of the given instances have sizes which, compared to the sizes of the original input instances, are small such that one can afford applying brute-force algorithms to solve them. If the size of a "hard" part is bounded from above by a function depending only on the parameter, we call such a "hard" part a *problem kernel*,

[1] Observe, however, that as a matter of theoretical [142] as well as practical experience [87], data reduction rules are not only useful in a preprocessing phase but should be applied again and again during the whole solution process.

for a definition of problem kernels see Section 6.2. For other problems, although we cannot mathematically show such a provable small problem kernel, it turns out that data reductions are very powerful for most instances from real applications as demonstrated by the striking examples RED-BLUE DOMINATING SET in context of optimizing the European railroad network [175, 176] and DOMINATING SET [5, 6, 10]. In both cases, experimental studies show a graph size shrinking of more than 90% and that two simple data reduction rules followed by simple brute-force search on small isolated components sufficed to optimally solve most real-world instances. Hence, it was suggested in [141] that, whether developing polynomial-time approximation, fixed-parameter, or purely heuristic algorithms solving hard problems, one should always take into consideration the data reduction method.

The central part of the data reduction method are the *data reduction rules.*

Definition 6.1. *Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. A* data reduction rule *is a mapping*

$$\phi : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}, \ \ (x, k) \mapsto (x', k'),$$

where

1. *ϕ is computable in time polynomial in $|x|$ and k.*
2. *$(x, k) \in L$ iff $(x', k') \in L$ with $|x'| \leq |x|$ and $k' \leq k$.*

Given a finite set $\Phi := \{\phi_1, \phi_2, \ldots, \phi_i\}$ of data reduction rules, we use $\Phi((x, k))$ to denote the instance after exhaustive application of the data reduction rules in Φ to (x, k) and we call $\Phi((x, k))$ the reduced *instance. Then, by the term of* data reduction process *we refer to replacing (x, k) by $\Phi((x, k))$.*

Note that, in order to get a polynomial-time data reduction process, we have to guarantee, on the one hand, that each data reduction rule runs in time polynomial in $|x|$ and k, and, on the other hand, that the number of applications of the data reduction rules in Φ is upper-bounded by a polynomial of $|x|$ and k.

There are two types of data reduction rules, *parameter-independent* and *parameter-dependent* ones. Parameter-independent data reduction rules can be applied even when there is no given parameter value. Experiments demonstrate that, in real-world applications, the most useful data reduction rules tend to be the parameter-independent data reduction rules as, for example, the ones given in the case of DOMINATING SET [5, 6, 10]. Formally, a parameter-independent data reduction rule is a polynomial-time computable mapping $\phi : \Sigma^* \to \Sigma^*$. Only few problems admit a provable problem kernel size with only parameter-independent rules. In contrast, the applicability of parameter-dependent reduction rules depends on the parameter value. Thus, these reduction rules are often hard to apply in the practice. However, the problem kernels of most problems are achieved with these reduction rules.

According to Definition 6.1, a general design scheme of data reduction processes consists of three steps:

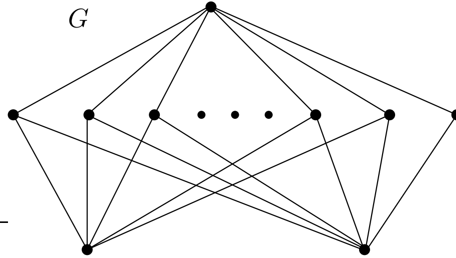

Figure 6.1: An FVS-instance irreducible with respect to the two data reduction rules: The graph G contains no degree-one or degree-two vertex. Deleting arbitrarily two of the three high-degree vertices can transform G into a tree. However, G may contain arbitrarily many degree-three vertices.

1. Identify the "trivially" solvable parts of the given problem instances;
2. develop data reduction rules for these parts;
3. prove the correctness and polynomial runtime of the rules.

Depending on the parts identified in the first step, the second and third steps can be highly technical as shown by the examples Cluster Editing (Chapter 7) and Multicut in Trees (Chapter 8). In the following, we give several application examples of the data reduction method.

6.1.1 Case Study 1: Feedback Vertex Set

The NP-complete Feedback Vertex Set (FVS) problem asks for a set F of at most $k \geq 0$ vertices such that each cycle of the given graph $G = (V, E)$ contains at least one vertex from F. Set F is called the *feedback vertex set*. Without loss of generality, we assume that G is connected and contains at least one vertex with degree three or higher.

There are two well-known data reduction rules for FVS: The easy instance parts identified in the first step are degree-one and degree-two vertices. Since none of the degree-one vertices can be in a cycle and, thus, they cannot be in an optimal feedback vertex set, they are deleted from the input graph. This is the first data reduction rule. It is obvious that, if there is a degree-two vertex in an optimal feedback vertex set, then it lies on a path with at least one degree-≥ 3 endpoint. Therefore, we can replace the degree-two vertex in the feedback vertex set by the degree-≥ 3 vertex, i.e., delete the degree-two vertex from it and then add the degree-≥ 3 vertex to it. The resulting set is clearly a feedback vertex set with the same cardinality. The data reduction rule with respect to a degree-two vertex is to delete it and to connect its two neighbors by an edge. These two rules are clearly correct and can exhaustively be applied in $O(|V|)$ time.

Remark: Observe that both rules are parameter-independent. FVS does not admit a problem kernel with only these two data reduction rules, as shown in

Figure 6.1. However, these two rules are crucial for the randomized algorithm by Becker et al. [19] and the fixed-parameter algorithm by Raman et al. [152]. The iterative compression algorithm in Chapter 3 uses also these two data reduction rules.

6.1.2 Case Study 2: Vertex Cover

The NP-complete Vertex Cover problem asks for a set C of at most $k \geq 0$ vertices in a given graph $G = (V, E)$ *covering* all edges in G, i.e., each edge in E has at least one endpoint in C.

The first data reduction rule for Vertex Cover is deleting all isolated vertices, i.e., the vertices with no incident edge. The second data reduction rule identifies a vertex v with more than k neighbors as an easy part of the instance. Such a vertex has to be in each vertex cover with at most k vertices; otherwise, one has to take more than k vertices to cover the edges incident to v. The data reduction rule deletes the vertices with more than k neighbors, recursively, and decreases parameter k by one after each deletion. If parameter k becomes negative, then the rule answers that the given graph has no vertex cover with at most k vertices. The runtime of the data reduction process with these two reduction rules is $O(|V| + |E|)$.

Remark: The applicability of the reduction rule by Buss and Goldsmith depends on the value of parameter k. One can show that there exists a problem kernel with at most $O(k^2)$ vertices after exhaustive application of this reduction rule. For more details on problem kernels, see the next section.

6.1.3 Case Study 3: Minimum Clique Cover

In Minimum Clique Cover (MCC), we seek for a set $\mathcal{C}$ of at most k cliques such that each edge of the input graph $G = (V, E)$ is contained in at least one clique in $\mathcal{C}$. This problem is NP-complete [123, 145] and cannot be approximated within $|V|^\epsilon$ for some $\epsilon > 0$ [132]. Among others, this problem has applications in applied statistics [88].

We give here two parameter-independent reduction rules for MCC and, in the next section, we show that there is a problem kernel for MCC using these rules. The first data reduction rule has isolated cliques as the identified easy parts of a given MCC instance. An isolated clique is a clique which has no connection to other parts of G. If an isolated clique contains only one vertex, we delete it from G; otherwise, we add this clique to $\mathcal{C}$ and decrease k by one. This rule is obviously correct and can be executed in $O(|E|)$ time.

The second data reduction rule considers two adjacent vertices u and v with $N(u)\backslash\{v\} = N(v)\backslash\{u\}$. This rule deletes arbitrarily one of u and v from G, say v. Concerning its correctness, we have to show that the new graph $G' := G \setminus \{v\}$ has a size-k clique cover iff G has a size-k clique cover.

The direction that, if G' has a size-k clique cover $\mathcal{C}$, then G has a size-k clique cover, is easy to show, since we can transform $\mathcal{C}$ into a size-k clique cover for G by adding v to all cliques in $\mathcal{C}$ which contain u. Suppose that we have a

size-k clique cover $\mathcal{C}$ for G. If all cliques in $\mathcal{C}$ containing either both of u and v or none of them, then we can get a size-k clique cover for G' by deleting all occurrences of v from the cliques in $\mathcal{C}$; otherwise, we modify each clique C in $\mathcal{C}$ which contains only v by replacing v by u. Due to $N(u) \setminus \{v\} = N(v) \setminus \{u\}$, C remains a clique. After deleting all occurrences of v in the cliques in $\mathcal{C}$, we have then a size-k clique cover for G'. This completes the correctness proof of the reduction rule.

The runtime of the second reduction rule is $O(|V| \cdot |E|)$: For each edge $\{u, v\} \in E$, we compare the neighborhoods of u and v, which can be done in $O(|V|)$ time. Thus, the data reduction process can be done in $O(|V| \cdot |E|)$ time.

Remark: Recently, Gramm et al. [87] gave further data reduction rules for MCC such as deleting an isolated vertex and then decreasing the parameter by one. However, to show a problem kernel for MCC (see Section 6.2.2), we need only these two rules.

6.2 Problem Kernel

As mentioned above, we can, for some problems, show that the remaining "hard" part after exhaustively applying the data reduction rules has a provably small size. This remaining part is then called *problem kernel*.

Definition 6.2. *Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem, Φ be a finite set of data reduction rules for L, $\Phi = \{\phi_1, \phi_2, \ldots, \phi_i\}$, and $(x', k') := \Phi((x, k))$ denote the reduced instance of (x, k) with $x \in \Sigma^*$ and $k \in \mathbb{N}$ after exhaustive application of the data reduction rules in Φ. We say that L admits a* problem kernel *if there exists an arbitrary computable function $g : \mathbb{N} \to \mathbb{N}$ such that $|x'| \leq g(k')$. We call $g(k')$ the* kernel size.

A data reduction process resulting in a problem kernel is called a *kernelization* process. It is clear that a computable parameterized problem L admitting a problem kernel is fixed-parameter tractable: By using $T_{\Phi((x,k))}$ to denote the run time of the kernelization process, we can solve L in $O(h((g(k), k)) + T_{\Phi((x,k))})$ time where $h((|x|, k))$ denotes the run time of an arbitrary brute-force algorithm solving L on instance (x, k). Cai et al. [33] show that the converse is also true.

Theorem 6.1 ([33]). *A parameterized problem L is in FPT iff L admits a problem kernel.*

It is clear that the kernel size has a significant impact on the overall runtime of the fixed-parameter algorithm combining a brute-force algorithm and the kernelization process. As small as possible problem kernels are desirable. Another benefit of kernelization is a speed-up method for depth-bounded search algorithms. Niedermeier and Rossmanith [142] showed an interesting technique to interleave kernelization processes with depth-bounded search trees (for more details on depth-bounded search trees see Part IV). More precisely, they proposed a speed-up method for depth-bounded search tree algorithms for problems admitting problem kernels.

Theorem 6.2 ([142]). *Let L be a parameterized problem solvable in $O(c^k \cdot n^{O(1)})$ time by a search tree algorithm. If L admits a problem kernel of size $g(k)$ with $g(k) < c^k$ and the kernelization process can be done in T_Φ time, then we can solve L in $O(c^k + T_\Phi)$ time.*

Showing problem kernels involves two tasks: developing data reduction rules and proving the existence of the kernel size function g in Definition 6.2. In Section 6.1, we have given some reduction rules for VERTEX COVER and MINIMUM CLIQUE COVER. Here, we show how to prove the existence of the functions g which upper-bound the sizes of the problem kernels of the two problems with respect to these data reduction rules.

6.2.1 Case Study 1: Vertex Cover

For a given VERTEX COVER instance with graph $G = (V, E)$ and parameter k, we exhaustively apply the following two data reduction rules:

- (DR 1) Delete all vertices with no incident edge;
- (DR 2) Add a vertex with more than k neighbors to the vertex cover C and delete this vertex and all its incident edges from G. Decrease parameter k by one. If k becomes negative, then stop and answer that the given graph has no size-k vertex cover.

The following lemma shows a problem kernel for VERTEX COVER with at most $k^2 + k$ vertices.

Lemma 6.1. *A* VERTEX COVER *instance reduced with respect to the two data reduction rules has a graph containing at most $k^2 + k$ vertices and k^2 edges.*

Proof. Let (G', k') be the reduced instance with $G' = (V', E')$ and $k' \leq k$. If G' has a size-k' vertex cover, there are at most k' vertices which are incident to all edges. Together with the fact that, due to (DR 2), each vertex in V' has at most k' incident edges, we get $|E'| \leq k'^2 \leq k^2$ and $|V'| \leq k^2 + k$, showing the claim. □

Remark: VERTEX COVER even allows for a much more sophisticated and stronger data reduction which leads to a problem kernel with at most $2k$ vertices. We refer to [138, 41, 2].

6.2.2 Case Study 2: Minimum Clique Cover

Given a graph $G = (V, E)$ and a parameter $k \geq 0$, we presented in Section 6.1.3 two data reduction rules for MINIMUM CLIQUE COVER (MCC):

- DR 1: Remove all connected components which are cliques from G. For each removed clique containing at least one edge, decrease k by one. If k becomes negative, then stop and answer that G has no clique cover of size k;

- DR 2: If, for two adjacent vertices u and v, $N(u) \setminus \{v\} = N(v) \setminus \{u\}$, then delete one of u and v from G.

To show a problem kernel for MCC we need the following well-known lemma. A proof of this lemma can be found in Halldórsson et al. [102, Lemma 2].

Lemma 6.2. *If a collection $\mathcal{C}$ of some subsets of a base set $S = \{s_1, s_2, \ldots, s_n\}$ "distinguishes" the elements of S, i.e., for every two elements s_i and s_j in S with $i \neq j$, there is at least one subset S' in $\mathcal{C}$ with $s_i \in S'$ and $s_j \notin S'$ or vice versa, then $|\mathcal{C}| \geq \lceil \log n \rceil$.*

Lemma 6.3. *A graph of an MCC instance that is reduced with respect to the two data reduction rules DR 1 and DR 2 contains at most $k \cdot 2^{k+1}$ vertices.*

Proof. Let $G' = (V', E')$ denote the reduced graph and $k' \leq k$. If G' has a size-k' clique cover $\mathcal{C}$, then every clique C in $\mathcal{C}$ covers at least one edge and, due to DR 1 and DR 2, there is at most one vertex w in C with $N(w) \setminus C = \emptyset$. Thus, there are some cliques in $\mathcal{C}$ covering the edges with one endpoint in C and the other not in C. Furthermore, due to DR 2, all vertices in C have distinct neighborhoods in $V' \setminus C$, namely,

$$\forall u \neq v \text{ with } u, v \in C: \; N(u) \setminus C \neq N(v) \setminus C.$$

This implies that, for every two vertices $u \neq v$ in C, there is at least one clique C' in $\mathcal{C}$ with $u \in C'$ and $v \notin C'$ or vice versa. By Lemma 6.2, $\mathcal{C}$ contains at least $\lceil \log |C| \rceil$ cliques, i.e., $k \geq |\mathcal{C}| \geq \lceil \log |C| \rceil$. Then, we have $|C| \leq 2^{k+1}$ and, thus, $|V'| \leq k \cdot 2^{k+1}$. □

Remark: MCC is one of the few problems known to admit problem kernels with only parameter-independent data reduction rules. It remains a challenging task to show a polynomial-size kernel for MCC.

6.3 Concluding Remarks

Data reduction rules and the concept of a problem kernel are one of the most important contributions of parameterized complexity theory to the design of efficient algorithms for NP-hard problems. To show the existence of problem kernels and to get them as small as possible is one of the most challenging tasks in fixed-parameter algorithmics. Many fixed-parameter tractable problems lack efficient data reduction procedures. Moreover, from a practical view, the data reduction rules, particularly the parameter-independent data reduction rules, should be implemented as a preprocessing for all kinds of algorithmic approaches attacking NP-hard problems. Finally, the connection of the kernelization algorithms and approximation algorithms remains an issue for future research.

Chapter 7

Cluster Editing

In this chapter, we present an efficient polynomial-time data reduction for the CLUSTER EDITING problem resulting in a problem kernel containing at most $O(k^2 + 4k)$ vertices. Later, this kernelization process will be interleaved with the depth-bounded search algorithm for CLUSTER EDITING as a speed-up method, see Chapter 10. We partly follow [86].

7.1 Problem Definition and Previous Results

The CLUSTER EDITING problem is defined as follows:

> **Input:** An undirected graph $G = (V, E)$, and an integer $k \geq 0$.
> **Task:** Find a set P of at most k vertex pairs, i.e., $P \subseteq V \times V$ and $|P| \leq k$, such that the graph $G' = (V, E')$ with $E' := E \ominus P$ consists of a disjoint union of cliques. (Adding the edges in $P \setminus E$ and deleting the edges in $E \cap P$ results in a disjoint union of cliques.)

A graph that consists of a disjoint union of cliques is called a *cluster graph*. The special version of CLUSTER EDITING where we can only delete edges from E is called CLUSTER DELETING. If we are only allowed to add edges, this problem can be trivially solved in linear time.

CLUSTER EDITING is motivated by biological applications of clustering gene expression data [165] and document clustering problems from machine learning [17]. These applications are based on the notion of a *similarity graph* whose vertices correspond to data elements and in which there is an edge between two vertices iff the similarity of their corresponding elements exceeds a predefined threshold. The goal is to obtain a cluster graph by as few edge modifications (i.e., edge deletions and additions) as possible.

Shamir et al. [163] showed that CLUSTER EDITING is NP-complete. The NP-completeness of CLUSTER EDITING, however, can already be extracted from work of Křivánek and Morávek [126] who studied more general problems in hierarchical tree clustering. Independently of Shamir et al.'s work, Bansal

et al. [17] initiated the research on "correlation clustering". It can be easily seen that an important special case of the general problem—also studied by Bansal et al.—is identical to CLUSTER EDITING. Bansal et al. mainly provide polynomial-time approximation results which partially have been improved by recent work [37, 54, 69]. Notably, the best known approximation factor for CLUSTER EDITING is 4 [37]; moreover, it is shown to be MaxSNP-hard (meaning that a polynomial-time approximation scheme (PTAS) is unlikely) [37].

A fixed-parameter algorithm with a runtime of $O(3^k \cdot |V|^4)$ for CLUSTER EDITING follows directly from a result of Cai [32]: A graph modification problem with a "goal" graph which can be characterized by a finite set of forbidden induced subgraphs is fixed-parameter tractable. For CLUSTER EDITING, the following lemma gives a forbidden subgraph characterization. A proof of this lemma can be found in [163].

Lemma 7.1. *A graph $G = (V, E)$ is a cluster graph iff there are no three vertices $u, v, w \in V$ which induce a P_3 in G, that is, a graph containing three vertices and two edges*

In graph theory a graph containing no induced P_3 is called "P_3-free". Recently, Damaschke [50] studied the enumeration version of CLUSTER EDITING.

7.2 Data Reduction Rules

Given a graph $G = (V, E)$ and a vertex pair $u, v \in V$, we use the term *common neighbor* of u and v to refer to a vertex $z \in V$ with $z \in N(u) \cap N(v)$. Similarly, a vertex z with $z \neq u$ and $z \neq v$ is a *non-common neighbor* of u and v if z is contained in exactly one of $N(u)$ and $N(v)$.

We present two reduction rules for CLUSTER EDITING. For each of them, we discuss its correctness and give the runtime which is necessary to execute the rule. In the next section, we show a problem kernel for CLUSTER EDITING that consists of at most $k^2 + 4k$ vertices and at most $k^3 + 4k^2 + k$ edges.

Although the following reduction rule also *adds* edges to the graph, we consider the resulting instances as *simplified*. The reason is that for every added edge, the parameter is decreased by one. In the following rule, it is implicitly assumed that, when an edge is added or deleted, parameter k is decreased by one.

Rule 1 For every pair of vertices $u, v \in V$:

1. If u and v have more than k common neighbors, then $\{u, v\}$ has to belong to E. If $\{u, v\}$ is not in E, we add it to E.
2. If u and v have more than k non-common neighbors, then $\{u, v\}$ cannot belong to E. If $\{u, v\}$ is in E, we delete it.
3. If u and v have both more than k common and more than k non-common neighbors, then the given instance has no size-k solution. □

Lemma 7.2. *Rule 1 is correct.*

Proof. **Case 1:** Vertices u and v have more than k common neighbors. If we did exclude $\{u,v\}$ from E, then we would have to, for every common neighbor z of u and v, delete at least one of the edges $\{u,z\}$ and $\{v,z\}$. This, however, would require at least $k+1$ edge deletions, a contradiction to the maximum of k edge modifications allowed.
Case 2: Vertices u and v have more than k non-common neighbors. If we did include $\{u,v\}$ in E, then we would have to, for every non-common neighbor z of u and v, edit edge $\{u,z\}$ or edge $\{v,z\}$ such that $z \in N(u) \cap N(v)$. Without loss of generality, assume that $z \in N(u)$ and $z \notin N(v)$. Then, we would have to either delete $\{u,z\}$ from E or to add $\{v,z\}$ to E. With at least $k+1$ non-common neighbors, this would require at least $k+1$ edge modifications which is not allowed.
Case 3: Vertices u and v have more than k common neighbors and more than k non-common neighbors. From the proofs for Case 1 and Case 2 it is clear that it would require more than k edge modifications both when including $\{u,v\}$ in E and when excluding $\{u,v\}$ from E. □

Note that Rule 1 applies to every vertex pair $\{u,v\}$ for which $|N(u) \cup N(v)| > 2k$.

Lemma 7.3. *A graph can in $O(|V|^3)$ time be transformed into a graph which is reduced with respect to Rule 1.*

Proof. It is clear that the check whether there exists a pair of vertices having more than k common or non-common neighbors can be done in $O(|V|^3)$ time: We examine all vertex pairs and, for each pair of vertices, all vertices which are adjacent to one of these two vertices. However, whenever we edit an edge, parameter k is decreased by one. Then, some vertex pairs to which Rule 1 could not be applied before the edge edition could have now more than k common or non-common neighbors. If we check applicability of Rule 1 after each edge edition, then this would imply a runtime of $O(k \cdot |V|^3)$. To get a runtime of $O(|V|^3)$, one can use some additional data structures and a preprocessing filling out these data structures before starting the data reduction process. After each edge edition during the data reduction process, these data structures help us in $O(|V|^2)$ time to determine where there is a vertex pair to which Rule 1 can be applied. More details on the data structures and the preprocessing can be found in [86, Lemma 2]. □

Note that the $O(|V|^3)$ runtime given here is only a worst-case bound and it is to be expected that the application of the rule is much more efficient in practice.

Rule 2 Delete the connected components which are cliques from the graph. □

The correctness of Rule 2 is straightforward. Computing the connected components of a graph and checking for cliques can easily be done in linear time:

Lemma 7.4. *Rule 3 can be executed in $O(|E|)$ time.* □

7.3 Problem Kernel

The following theorem shows that reducing a graph with respect to Rules 1 and 2 leads to a problem kernel consisting of at most $k^2 + 4k$ vertices for CLUSTER EDITING, which slightly improves the kernel size of at most $2k^2 + k$ vertices shown in [86, Theorem 1] for $k \geq 3$.

Theorem 7.1. CLUSTER EDITING *admits a problem kernel which contains at most $k^2 + 4k$ vertices and at most $k^3 + 4k^2 + k$ edges.*

Proof. Given an input graph $G = (V, E)$, we use $G' = (V', E')$ to denote the reduced graph with respect to Rules 1 and 2. Since, if G' is not a connected graph, then we can analyze every connected component separately, we assume that G' is connected. Let k be the minimum number of required edge modifications to transform G' into a cluster graph $G'' = (V', E'')$, namely k_a edge additions and k_d edge deletions. Since Rule 2 deletes all isolated cliques from G, we know $k > 0$. For the case $k = 1$ we have edited only one edge $\{u, v\}$. Due to Rule 1, vertices u and v have at most one common neighbor x and at most one non-common neighbor y. Besides u and v, vertices x and y cannot have any other neighbor; otherwise, we would have to make more edge editions. By the assumption that G' is connected and G' is reduced with respect to Rule 2, we obtain $|V'| \leq 4 < 5$ and $|E'| \leq 4 < 6$. We consider in the following only $k \geq 2$.

If $k_d = 0$, then G'' contains only one clique and at least two edges are inserted, say one is between vertices u and v. Due to Rule 1, vertices u and v have at most k common neighbors in G'. With at most k edges inserted, the only clique in G'' cannot contain more than $2k+2$ vertices, i.e., $|V'| \leq 2k+2 < k^2+4k$.

We assume now $k_d > 0$. Let $V'_C \subseteq V'$ denote the vertex set of a largest clique C in the G''. From $k_d > 0$ and the fact that G' is connected, we know that the transformation from G' to G'' deletes at least one edge between a vertex $u \in V'_C$ and a vertex $v \notin V'_C$. Let k'_d denote the number of the edges between v and the vertices in V'_C which are deleted by the transformation. Since we can delete at most k_d edges and since there are k'_d edges between v and the vertices in V'_C, G'' contains at most $k_d - k'_d + 2$ cliques, we have $|V'_C| \geq |V'|/(k_d - k'_d + 2)$.

Since the vertices of V'_C form a clique in G'' and at most k_a many edges are added in the transformation from G' to G'', there are at most k_a vertices in V'_C which are not adjacent to u in G'. Therefore, the number of non-common neighbors of u and v amounts to at least $|V'_C| - k_a - k'_d$. Since G' is reduced with respect to Rule 1, $|V'_C| - k_a - k'_d \leq k$. Combining the two inequalities, we

have

$$\begin{aligned} |V'| &\leq (k + k_a + k'_d) \cdot (k_d - k'_d + 2) \\ &= (k + k_a + k'_d) \cdot (k - k_a - k'_d + 2) \\ &= k^2 - (k_a + k'_d)^2 + 2(k + k_a + k'_d) \\ &\leq k^2 + 4k. \end{aligned}$$

The bound on the number of the edges in E' follows analogously: For the case that $k_d = 0$, we have, as shown above, that $|V'| \leq 2k + 2$ and $k_a = k$. Then, $|E'| \leq (|V'| \cdot (|V'| - 1))/2 - k = 2k^2 + 2k + 1 \leq k^3 + 4k^2 + k$. We have shown above that, if $k_d > 0$, then the largest clique C in G'' contains at most $k + k_a + k'_d$ many vertices, i.e., $|V'_C| \leq k + k_a + k'_d$. Therefore, the number of edges in C is bounded by $(k + k_a + k'_d)^2/2$. Since the number of cliques in G'' is bounded by $k_d - k'_d + 2$, $|E''| \leq (k_d - k'_d + 2)(k + k_a + k'_d)^2/2$. With k_a edge insertions and k_d edge deletions, we obtain

$$\begin{aligned} |E'| &\leq |E''| - k_a + k_d \leq \frac{(k_d - k'_d + 2)(k + k_a + k'_d)^2}{2} - k_a + k_d \\ &= \frac{(k - (k_a + k'_d))(k + k_a + k'_d)^2}{2} + (k + k_a + k'_d)^2 - k_a + k_d \\ &\leq \frac{(k^2 - (k_a + k'_d)^2)(k + k_a + k'_d)}{2} + (k + k_a + k'_d)^2 - k_a + k_d \\ &\leq k^3 + 4k^2 + k. \end{aligned}$$

Summarizing, the reduced graph contains at most $k^2 + 4k$ vertices and at most $k^3 + 4k^2 + k$ edges (otherwise, no solution exists). □

7.4 Concluding Remarks

Rule 1 plays a decisive role in the kernelization process of CLUSTER EDITING. Observe that this rule is parameter-dependent and require a worst-case running time of $O(|V|^3)$. Thus, an issue for future work is to investigate how to implement this rule more efficiently, for example, by using more complicated data structures. A further research subject with implementation and experimentation of the reduction rules would be to examine whether Rule 1, interleaved with the depth-bounded search tree algorithm for CLUSTER EDITING given in Chapter 10, only increases the administrative overhead instead of really speeding up the algorithm.

Note that for CLUSTER DELETING where only edge deletions are allowed there is no known problem kernel smaller than the one for CLUSTER EDITING. It is a challenge to develop data reduction rules for CLUSTER DELETING such that we can achieve a smaller kernel than the one for CLUSTER EDITING.

We conclude with a further open question: Does CLUSTER EDITING even allow for a problem kernel of linear size $O(k)$? For VERTEX COVER on general graphs [41] and DOMINATING SET on planar graphs [10] such results are known, but it seems hard to derive similar results in our setting.

Chapter 8

Multicut in Trees

In this chapter, based on some polynomial-time data reduction rules which appear to be of particular interest from an applied point of view, we show a problem kernel for Multicut in Trees by an intricate mathematical analysis. The description of the data reduction rules and the proof of the problem kernel follow parts of [97].

8.1 Problem Definition and Previous Results

Many hard network problems become easy when restricted to trees. There are, however, notable exceptions of important graph problems that remain hard even on trees. A well-known example is the Bandwidth Minimization problem restricted to trees of maximum vertex degree three, where it remains NP-complete [134]. In this work, we will study another graph problem that remains NP-complete when restricted to trees [82].

The problem is Multicut in Trees:

> **Input:** An undirected tree $T = (V, E)$, $n := |V|$, a collection H of h pairs of vertices in V, $H = \{(u_i, v_i) \mid u_i, v_i \in V, u_i \neq v_i, 1 \leq i \leq h\}$, and an integer $k \geq 0$.
> **Task:** Find a subset M of E with $|M| \leq k$ whose removal separates each pair of vertices in H.

Note that by removing edges a tree decomposes into subtrees forming a forest. Then, two vertices are *separated* if they are in different trees of the forest. An edge subset M of E as specified above is called a *multicut.* We refer to a pair of vertices $(u_i, v_i) \in H$ as a *demand path* P due to the fact that, in a tree, the path is uniquely determined by u_i and v_i.

Multicut in Trees was shown to be NP-complete and MaxSNP-hard even for an input tree being a star [82][1]. Whereas the latter implies that polynomial-

[1] More specifically, this special case is shown to be equivalent to Vertex Cover, also with respect to approximability [82].

time approximation schemes are out of reach, Garg et al. [82] gave a factor-2 approximation algorithm that also works for the more general case with edge weights. Călinescu et al. [48] provided a polynomial-time approximation scheme (PTAS) for finding unweighted multicuts in graphs with bounded degree and bounded treewidth.

See Costa et al. [47] for a recent survey on MULTICUT problems.

We need some special notation concerning networks (trees). We often *contract an edge* e. Let $e = \{v, w\}$ and let $N(v)$ and $N(w)$ denote the sets of neighbors of v and w, respectively. Then, contracting e means that we replace v and w by one new vertex x and we set $N(x) := (N(v) \cup N(w)) \setminus \{v, w\}$. Using an adjacency list representation of graphs, edge contraction can be done in constant time. We occasionally consider paths P_1 and P_2 in the tree and we write $P_1 \subseteq P_2$ when the vertex set (and edge set) of P_2 contains that of P_1.

8.2 Parameter-Independent Reduction Rules

In this section we give four parameter-independent data reduction rules which are of central importance for deriving a problem kernel for MULTICUT IN TREES as shown in Section 8.5. We can often observe that parameter-independent data reduction rules are very useful in practice as, for example, the ones given in the case of the NP-complete DOMINATING SET problem [6, 10]. We call a data reduction rule independent of the parameter k if it can be applied without any knowledge of the value of k. Four more parameter-dependent data reduction rules will be given in Section 8.3.

Idle Edge. If there is a tree edge with no demand path passing through it, then contract this edge.

Unit Path. If a demand path has length one, then the corresponding edge e has to be in M. Contract e and remove all demand paths passing through e and decrease the parameter k by one.

Dominated Edge. If all demand paths that pass through edge e_1 of T also pass through edge e_2 of T, then contract e_1.

Dominated Path. If $P_1 \subseteq P_2$ for two demand paths, then delete P_2.

Observe that only the Unit Path rule decreases the value of parameter k.

Lemma 8.1. *The above four reduction rules are correct and they can be executed in $O(h \cdot n^3 + h^3 \cdot n)$ worst-case time such that finally no more rules are applicable.*

Proof. The correctness of the Idle Edge and Unique Path rules is easy to observe. The Dominated Edge rule is correct since, if all demand paths that pass through edge e_1 also pass through edge e_2, then adding e_1 to M is never better than adding e_2 to M. The Dominated Path rule follows from the observation that if $P_1 \subseteq P_2$ for two demand paths, then each edge removal which destroys P_1 also destroys P_2.

Next, we estimate the runtime for each particular rule. Then, we estimate the maximum overall runtime of successive applications of these rules until none of them applies any more.

Idle Edge. During a depth-first traversal of the tree, we clearly can mark each edge e as to whether or not a path passes through e. Accordingly, e may be contracted. This is doable in $O(h \cdot n)$ time.

Unit path. Inspecting each demand path, this rule is executable in $O(h)$ time.

Dominated Edge. Basically, for each pair of edges in the tree we compare their corresponding sets of demand paths, i.e., the demand paths passing through these edges, respectively. Doing this for all of the $O(n^2)$ pairs, each comparison taking $O(h)$ time, we end up with $O(h \cdot n^2)$ time in total.

Dominated Path. Comparing all $O(h^2)$ pairs of demand paths, in each case we basically have to compare two paths of length $O(n)$, leading to $O(h^2 \cdot n)$ runtime.

Eventually, we have to estimate for each rule how often it may apply. Clearly, we have $O(n)$ possible applications for the first three rules. As to the Dominated Path rule, $O(h)$ is an upper bound for the possible number of applications. Altogether, we thus can conclude that after $O(h \cdot n^3 + h^3 \cdot n)$ worst-case runtime for applying the rules, none of them will be applicable any longer. □

Obviously, the runtime bound of Lemma 8.1 gives a very rough estimate. In particular, it is conceivable that the reduction rules will perform much better in practical implementations and tests. This is a typical observation also for other data reduction rules with relatively high polynomial worst-case runtimes, as, for example, was observed for the data reduction rules for Dominating Set [6, 10].

8.3 Parameter-Dependent Reduction Rules

In this section we introduce four more data reduction rules which are parameter-dependent. Thereby, we need several notations and we define two special cases of trees, *caterpillars* and *spiders of caterpillars*. In Section 8.5, as sort of a warm-up with helpful results for the general case, we will also show problem kernels for Multicut in these two special cases of trees.

8.3.1 Some Notation and Definitions

In Section 8.5 the bound on the size of the input tree $T = (V, E)$ (and, thus, also the set of demand pairs H) will be achieved by first partitioning the vertices of T into six disjoint sets and then giving for each of these vertex sets a bound on its size. For an undirected and unrooted tree T, we distinguish two sorts of vertices, *leaves* having only one incident edge and *inner vertices* having more than one incident edge. The sets of leaves and inner vertices are denoted by L and I, respectively. For an inner vertex v, we call the leaves (if existing) adjacent to it *v's leaves.*

The desired partition of V is then defined as follows:

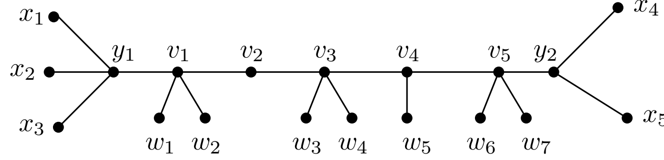

Figure 8.1: A caterpillar: There is no I_3-vertex and no L_3-leaf. Vertices y_1 and y_2 are the two I_1-vertices. In addition, $L_1 = \{x_1, x_2, x_3, x_4, x_5\}$, $I_2 = \{v_1, v_2, v_3, v_4, v_5\}$, and $L_2 = \{w_1, w_2, w_3, w_4, w_5, w_6, w_7\}$.

- $I_1 := \{ v \in I \mid |N(v) \cap I| \leq 1\}$;
  Observe that, if we delete all leaves from T, the vertices in I_1 become leaves in the resulting tree.
- $I_2 := \{ v \in I \mid |N(v) \cap I| = 2\}$;
  Note that $I_2 \neq \emptyset$ only if $|I_1| \geq 2$.
- $I_3 := \{ v \in I \mid |N(v) \cap I| \geq 3\}$;
  Set I_3 is empty iff, by deleting all leaves from T, the resulting tree is a path.
- $L_1 := \{ v \in L \mid N(v) \subseteq I_1\}$;
- $L_2 := \{ v \in L \mid N(v) \subseteq I_2\}$;
- $L_3 := \{ v \in L \mid N(v) \subseteq I_3\}$;

We use the terms L_i*-leaves* and I_i*-vertices* for $1 \leq i \leq 3$ in the obvious way.

The definitions of the special trees considered in the next sections—caterpillar and spider of caterpillars—are as follows.

Definition 8.1. *Given a tree $T = (V, E)$, we partition V as described above. A tree $T = (V, E)$ is a* caterpillar *if $|I_1| = 2$ and $|I_3| = 0$. Then, the inner vertices of I_2 form a path between the two vertices in I_1. We call this path the* backbone *of the caterpillar.*

Definition 8.2. *Given a tree $T = (V, E)$, we partition V as described above. A tree $T = (V, E)$ is a* spider of caterpillars *if $|I_3| = 1$. Then, the inner vertices of I induce a spider with the I_3-vertex as the center vertex. The paths induced by the I_2-vertices are called the* backbones *of the spider.*

Figure 8.1 and Figure 8.2 display examples for a caterpillar and a spider of caterpillar, respectively.

Finally, we define the "caterpillar component" of a tree as follows:

Definition 8.3. *Given a tree $T = (V, E)$, we partition V as described above. A* caterpillar component *of T is an induced subtree of T exclusively consisting of I_2-vertices and their L_2-leaves.*

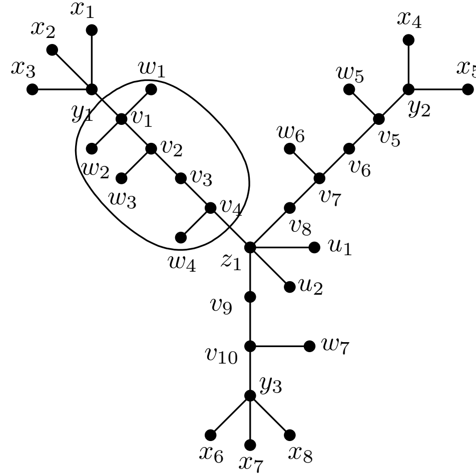

Figure 8.2: A spider of caterpillars: Vertex z_1 is the only I_3-vertex with $L_3 = \{u_1, u_2\}$. In addition, $I_1 = \{y_1, y_2, y_3\}$, $I_2 = \{v_1, \ldots, v_{10}\}$, $L_1 = \{x_1, \ldots, x_8\}$, and $L_2 = \{w_1, \ldots, w_7\}$. The *oval* depicts a maximal caterpillar component.

Note that a caterpillar component can be contained in other caterpillar components. We call a caterpillar component *maximal* if it is not contained in any other caterpillar component. See Figure 8.2 for an example of a maximal caterpillar component. Clearly, the set of all maximal caterpillar components of a tree is unique and no two maximal caterpillar components intersect each other. We say that an I_1-vertex or an I_3-vertex is *adjacent to a caterpillar component* if it is adjacent to an I_2-vertex of the component.

8.3.2 Parameter-Dependent Data Reduction Rules

In this subsection we extend our set of so far four reduction rules (Section 8.2) by four parameter-dependent rules. We need these rules to show the bound on the size of the reduced input tree, the problem kernel.

Disjoint Paths. If an instance of Multicut in Trees has more than k pairwise edge-disjoint demand paths, then there is no solution with parameter value k.

Overloaded Edge. If more than k length-two demand paths pass through an edge e, then contract e, remove all demand paths going through e, and decrease the parameter k by one.

Overloaded Caterpillar. If there are $k+1$ demand paths (v, u_1), (v, u_2), ..., (v, u_{k+1}) such that vertices $u_1, \ldots, u_{k+1}$ belong to the same caterpillar component that does not contain v, then (one of) the longest of these demand paths can be deleted.

Overloaded L_3-Leaves. If there are $k+1$ demand paths (v, u_1), (v, u_2), ..., (v, u_{k+1}) such that vertices $u_1, \ldots, u_{k+1}$ are all L_3-leaves of an I_3-vertex u, then remove all these demand paths and add a new demand path between v and u.

Lemma 8.2. *The above four reduction rules are correct and they can be executed, together with the four rules in Section 8.2, in polynomial time such that finally no further rule is applicable.*

Proof. Disjoint Paths. The correctness of this reduction rule is obvious since, for every two edge-disjoint demand paths, we need to add at least two edges to M. The maximum edge-disjoint paths problem can be solved for trees in polynomial time [82].

Overloaded Edge. The correctness of this rule follows from the fact that if edge e were not contracted, then one would need to remove more than k edges in order to cut all length-two demand paths passing through e. Note that the Overloaded Edge rule is "similar in spirit" to the removal of high-degree vertices in the second data reduction rule for VERTEX COVER in Section 6.1.2. It can be clearly done in $O(h \cdot n)$ time.

Overloaded Caterpillar. In order to cut more than k demand paths by removing only k edges one has to remove an edge that is contained in at least two demand paths. Moreover, if there are $k+1$ demand paths between a vertex u and $k+1$ distinct vertices in a caterpillar component which does not contain u, then every edge which is a part of the caterpillar component and is passed by at least two of the $k+1$ demand paths has to be an edge of the backbone of the caterpillar component. Then, we have to remove at least one backbone edge. Observe that the set of the backbone edges passed by a longest demand path is a superset of the set of backbone edges passed by other k demand paths. This longest demand path is always cut by removing backbone edges to cut other k demand paths and, hence, it can be omitted. Since there can be $O(n^2)$ caterpillar components, one for each vertex pair, this rule can be executed in $O(n^3 \cdot h)$ time.

Overloaded L_3-Leaves. In order to cut these more than k demand paths by removing only k edges one has to remove at least one edge on the path between u and v. Then, cutting these demand paths is equivalent to cutting a demand path between u and v. This rule can clearly be done in $O(h \cdot n)$ time, since there are at most h demand paths starting at a vertex.

Together with the polynomial runtime of the four rules in Section 8.2, we get the polynomial runtime for all these rules. □

8.4 Some Observations on Reduced Instances

With the data reduction rules given in Sections 8.2 and 8.3.2, we arrive at the following observations on a reduced instance of MULTICUT IN TREES which are very useful for showing a problem kernel for MULTICUT IN TREES.

Definition 8.4. *We call an instance of* Multicut in Trees reduced *when none of the eight given data reduction rules applies.*

Without loss of generality, we assume that the reduced tree instance has at least three vertices.

Lemma 8.3. *In a reduced instance, each I_1-vertex has at least two L_1-leaves adjacent to it.*

Proof. Consider an I_1-vertex u of the reduced instance. It has at least one L_1-leaf. Suppose that u has only one L_1-leaf called v. Since the instance has at least three vertices, there is another vertex $w \neq v$ adjacent to u. By the assumption that u has only one L_1-leaf, w is an inner vertex. Due to the Idle Edge rule there must be a demand path starting at v. Furthermore, because of the Unit Path rule all demand paths going through edge $\{u, v\}$ have to go through edge $\{u, w\}$ as well. This, however, means that the Dominated Edge rule could be applied to $\{u, v\}$, a contradiction to the fact that the given instance is reduced. □

Lemma 8.4. *In a reduced instance, for each L_1-leaf v of an I_1-vertex u, there exists a demand path between v and another L_1-leaf of u.*

Proof. Assume that there is an L_1-leaf v adjacent to u with $u \in I_1$ and that there is no demand path between v and other L_1-leaves of u. Note that by Lemma 8.3 I_1-vertex u has at least two L_1-leaves. Since the instance is reduced, due to the Idle Edge rule there must be a demand path starting at v. Moreover, the Unit Path rule implies that each demand path starting at v then also has to pass an edge different from $\{u, v\}$. This implies that u has a uniquely determined inner vertex w adjacent to it and all demand paths starting at v also pass $\{u, w\}$. But then the Dominated Edge rule would apply to edge $\{u, v\}$, a contradiction to the fact that the instance is reduced. □

Lemma 8.5. *In a reduced instance, there are at most k edge-disjoint demand paths.*

Proof. The claim follows directly from the Disjoint Paths rule. □

Lemma 8.6. *In a reduced instance, there are at most k^2 length-2 demand paths.*

Proof. The claim follows from the fact that there are at most k edge deletions allowed and, due to the Overloaded Edge rule, deleting one edge can destroy at most k length-2 demand paths. □

Lemma 8.7. *In a reduced instance, there can be at most $2k^2$ L_1-leaves.*

Proof. The claim directly follows from Lemma 8.4 and Lemma 8.6. □

Lemma 8.8. *In a reduced instance, there can be at most k I_1-vertices and at most $k - 1$ I_3-vertices.*

Proof. Lemma 8.3 and Lemma 8.4 imply that for each I_1-vertex, there is at least one length-2 demand path between two of its L_1-leaves. Moreover, the length-2 demand paths for different I_1-vertices are pairwise edge-disjoint. Then, by Lemma 8.5, there can be at most k I_1-vertices. Furthermore, consider the subgraph T' of the input tree T that is induced by the inner vertices of T. It is clear that T' is a tree and the leaves of T' correspond one-to-one to the I_1-vertices of T. Since, in a tree with k leaves, there are at most $k-1$ inner vertices having at least three neighbors, it is easy to derive that $|I_3| \leq k-1$. □

Now, with Lemma 8.7 and Lemma 8.8, it "only" remains to show that the sizes of sets I_2, L_2, and L_3 of a reduced MULTICUT IN TREES instance can be bounded by a function in k. To this end, we need the following two lemmas which are decisive for showing the size upper-bounds for L_3 and $I_2 \cup L_2$, respectively.

Lemma 8.9. *For each I_3-vertex u in a reduced instance, each of its L_3-leaves is the starting point of at least two demand paths which pass through two distinct neighbors of u.*

Proof. Consider an L_3-leaf v of u. If only one demand path starts at v, then either the Unit Path rule or the Edge Domination rule would apply to edge $\{u, v\}$. If all demand paths starting at v passed only through one neighbor $w \neq v$ of u, then the Edge Domination rule would apply to edges $\{u, v\}$ and $\{u, w\}$. This is a contradiction to the fact that the input instance is reduced. □

Lemma 8.10. *1. In a reduced instance, an I_2-vertex v having no L_2-leaf adjacent to it has to be a starting point of at least two demand paths, passing through two distinct inner vertices adjacent to v.*

2. In a reduced instance, for an I_2-vertex v with some L_2-leaves adjacent to it, each of these L_2-leaves has at least two demand paths passing through two distinct neighbors of v.

Proof. 1. Consider an I_2-vertex v with two adjacent inner vertices u and w. If there is no demand path starting at v and passing through u, then all demand paths passing through edge $\{u, v\}$ also pass through edge $\{v, w\}$ and, hence, the Edge Domination rule would apply. This contradicts the fact that the input instance is reduced. If there is no demand path starting at v and passing through w, an analogous argument applies.

2. Consider an I_2-vertex v with two adjacent inner vertices u and w where v has r L_2-leaves $w_1, w_2, \ldots, w_r$. Note that due to the Unit Path rule all demand paths have length at least two. If there is only one demand path starting at w_i, $1 \leq i \leq r$, then clearly the Edge Domination rule applies to the edge $\{v, w_i\}$. The Edge Domination rule also applies to $\{v, w_i\}$ when all demand paths starting at w_i either pass through edge $\{u, v\}$, or $\{v, w\}$, or $\{v, w_j\}$ for $i \neq j$. □

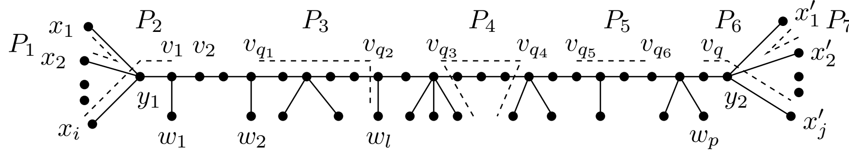

Figure 8.3: An instance of Multicut in Caterpillars. There are q I_2-vertices, $v_1, \ldots, v_q$, and p L_2-leaves, $w_1, \ldots, w_p$. The backbone of this caterpillar is the path between v_1 and v_q. The dashed lines denote seven edge-disjoint demand paths, $P_1, \ldots, P_7$.

8.5 Problem Kernel

In the following, we prove a problem kernel for Multicut in Trees by giving an upper bound on the size of a reduced input tree. Recall that a parameterized problem such as Multicut in Trees is said to have a problem kernel if, after the application of the data reduction rules in polynomial time, the resulting instance (T, H) with parameter k has size $g(k)$ for a function g depending only on k. In order to simplify the presentation, we adopt a stepwise manner. That is, first we show a bound for a caterpillar, then for a spider of caterpillars, and, finally, for general trees. More precisely, in the case of caterpillars where there is neither an I_3-vertex nor a L_3-leaf, we show how to bound the size of $I_2 \cup L_2$. In the case of spiders of caterpillars where there is only one I_3-vertex, we present the basic idea for showing the size bound for L_3. The problem kernel size for general trees follows by combining the arguments developed for the first two cases.

8.5.1 Problem Kernel for Caterpillars

With the observations made in Section 8.4, we show the problem kernel for Multicut in Trees when the input tree is restricted to be a caterpillar. Multicut in Caterpillars is also NP-complete, even if the tree vertices have at most five neighbors [94, 117].

Consider a reduced instance with a caterpillar $T = (V, E)$, i.e., there are no I_3-vertices, no L_3-leaves, and there are exactly two I_1-vertices, y_1 and y_2, as illustrated in Figure 8.3. Since the number of L_1-leaves is bounded by $2k^2$ (Lemma 8.7), in order to give a bound on the size of V it suffices to show that $|I_2| + |L_2|$ is bounded by a function of k.

We assume that the inner vertices of the caterpillar are ordered on a line, the first is y_1, the last is y_2, and the I_2-vertices in between are ordered from left to right in ascending order of their indices as shown in Figure 8.3. An I_2-vertex v_i is to the right of another I_2-vertex v_j if $i > j$. Furthermore, we use H_{I_2} to denote the set of the demand paths in H which pass through at least one I_2-vertex. We define the "right backbone endpoint" of a demand path in H_{I_2} as the I_2-vertex with highest index among the I_2-vertices that the demand path

passes through. The "left backbone endpoint" is defined symmetrically. In Figure 8.3, $H_{I_2} = \{P_2, P_3, P_4, P_5, P_6\}$. The left backbone endpoint of demand path P_3 is v_{q_1} and the right backbone endpoint of P_3 is v_{q_2}.

By Lemma 8.5, there can be at most k edge-disjoint demand paths in T. In the following, we show that there is a maximum cardinality set of edge-disjoint demand paths with some special properties. These special properties are useful for giving a bound on the size of the reduced caterpillar.

Lemma 8.11. *In polynomial time one can find a maximum cardinality set of edge-disjoint demand paths $\mathcal{P} := \{P_1, P_2, \ldots, P_l\}$ with $l \leq k$ which has the following two properties.*

Property (1). *One demand path is between two L_1-leaves of y_1 and one is between two L_1-leaves of y_2. Let P_1 and P_l denote these two demand paths; then we have $\{P_1, P_l\} \subseteq (\mathcal{P} \setminus H_{I_2})$.*

Property (2). *If the paths in $\mathcal{P} \cap H_{I_2}$ are ordered in ascending order of the indices of their left backbone endpoints, i.e., for $P_i, P_j \in (\mathcal{P} \cap H_{I_2})$ with $i < j$, P_i's left backbone endpoint is to the left of P_j's left backbone endpoint, then, for each $P_i \in (\mathcal{P} \cap H_{I_2})$ with $1 < i < l$,*

- *there is no other demand path $P \in (H_{I_2} \setminus \mathcal{P})$ such that P is edge-disjoint to all paths in $\mathcal{P} \setminus \{P_i\}$ and P_i's right backbone endpoint is to the right of P's right backbone endpoint;*
- *there is no other demand path $P \in (H_{I_2} \setminus \mathcal{P})$ such that P is edge-disjoint to all paths in $\mathcal{P} \setminus \{P_i\}$, P and P_i have the same right backbone endpoint, and P_i's left backbone endpoint is to the left of P's left backbone endpoint.*

Proof. Since a maximum cardinality set of edge-disjoint demand paths can be found in polynomial time [82], we only need to show how to, in polynomial time, modify an arbitrary maximum cardinality set of edge-disjoint demand paths $\mathcal{P}$ such that it fulfills the above properties.

Property (1). By Lemma 8.4, there always exists for an I_1-vertex a demand path between two of its L_1-leaves. Without loss of generality, assume that there is a demand path P between x_1 and x_2, two L_1-leaves of y_1. If $\mathcal{P}$ contains no demand path between two of y_1's L_1-leaves, i.e., $P \notin \mathcal{P}$, then there must be a demand path P' in $\mathcal{P}$ passing through one of the edges $\{x_1, y_1\}$ and $\{x_2, y_1\}$; otherwise, P would be edge-disjoint to all demand paths in $\mathcal{P}$, a contradiction to the maximality of $\mathcal{P}$. Moreover, P' cannot end in y_1 since T is reduced with respect to the Unit Path rule. Therefore, P' has to pass the edge $\{y_1, v_1\}$. Then, replacing P' by P in $\mathcal{P}$, the resulting set remains a maximum cardinality set of edge-disjoint demand paths.

Property (2). For each $P_i \in (\mathcal{P} \cap H_{I_2})$, we can in $O(h \cdot n)$ time find all demand paths $P \in (H_{I_2} \setminus \mathcal{P})$ which are edge-disjoint to all paths in $\mathcal{P} \setminus \{P_i\}$, where h denotes the number of demand paths in H. Let v_i^l and v_i^r denote P_i's left and right backbone endpoints. For each of these paths P with v^l and v^r

denoting P's left and right backbone endpoints, to check whether v_i^r is to the right of v^r or $v_i^r = v_r$ and v_i^l is to the left of v^l can be done in constant time. If there exists such a demand path P for P_i, replace P_i by P; the demand paths in $\mathcal{P}$ remain pairwise edge-disjoint. □

Based on a maximum cardinality set of edge-disjoint demand paths $\mathcal{P}$ as given in Lemma 8.11, we prove the main theorem of this subsection.

Theorem 8.1. Multicut in Caterpillars *has a problem kernel which consists of a caterpillar containing at most $O(k^{k+1})$ vertices.*

Proof. Suppose that we have computed a maximum cardinality set of edge-disjoint demand paths $\mathcal{P}$ as described in Lemma 8.11. We assume that P_1 is between x_1 and x_2, two L_1-leaves of y_1, and P_l is between x_1' and x_2', two L_1-leaves of y_2. Note that there can be more than one path in $\mathcal{P}$ between two L_1-leaves of y_1 (or y_2). However, it will be clear from the following analysis that we can derive a better bound on the size of the caterpillar if there is more than one path in $\mathcal{P}$ between L_1-leaves of y_1 (or y_2). Therefore, we assume that none of $P_2, \ldots, P_{l-1}$ is between two L_1-leaves of y_1 or y_2. We use v_{l_i} and v_{r_i} to denote the left and right backbone endpoints of demand path P_i with $2 \leq i \leq l-1$, respectively. Furthermore, we partition the I_2-vertices together with their L_2-leaves into $2l-1$ sets and bound from above the size of each set. These sets are

$$
\begin{aligned}
A_1 &:= \{ v_j \mid 1 \leq j \leq l_2 \} \cup \{ \text{their } L_2\text{-leaves} \}; \\
A_2 &:= \{ v_j \mid l_2 < j \leq r_2 \} \cup \{ \text{their } L_2\text{-leaves} \}; \\
A_3 &:= \{ v_j \mid r_2 < j \leq l_3 \} \cup \{ \text{their } L_2\text{-leaves} \}; \\
A_4 &:= \{ v_j \mid l_3 < j \leq r_3 \} \cup \{ \text{their } L_2\text{-leaves} \}; \\
&\vdots \\
A_{2l-2} &:= \{ v_j \mid l_{l-1} < j \leq r_{l-1} \} \cup \{ \text{their } L_2\text{-leaves} \}; \\
A_{2l-1} &:= \{ v_j \mid r_{l-1} < j \leq q \} \cup \{ \text{their } L_2\text{-leaves} \}.
\end{aligned}
$$

Informally, the sets with odd indices contain the I_2-vertices which are not on any demand path in $\mathcal{P}$ together with the left backbone endpoints of these demand paths, while the sets with even indices contain the remaining I_2-vertices. Note that some of these sets can be empty since two consecutive demand paths can share an endpoint. In particular, if P_2 (or P_{l-1}) starts at an L_1-leaf and ends at v_1 (or v_q), then $A_2 = \emptyset$ (or $A_{2l-2} = \emptyset$).

First, consider the vertices in A_1. By Lemma 8.10, each I_2-vertex with no L_2-leaf has a demand path starting at it and going to its left, and each L_2-leaf in A_1 has a demand path starting at it and going to the left of the adjacent I_2-vertex. However, a demand path starting at a vertex v in A_1 and ending at a vertex left to it cannot end at a vertex in A_1; otherwise, this demand path would be edge-disjoint to all demand paths in $\mathcal{P}$, which contradicts the maximality of $\mathcal{P}$. With the same argument, this demand path cannot end at y_1

or one of $x_3, \dots, x_i$. Thus, the other endpoint of the demand path can only be x_1 or x_2. Since T is reduced, there can be at most $2k$ demand paths starting at x_1 and x_2 and ending at one of the I_2-vertices and the L_2-leaves (due to the Overloaded Caterpillar rule). Thus, there can be at most $2k$ I_2-vertices without L_2-leaf and L_2-leaves in A_1. Since there are at most as many I_2-vertices with L_2-leaves as there are L_2-leaves, we can conclude that

$$|A_1| \leq 4k. \tag{8.1}$$

This analysis works analogously for $A_2, A_3, \dots, A_{2l-1}$. The demand paths starting at a vertex in A_2 and going to its left cannot end at an A_2-vertex; otherwise, we have a demand path P which is edge-disjoint to P_1 and P_3 and which has either a right backbone endpoint to the left of v_{r_2} or a left backbone endpoint to the right of v_{l_2}, which contradicts the fact that $\mathcal{P}$ has Property (2) in Lemma 8.11. Then, the demand paths starting at a vertex in A_2 and going left can have only the vertices in A_1, y_1, or $x_1, \dots, x_i$ as the other endpoint. For a vertex v in A_1, consider the demand paths starting at v and going right and ending at some I_2-vertices or their L_2-leaves. Since all I_2-vertices to the right of v together with their L_2-leaves induce a caterpillar component of T and v is outside this caterpillar component, then, using the fact that T is reduced with respect to the Overloaded Caterpillar rule, there can be at most k demand paths starting from v and ending at some vertices to the right of it. Therefore, with $|L_1| \leq 2k^2$ (Lemma 8.7), we get

$$|A_2| \leq k \cdot (|\{x_1, x_2, \dots, x_i\} \cup \{y_1\}| + |A_1|) \leq k \cdot (2k^2 + 1 + |A_1|).$$

Analogously, we have a bound on $|A_r|$ for an arbitrary r with $3 \leq r \leq 2l - 1$,

$$|A_r| \leq k \cdot (2k^2 + 1 + \sum_{j=1}^{r-1} |A_j|). \tag{8.2}$$

Therefore,

$$
\begin{aligned}
\sum_{j=1}^{2l-1} |A_j| &= \sum_{j=1}^{2l-2} |A_j| + |A_{2l-1}| \\
&\overset{(8.2)}{\leq} \sum_{j=1}^{2l-2} |A_j| + k \cdot (2k^2 + 1 + \sum_{j=1}^{2l-2} |A_j|) \\
&\leq (k+1) \cdot (\sum_{j=1}^{2l-2} |A_j| + 2k^2 + 1) \\
&\overset{(8.2)}{\leq} (k+1) \cdot (\sum_{j=1}^{2l-3} |A_j| + k \cdot (2k^2 + 1 + \sum_{j=1}^{2l-3} |A_j|) + 2k^2 + 1) \\
&= (k+1)^2 \cdot (\sum_{j=1}^{2l-3} |A_j| + 2k^2 + 1) \\
&\leq (k+1)^{2l-2} \cdot (|A_1| + 2k^2 + 1) \\
&\overset{(8.1)}{\leq} (k+1)^{2l-2}(4k + 2k^2 + 1) \\
&= O(k^{2l}).
\end{aligned}
$$

However, this bound can be improved if we take into account the symmetry of the caterpillar structure: Observe that the analysis for $|A_{2l-1}|$ can be done in the same way as for $|A_1|$, the analysis for $|A_{2l-2}|$ can be done in the same way as for $|A_2|$, and so on. Therefore, the bound on the number of I_2-vertices and L_2-leaves is as follows:

$$
\begin{aligned}
|I_2| + |L_2| &= \sum_{j=1}^{2l-1} |A_j| \\
&= \sum_{j=1}^{l} |A_j| + \sum_{j=l+1}^{2l-1} |A_j| \\
&\overset{(8.2)}{\leq} (k+1) \cdot (\sum_{j=1}^{l-1} |A_j| + 2k^2 + 1) \\
&\qquad\qquad +(k+1) \cdot (\sum_{j=l+2}^{2l-1} |A_j| + 2k^2 + 1) \\
&\leq (k+1)^2 \cdot (\sum_{j=1}^{l-2} |A_j| + 2k^2 + 1) \\
&\qquad\qquad +(k+1)^2 \cdot (\sum_{j=l+3}^{2l-1} |A_j| + 2k^2 + 1) \\
&\leq (k+1)^{l-1} \cdot (|A_1| + 2k^2 + 1) \\
&\qquad\qquad +(k+1)^{l-2} \cdot (|A_{2l-1}| + 2k^2 + 1) \\
&\overset{(8.1)}{\leq} (k+1)^{l-1}(4k + 2k^2 + 1) + (k+1)^{l-2}(4k + 2k^2 + 1) \\
&= O(k^{l+1}).
\end{aligned}
$$

Since there are at most k edge-disjoint paths in $\mathcal{P}$, that is, $l \leq k$, $|I_2|+|L_2| = O(k^{k+1})$. Together with $|L_1| \leq 2k^2$, $|I_1| = 2$, and $|I_3| = |L_3| = 0$, we have the claimed problem kernel size. ☐

8.5.2 Problem Kernel for Spiders of Caterpillars

The next special case of a tree, a spider of caterpillars T (also see Figure 8.2), has exactly one I_3-vertex u which is also the central vertex of the spider induced by the inner vertices. There are at most k I_1-vertices due to Lemma 8.8 and thus the number of maximal caterpillar components is bounded from above by k. Each of these maximal caterpillar components is adjacent to u and to one I_1-vertex. We call the subgraph of T that consists of a maximal caterpillar component, its adjacent I_1-vertex, and the L_1-leaves of this I_1-vertex a *semi-caterpillar*. The backbone of a semi-caterpillar then means the path induced by the I_2-vertices in the maximal caterpillar component.

In the following we adapt the analysis in the proof of Theorem 8.1 to show the upper-bound on the size of T. Recall that the proof of Theorem 8.1 is heavily based on a special maximum cardinality set of edge-disjoint demand paths as described in Lemma 8.11. Therefore, we first need to show that, in a

spider of caterpillars T, we can also find such special maximum cardinality sets of edge-disjoint demand paths.

Lemma 8.12. *For each of the semi-caterpillars of a spider of caterpillars T, one can in polynomial time find a maximum cardinality set $\mathcal{P} := \{P_1, P_2, \ldots, P_l\}$ of edge-disjoint demand paths passing through only edges of this semi-caterpillar which has the following two properties.*

Property (1). *Let y denote the only I_1-vertex in this semi-caterpillar. Then, one demand path in $\mathcal{P}$ is between two L_1-leaves of y. Let P_1 denote this demand path; then we have $P_1 \in (\mathcal{P} \setminus H_{I_2})$.*

Property (2). *If the paths in $\mathcal{P} \cap H_{I_2}$ are ordered in ascending order of the indices of their left backbone endpoints, i.e., for $P_i, P_j \in (\mathcal{P} \cap H_{I_2})$ with $i < j$, P_i's left backbone endpoint is to the left of P_j's left backbone endpoint, then, for each $P_i \in (\mathcal{P} \cap H_{I_2})$ with $1 < i < l$,*

- *there is no other demand path $P \in (H_{I_2} \setminus \mathcal{P})$ passing through only edges of this semi-caterpillar such that P is edge-disjoint to all paths in $\mathcal{P} \setminus \{P_i\}$ and P_i's right backbone endpoint is to the right of P's right backbone endpoint;*
- *there is no other demand path $P \in (H_{I_2} \setminus \mathcal{P})$ passing through only edges of this semi-caterpillar such that P is edge-disjoint to all paths in $\mathcal{P} \setminus \{P_i\}$, P and P_i have the same right backbone endpoint, and P_i's left backbone endpoint is to the left of P's left backbone endpoint.*

Proof. Observe that a semi-caterpillar is a subgraph of a caterpillar. Therefore, the proof of Lemma 8.11 can be easily adapted to prove this lemma. □

We can then extend Theorem 8.1 to spiders of caterpillars.

Theorem 8.2. Multicut in Spiders of Caterpillars *has a problem kernel which consists of a spider of caterpillars containing at most $O(k^{2k+1})$ vertices.*

Proof. For each semi-caterpillar of a spider of caterpillars T, after computing a maximum cardinality set $\mathcal{P}$ of edge-disjoint demand paths as described in Lemma 8.12, we can bound from above the size of this semi-caterpillar by $O(k^{2l})$ with $l = |\mathcal{P}|$ by using the arguments in the proof of Theorem 8.1. Note that, since a semi-caterpillar does not have the symmetrical structure of a caterpillar, we can only give the bounds for each set $A_1, A_2, \ldots, A_{2l-1}$ one-by-one from A_1 to A_{2l-1}. Therefore, $|I_2 \cup L_2| = O(k^{2k})$.

It remains to give a bound on $|L_3|$. Now, let u denote the only I_3-vertex. By Lemma 8.6, the number of L_3-leaves that have a demand path of length 2 starting at them is bounded by $2k^2$. Thus, we omit such L_3-leaves from further consideration. At each of the remaining L_3-leaves starts at least one demand path which ends at one vertex of the semi-caterpillars of T. From the Overloaded L_3-Leaves rule we know that at an arbitrary vertex v, there can start at most k

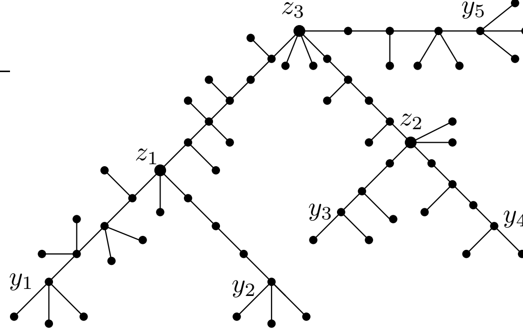

Figure 8.4: An example of a general tree: $I_1 = \{y_1, \ldots, y_5\}$ and $I_3 = \{z_1, z_2, z_3\}$. The tree is rooted at z_3. Vertex z_1 has maximum depth among the I_3-vertices.

demand paths which end at some L_3-leaves of u. Thus, with $|L_1| \leq 2k^2$, $|I_1| \leq k$, and $|I_2 \cup L_2| = O(k^{2k})$, we have

$$|L_3| \leq k \cdot |I_1 \cup I_2 \cup L_1 \cup L_2| = O(k^{2k+1}).$$

Altogether, the size of T is bounded by $O(k^{2k+1})$ and we have the claimed problem kernel size. □

8.5.3 Problem Kernel for General Trees

Based on the results of Sections 8.5.1 and 8.5.2, we have now all results and techniques in place to develop a problem kernel for MULTICUT IN TREES. For general trees T, there can be more than one I_3-vertex. We assume that there is at least one I_3-vertex in T and root T at an arbitrary I_3-vertex. Consider an I_3-vertex having maximum depth in the now rooted tree among all I_3-vertices. Observe that this I_3-vertex together with its leaves and all adjacent maximal caterpillar components which are not adjacent to any other I_3-vertices, i.e., the subtree of T rooted at this I_3-vertex, induces a structure similar to a spider of caterpillars. With this observation, we process all I_3-vertices in a bottom-up manner and, for each I_3-vertex, we give a bound on the size of the subtree rooted at it. See Figure 8.4 for an example.

Theorem 8.3. MULTICUT IN TREES *has a problem kernel which consists of a tree containing at most $O(k^{3k})$ vertices.*

Proof. First, consider an I_3-vertex u with maximum depth among all I_3-vertices, for instance, z_1 in Figure 8.4. The subtree of T rooted at u, denoted by $T[u]$, can be seen as a spider of caterpillars with u as the center vertex. Moreover, each L_3-leaf of u has at least one path starting at it and ending at a vertex of $T[u]$ (Lemma 8.9). Following from the analysis in Section 8.5.2,

$$|T[u]| = O(k^{2l_u+1}), \tag{8.3}$$

where l_u denotes the number of maximum edge-disjoint demand paths using only edges of $T[u]$.

Then, consider the maximal caterpillar component C between u and its I_3-*parent* v, the first I_3-vertex on the path from u to the root. In Figure 8.4, z_3 is the I_3-parent of both z_1 and z_2. Recall that, in Section 8.5.2, we bounded the size of a maximal caterpillar component of a spider of caterpillars based on the fact that the caterpillar component is adjacent to an I_1-vertex that has at most $2k^2$ L_1-leaves, i.e., the maximal caterpillar component is in a semi-caterpillar. Here, the maximal caterpillar component C between u and v is not adjacent to any I_1-vertex. However, the analysis in the proof of Theorem 8.1 can be easily extended to deal with a caterpillar component adjacent to an I_3-vertex that is the root of a subtree with bounded size. We can treat C as a caterpillar component adjacent to an I_1-vertex with as many L_1-leaves as the size of the subtree. Then, we partition the vertices of C as in the proof of Theorem 8.1 into $A_1, A_2, \dots, A_{2l_C-1}$, where l_C denotes the number of maximum edge-disjoint demand paths using only edges of C. The bound on the size of A_1 is then $k \cdot |T[u]|$ since each vertex in A_1 has to be the start vertex of at least one demand path ending at a vertex of $T[u]$ (Lemma 8.10). Then, $|A_1| \leq k \cdot |T[u]|$, $|A_2| \leq k \cdot (|A_1| + |T[u]|)$, and so on. With the size bound on $T[u]$, we have

$$|C| \overset{(8.3)}{=} O(k^{2l_u+1+2l_C}). \tag{8.4}$$

In the next step, we consider the subtree $T[v]$ rooted at u's I_3-parent v. Accordingly, we call all I_3-vertices that have v as their I_3-parent v's I_3-*children*, i.e., u is an I_3-child of v. Let $u_1, \dots, u_s$ denote v's I_3-children with $u = u_1$. Then, subtree $T[v]$ can be divided into the following disjoint subtrees:

- the subtrees $T[u_1], \dots, T[u_s]$ rooted at v's I_3-children,
- the caterpillar components $C_1, \dots, C_s$ between v and its I_3-children,
- the caterpillar components $C'_1, \dots, C'_r$ between v and the I_1-vertices that have v as their I_3-parent,
- and the star induced by v and its L_3-leaves.

In Figure 8.4, the tree T is rooted at z_3 which is the I_3-parent of z_1 and z_2. Here, the disjoint subtrees of T by dividing T at z_3 are the subtrees $T[z_1]$ and $T[z_2]$, the caterpillar components between z_3 and z_1 and between z_3 and z_2, the caterpillar component between z_3 and y_5, and the star consisting of z_3 and its L_3-leaves.

When arriving at v in the course of the bottom-up process, we have already the size bounds on all $T[u_i]$ and C_i for $1 \leq i \leq s$. In order to show that $T[v]$ has bounded size, it remains to give size bounds on $C'_1, \dots, C'_r$ and the set of v's L_3-leaves, respectively. Since each of $C'_1, \dots, C'_r$ with the adjacent I_1-vertex forms a semi-caterpillar, we have

$$|C'_1 \cup \dots \cup C'_r| = O(k^{2l_{C'}}) \tag{8.5}$$

as shown in Section 8.5.2, where $l_{C'}$ denotes the number of edge-disjoint demand paths using only the edges of $C'_1, \ldots, C'_r$.

Let L_3^v denote the set of v's L_3-leaves. By Lemma 8.9, each L_3^v-leaf has at least one demand path starting at it and ending at one vertex in $T[v] \setminus (L_3^v \cup \{v\})$. Thus, using the Overloaded L_3-Leaves rule, we get

$$|L_3^v| \leq k \cdot |T[v] \setminus (L_3^v \cup \{v\})|. \tag{8.6}$$

Furthermore, $T[v] \setminus (L_3^v \cup \{v\})$ is the union of $T[u_1], \ldots, T[u_s]$, $C_1, \ldots, C_s$, and $C'_1, \ldots, C'_r$. Let $l_1 = l_{u_1} + l_{u_2} + \cdots + l_{u_s}$ denote the number of edge-disjoint demand paths passing only the edges of $T[u_1], \ldots, T[u_s]$, and let $l_2 = l_{C_1} + l_{C_2} + \cdots + l_{C_s}$ denote the number of edge-disjoint demand paths passing only the edges of $C_1, \ldots, C_s$. We have

$$\begin{aligned}
|T[v]| &= \sum_{i=1}^{s} |T[u_i]| + \sum_{i=1}^{s} |C_i| + \sum_{j=1}^{r} |C'_j| + |L_3^v| + 1 \\
&\overset{(8.6)}{\leq} (k+1) \cdot (\sum_{i=1}^{s} |T[u_i]| + \sum_{i=1}^{s} |C_i| + \sum_{j=1}^{r} |C'_j|) + 1 \\
&\overset{(8.3),(8.4),(8.5)}{=} (k+1) \cdot (O(k^{2l_1+1}) + O(k^{2l_1+2l_2+1}) + O(k^{2l_{C'}})) + 1 \\
&= O(k^{2l_v+2}),
\end{aligned} \tag{8.7}$$

where l_v denotes the number of edge-disjoint demand paths in $T[v]$ and $l_v \geq l_1 + l_2 + l'_C$.

Finally, at the root r of T, we have then $l_r \leq k$. Starting from an I_3-vertex with maximum distance to the root r of T during the bottom-up process, we can encounter at most k I_3-vertices. Therefore, at the root r, we get $|T[r]| = O(k^{2l_r+k})$ by (8.7) and, thus, $|T[r]| = O(k^{3k})$. This gives the claimed problem kernel size. □

8.6 Concluding Remarks

In this chapter, we have an example for a problem kernel of size exponential with respect to parameter k (more precisely, size $O(k^{3k})$) where it seems hard to show a polynomial- or even linear-size problem kernel. At first glance, this seems a little disappointing because the size of the problem kernel exceeds the size of the search tree $O(2^k)$ shown in Chapter 9. However, first, one has to take into account that MULTICUT IN TREES is a more general problem than VERTEX COVER, already making problem kernelization a harder thing to do. Secondly, the developed data reduction rules are of comparable simplicity as the ones developed by Weihe [175, 176] for his problem such that we nevertheless may expect a strong practical impact of our rules. Observe that all our bounds are purely worst-case results (relying on very special or even artificial cases that may very rarely occur) and the practical experiences for real-world or other test data sets may be much better. Thirdly, our extensive worst-case analysis of the

problem kernel size and the discovered "worst-case structures" may help to spot future points of attack for improved kernelization strategies or to get a better understanding of what really makes the problem so hard. Fourthly, we consider it as a worthwhile task of also purely mathematical interest to show upper size bounds on problem kernels.

Clearly, the immediate challenge is to improve the problem kernel size significantly. We felt that this will be a particularly hard task when only using the given set of data reduction rules.

Part IV

Search Trees Based on Forbidden Subgraphs

Chapter 9

Basic Concepts and Ideas

Recall that graph modification problems seek for a minimum size set of edge editions (insertions and deletions) or/and vertex deletions such that the resulting graphs have some specified properties. Many of these graph properties have nice characterizations in terms of forbidden subgraphs. Such graph modification problems can be also formulated as to get rid of the forbidden subgraphs by a minimum number of modifications. For example, consider the NP-complete INDEPENDENT SET problem [80]: For a given graph $G = (V, E)$ and an integer $\ell \geq 0$, we seek for a set V' of vertices with $V' \subseteq V$ and $|V'| \geq \ell$ such that the subgraph of G induced by V' contains no edge. The vertex subset V' is called an *independent set*. Obviously, the forbidden subgraph for a graph with an independent vertex set is an edge. Therefore, we can formulate INDEPENDENT SET as a graph modification problem:

> **Input**: A graph $G = (V, E)$ and an integer $k \geq 0$;
> **Task**: Find a set C of at most k vertices such that, after deleting the vertices in C and their incident edges from G, the vertex set of the remaining graph is an independent set.

It is easy to observe that this graph modification problem is equivalent to the VERTEX COVER problem. Further examples include FEEDBACK VERTEX SET (Chapter 3), GRAPH BIPARTIZATION (Chapter 4), and many others.

Most graph modification problems with properties characterized by forbidden subgraphs are NP-complete. To obtain "efficient" fixed-parameter algorithms for these problems, the most commonly used technique so far is based on *depth-bounded search trees*. The reason for this is that this technique is easy to describe, to implement, and to understand. For applications of depth-bounded search trees to non-graph problems, we refer to [11, 83, 89, 91, 166].

In this chapter we give a brief introduction to forbidden subgraph characterizations and depth-bounded search trees. Then, we present two simple case studies, VERTEX COVER and MULTICUT IN TREES, to illustrate how to design search tree algorithms based on forbidden subgraphs. In Chapters 10 and 11 we give two more sophisticated applications of the method.

9.1 Forbidden Subgraph Characterizations

Many graph properties allow for characterizations in terms of forbidden subgraphs.

Definition 9.1. *Let $\mathcal{F}$ be a collection of graphs. A graph property* Π *can be characterized by the* forbidden induced subgraphs *in $\mathcal{F}$ iff each graph having* Π *contains no graph in $\mathcal{F}$ as induced subgraph. A graph having* Π *is called $\mathcal{F}$-*free.

Forbidden subgraph characterizations are a well-studied topic in the graph theory literature. It is out of the scope of this thesis to describe the deep impacts so-called finite-basis characterizations and well-quasi ordering theory had on the genesis of parameterized complexity theory. We refer to the monograph [65] for a proper account on this. Forbidden subgraph characterizations are valuable in various ways. First, many containment relations between graph classes follow immediately from forbidden subgraph characterizations, for more details we refer to [30]. Second, these characterizations also help with the recognition of graphs having specified properties and lead to the polynomial-time solvability of the recognition problems in a straightforward way. For example, cographs contain no P_4—a path consisting of four vertices—as an induced subgraph [30]. We can easily recognize a cograph $G = (V, E)$ by examining whether there are four vertices in V inducing a P_4. This can be done by a straightforward $O(|V|^4)$ time algorithm. Actually, the recognition of cographs can be done in linear time [46]. Third, we can also make use of forbidden subgraph characterizations for proving NP-completeness results of the corresponding graph modification problems. As two examples, the NP-completeness of Closest Tree Power with Bounded Degree and Closest Leaf Power by Edge Insertions are shown based on two reductions making use of the forbidden subgraph characterizations for these two problems [61, 59]. For the definitions of the two problems see Chapter 11. Finally, making use of the forbidden subgraph characterizations is the decisive ingredient of polynomial-time approximation and fixed-parameter algorithms for the corresponding graph modification problems. For example, Natanzon et al. [137] gave constant-factor polynomial-time approximation algorithms for edge modification problems with forbidden subgraph characterizations on bounded degree graphs. We will discuss how to derive fixed-parameter algorithms based on forbidden subgraph characterizations in more detail in Section 9.3.

For some graph properties such as "edgeless" in the Independent Set problem, the forbidden subgraph characterizations follow directly from the definitions while it can become technically demanding to show forbidden subgraph characterizations in other cases. For example, the forbidden subgraph characterization of so-called 4-leaf powers—see Chapter 11 for the definition of leaf powers—requires a substantial amount of technical expenditure, including an intricate algorithm [58].

9.2 Depth-Bounded Search Trees

The basic idea behind the depth-bounded search tree technique is to organize the systematic and exhaustive exploration of the search space in a tree-like manner. More precisely, given an instance (I, k) of a parameterized problem, search tree algorithms replace (I, k) by a finite set $\mathcal{C}$ of smaller instances (I_i, k_i) with $1 \leq i \leq |\mathcal{C}|$, $|I_i| < |I|$, and $k_i < k$ specified by some *branching rules*. If a smaller instance (I_i, k_i) satisfies none of a set of *termination conditions*, then the algorithm recursively applies this replacing procedure to (I_i, k_i). The algorithm terminates when the replacing procedure is no longer applicable, i.e., at least one of the termination conditions is satisfied.

The recursive applications of the replacing procedure can be illustrated in a tree structure: The original instance (I, k) is placed at the root of the tree. The smaller instances in $\mathcal{C}$ are the children of the root. If a smaller instance (I_i, k_i) for $1 \leq i \leq |\mathcal{C}|$ satisfies one of the termination conditions, then it represents a leaf of the tree; otherwise, (I_i, k_i) has several children corresponding to the instances which are specified to replace (I_i, k_i) by the branching rules. This tree is called *search tree*. For each (I_i, k_i) in $\mathcal{C}$ specified by the branching rules for (I, k), we have $k_i < k$. Thus, the depth of the search tree is bounded from above by a function of the parameter and we obtain a depth-bounded search tree.

In the following, we separately describe termination conditions and branching rules and provide the mathematical tools to compute the size of a search tree. The size of a search tree is crucial for the runtime of the corresponding search tree algorithm.

Termination conditions For most parameterized problems, there are two termination conditions: Dealing with the current instance (I, k), the first one is whether $k \leq 0$. Since new smaller instances specified by the branching rules have a smaller parameter value, it makes no sense to further apply the replacing procedure to instances with a parameter value equal to zero or less. The second termination condition is whether the set of smaller instances specified by the branching rules for (I, k) is empty. If one of the two conditions is satisfied, instance (I, k) represents a leaf of the search tree. The determination of whether the termination conditions are satisfied is easy for most problems: It usually can be done in time polynomial in $|I|$.

Branching rules Branching rules are the most important ingredient of a search tree algorithm.

Definition 9.2. *Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. A* branching rule *is a mapping Γ from $(I, k) \in \Sigma^* \times \mathbb{N}$ to a subset of $\Sigma^* \times \mathbb{N}$ denoted by $\Gamma((I, k)) := \{(I_1, k_1), \ldots, (I_l, k_l)\}$ where*

1. *Γ is computable in time polynomial in $|I|$,*
2. *$(i, k) \in L \Longleftrightarrow \exists\, (I_i, k_i) \in \Gamma((I, k))$ with $(I_i, k_i) \in L$, and*

3. $k_i < k$ *for* $1 \leq i \leq l$*.*

The branching vector δ *of a branching rule* Γ *is defined as:*

$$\delta(\Gamma) := (k - k_1, k - k_2, \ldots, k - k_l).$$

The instances in $\Gamma((I, k))$ are called *branching subcases* and we call the application of the replacing procedure to an instance (I, k) *branching into subcases.*

Intuitively, a branching rule points out the "right" direction to search by guaranteeing that (I, k) is a "Yes"-instance iff there is at least one smaller instance in $\Gamma((I, k))$ which is a "Yes"-instance. For some problems, we can have several branching rules. Based on a case distinction on (I, k) we decide which one should be used. Several branching rules associated with case distinctions on (I, k) often lead to search trees with smaller size than those with only one branching rule.

Size of search trees The size of a search tree is governed by homogeneous, linear recurrences with constant coefficients. It is well-known how to solve them and the asymptotic solution is determined by the roots of the characteristic polynomial (see, e.g., Kullmann [125] for more details).

We use T to denote the search tree. If a branching rule Γ, considering an instance (I, k), has $\delta(\Gamma) = (k - k_1, k - k_2, \ldots, k - k_l)$ as its branching vector, then this vector $\delta(\Gamma)$ corresponds to the recurrence

$$T_k = T_{k_1} + \cdots + T_{k_l}$$

where T_k denotes the size of the subtree of T rooted at (I, k). Let $d_i := k - k_i$ for $1 \leq i \leq l$ and $d := \max\{d_1, d_2, \ldots, d_l\}$. The recurrence corresponds to the *characteristic polynomial*

$$z^d = z^{d-d_1} + \cdots + z^{d-d_l}.$$

If α is a root of the characteristic polynomial which has maximum absolute value and is positive, then T_k is α^k up to a polynomial factor. We call α the *branching number* that corresponds to the branching vector $\delta(\Gamma) = (d_1, \ldots, d_l)$. Moreover, if α is a single root, then $T_k = O(\alpha^k)$; all branching numbers that occur in this thesis are single roots. The size of the search tree is therefore $O(\beta^k)$ where β is the largest branching number among them of all branching rules.

In summary, using $p(|I|)$ to denote the time for determining the termination conditions and computation of the branching rules for an instance (I, k), where $p(x)$ denotes a polynomial of x, and using β^k to denote the size of the search tree, the runtime of a bounded-depth search tree algorithm is then equal to $O(\beta^k \cdot p(|I|))$. The runtime is asymptotically dominated by the size of the search tree. Therefore, developing branching rules with smaller branching numbers is the most important point to speed up search tree algorithms.

As already mentioned in Chapter 6, another general speed-up method is the *interleaving* technique introduced by Niedermeier and Rossmanith [142]: The

kernelization process is invoked not only at the beginning of the depth-bounded search tree algorithm, but before every determination of termination conditions and before the computation of the branching rules for an instance (I, k). It allows an improved runtime of the search tree algorithm. Suppose that we have a search tree algorithm with a runtime of $O(\beta^k \cdot p(|I|))$. Let T_D denote the runtime of the kernelization process resulting in a kernel size bounded from above by a function $g(k)$ only depending on parameter k. Then, it is clear that the time for determining the termination conditions and the computation of branching rules is only dependent on $g(k)$, i.e., $p(g(k))$. If both functions p and g are polynomial in k, then interleaving the branching steps and the problem kernel reduction yields a runtime of $O(\beta^k + T_D)$.

9.3 Search Trees Based on Forbidden Subgraphs

When designing depth-bounded search tree algorithms for graph modification problems transforming an input graph G into a graph characterized by a set $\mathcal{F}$ of forbidden subgraphs, we distinguish two cases of forbidden subgraph characterizations.

If the set $\mathcal{F}$ is finite, then we can easily solve the corresponding graph modification problem by a depth-bounded search algorithm: For every subgraph F in $\mathcal{F}$, we give a branching rule Γ_F. Given an instance (G, k), Γ_F determines whether G is $\{F\}$-free. If so, it specifies an empty set; otherwise, a subgraph G' of G which is equal to F is identified and Γ_F specifies a set of smaller instances that consist of a graph transformed from G by editing edges or deleting vertices of G'. The termination conditions are whether the parameter k is equal to zero or less and whether the graph G is $\mathcal{F}$-free. The resulting algorithm, in each search tree node, finds a forbidden subgraph from $\mathcal{F}$ in G and, based on the corresponding branching rule, branches into (finitely many) subcases that lead to a destruction of the considered forbidden induced structure.

Cai [32] was the first making the observation that finite forbidden subgraph characterizations lead to the fixed-parameter tractability of the corresponding graph modification problems. Moreover, he concluded that the runtime of such search tree algorithms is $O(N^k \cdot |G|^{N+1})$ for any graph property with finite forbidden subgraph characterization. Herein, $|G|$ denotes the size of the input graph, k denotes the number of allowed modifications and N denotes the maximum size over all forbidden subgraphs in $\mathcal{F}$.[1] The determination of termination conditions and the computation of branching rules can be done in $O(|G|^{N+1})$ time. We remark that this result by Cai can be used as a classification tool and is of mainly theoretical interest. As we will show with Cluster Editing in Chapter 10, more efficient bounded-search search tree algorithms can be achieved in more specific cases by making a refined analysis of the forbidden subgraphs.

[1] When dealing with vertex deletion problems, N is equal to the maximum number of vertices of the forbidden subgraphs; it is set to the maximum number of vertex pairs of the forbidden subgraphs for edge modification problems.

The scheme described above is quite simple but it is not applicable to forbidden subgraph characterizations which have infinitely many forbidden subgraphs: If $\mathcal{F}$ is infinite, then we have infinitely many branching rules. This makes it hard for an algorithmic approach since we would have to check infinitely many cases. Many graph properties have infinitely many forbidden subgraphs such as "being a forest" (all cycles are forbidden subgraphs) and chordality (all induced cycles with length four or higher are forbidden). Dealing with graph modification problems with graph properties characterized by infinitely many forbidden subgraphs, there is no general scheme known like the one for finite forbidden subgraph characterizations. Fixed-parameter algorithms solving such problems are based on methods exploring problem-specific properties which seemingly cannot be generalized to other problems. The iterative compression algorithm for FEEDBACK VERTEX SET (seeking for a "goal" graph being forest) in Chapter 3 and the algorithm given by Kaplan et al. [114] for the MINIMUM FILL-IN problem where one seeks for a minimum set of edge insertions to transform a graph into a chordal graph provide two examples.

However, the above scheme can still be applied to a subclass of the graph modification problems with infinite sets of forbidden subgraphs. This subclass contains problems whose forbidden subgraph characterizations admit *data reductions to base subsets*. A base subset $\mathcal{B}$ is a finite subset of the infinite forbidden subgraph set $\mathcal{F}$. A data reduction to a base subset $\mathcal{B}$ is a data reduction process running in time polynomial in the size of the input graph G and satisfying the following condition:

> Graph G is not $\mathcal{F}$-free iff the reduced graph G' contains a subgraph from $\mathcal{B}$.

With such data reductions to base subsets, we modify the above scheme slightly: We construct the branching rules as described above for all subgraphs in $\mathcal{B}$ instead of $\mathcal{F}$. The termination conditions remain the same. The only difference is that, every time before the computation of the branching rules, we do the polynomial-time data reduction by taking into account that the solution for G can be easily computed based on the solution for G'. By the "iff"-condition of the data reduction, the correctness of this modified scheme is obvious. Concerning the runtime of the resulting search tree algorithm, we only have to take into account the runtime of the data reduction which gives a polynomial factor for each search tree node. The overall runtime is still dominated by the size of the search tree and, therefore, we have a fixed-parameter algorithm. We will give an example of this modified scheme in the case study with MULTICUT IN TREES (Section 9.4.2).

9.4 Two Case Studies

In this section we give two example applications of depth-bounded search trees based on forbidden subgraph characterizations, one dealing with a finite set of forbidden subgraphs and one with infinitely many forbidden subgraphs.

9.4.1 Case Study 1: Vertex Cover

Recall that, in Vertex Cover, we seek for a set C of at most k vertices covering all edges in graph $G = (V, E)$. In other words, the removal of the vertices in C results in an edgeless graph whose forbidden subgraph is clearly an edge. Thus, the "goal" graph of Vertex Cover has a finite forbidden subgraph characterization. More specifically, $\mathcal{F}$ only contains one subgraph simply formed by one edge.

According to the design scheme for finite forbidden subgraph characterization, we have two termination conditions:

1. $k \leq 0$?
2. G contains no edge?

The only branching rule first determines whether G is edgeless and, if not, finds one. Let $\{u, v\}$ denote this edge. We know that at least one of u and v has to be in C to destroy this edge. Then, this branching rule specifies two smaller instances, one of them putting u into C and deleting u and u's incident edges from G, the other doing the same operation on v. More precisely, the two smaller instances are as follows:

- $G_1 = (V_1, E_1)$ with $V_1 := V \setminus \{u\}$ and $E_1 := E \setminus \{\{u, x\} \mid x \in V\}$ and $k_1 := k - 1$;
- $G_2 = (V_2, E_2)$ with $V_2 := V \setminus \{v\}$ and $E_2 := E \setminus \{\{v, x\} \mid x \in V\}$ and $k_2 := k - 1$.

The resulting search tree algorithm, given an instance (G, k), first determines whether the termination conditions are satisfied. If not, it computes the branching rule and produces two smaller instances by replacing (G, k) by (G_1, k_1) and (G_2, k_2), respectively. The termination conditions and the branching rules are then recursively applied to (G_1, k_1) and (G_2, k_2). We get a vertex cover of size at most k if the algorithm terminates in a search tree node with the second termination condition is satisfied but the first not.

Concerning the size of the search tree, every application of the branching rule produces two smaller instances. In particular, the value of parameter k is decreased by one in both smaller instances. Hence, the depth of the search tree is bounded from above by k. The branching vector is clearly $(1, 1)$ and we have

$$T_k = T_{k-1} + T_{k-1}.$$

With $d_1 = k - k_1 = 1$, $d_2 = k - k_2 = 1$, and $d := \max\{d_1, d_2\}$, the characteristic polynomial is then

$$z^d = z^{d-d_1} + z^{d-d_2}$$

which gives the only root $\alpha = 2$. Then, the size of the search tree is upper-bounded by $O(2^k)$ and the runtime of the algorithm is $O(2^k \cdot |V|)$.

Remark: With refined case distinctions, more sophisticated branching rules, and the interleaving technique [142], the runtime of the search tree algorithm is improved to $O(1.2745^k k^4 + k|V|)$ [36].

9.4.2 Case Study 2: Multicut in Trees

MULTICUT IN TREES asks for, given an input tree $T = (V, E)$, a *multicut set* M of at most k edges whose removal separates all vertex pairs given in a set H of $h \geq 0$ pairs of vertices in V. We call the uniquely determined path in T between vertices u and v of a vertex pair $(u, v) \in H$ the *demand path* of (u, v). For paths P_1 and P_2, we write $P_1 \subseteq P_2$ when the vertex set (and edge set) of P_2 contains that of P_1.

All paths with two endpoints u and v for which there is a vertex pair $(u, v) \in H$, i.e., all demand paths, are the forbidden subgraphs characterizing the goal graph of MULTICUT IN TREES. Since H is a part of the problem input and a demand path can be arbitrarily long, this characterization is an infinite forbidden subgraph characterization. We show that this characterization admits a data reduction to a base subset.

The base subset contains only two subgraphs, demand paths of length one or two. The data reduction consists of three rules. We say that we *contract* an edge $\{u, v\}$ by replacing this edge and its two endpoints by a new vertex w with $N(w) := (N(u) \cup N(v)) \setminus \{u, v\}$.

Idle Edge. If there is a tree edge with no demand path passing through it, then contract this edge.

Dominated Edge. If all demand paths that pass through edge e_1 of T also pass through edge e_2 of T, then contract e_1.

Dominated Path. If $P_1 \subseteq P_2$ for two demand paths, then delete P_2.

The correctness and the polynomial runtimes of the rules are shown in Chapter 8. We summarize these findings in Lemma 9.1.

Lemma 9.1. *The above reduction rules are correct and they can be executed in $O(h \cdot |V|^3 + h^3 \cdot |V|)$ worst-case time such that finally no more rules are applicable.*

The following lemma shows that the "iff"-relation as required for a data reduction to base subsets in Section 9.3 holds.

Lemma 9.2. *There is a demand path in T iff there is a length-two or length-one demand path in the reduced tree T' after exhaustive application of the three data reduction rules.*

Proof. The claim follows directly from Lemmas 8.3 and 8.4. □

Note that, in order to destroy a length-one demand path, we have to put the only edge of this path into the multicut set M. Therefore, the branching rule for this forbidden subgraph finds a length-one demand path in the reduced tree T' and specifies only one smaller instance: contracting the edge, deleting the vertex pairs from H whose demand paths passing through this edge, and decreasing parameter value by one. For a length-two demand path we have

a branching rule finding a length-two demand path in T' and specifying two smaller instances: one of them contracting the first edge of the length-two path and deleting all vertex pairs from H whose demand paths pass through this edge, the other doing the same with the second edge of this path. In both new instances the value of parameter k is decreased by one. The termination conditions are whether $k \leq 0$ and whether $H = \emptyset$.

With branching vectors (1) and $(1, 1)$, it is easy to compute the size of the search tree which is upper-bounded by 2^k. By combining the runtime in Lemma 9.1 with the search tree size, the runtime of the algorithm amounts to $O(2^k \cdot (h \cdot |V|^3 + h^3 \cdot |V|))$.

Remark: A more simple search tree algorithm for Multicut in Trees with the same upperbound on the search tree size but not based on the forbidden subgraph characterization is discussed in [97].

9.5 Concluding Remarks

Graph modification problems with forbidden subgraph characterizations are a fertile ground for fixed-parameter research, in particular, designing depth-bounded search tree algorithms.

One direction for future research could be to investigate how standard heuristic techniques such as branch-and-bound or A^* from artificial intelligence [164] can be used in the standard depth-bounded search tree technique to obtain further speed-ups in practice.

The automated generation of branching rules has become a hot topic in the field of fixed-parameter algorithm engineering [72, 85, 107, 124, 143]. Herein, the idea is to use the sheer computing power of machines to explore a large number of case distinctions. A successful framework for computer-generated search trees for graph modification problems is provided in [85, 107]. To extend the framework to other graph or non-graph problems and to examine the practical relevance of these search tree algorithms remains an issue of future research.

Chapter 10

Cluster Editing

In this chapter we show that Cluster Editing is solvable in $O(2.27^k + |V|^3)$ worst-case time by a depth-bounded search tree algorithm based on a forbidden subgraph characterization. This gives a simple and efficient exact algorithm for this NP-complete problem in case of small parameter values k (number of deletions and additions). We follow parts of [86].

Recall that the Cluster Editing problem is defined as follows:

> **Input:** An undirected graph $G = (V, E)$ and an integer $k \geq 0$.
> **Task:** Find a set P of at most k two-element subsets of V such that the graph $G' = (V, E')$ with $E' := E \ominus P$ consists of a disjoint union of cliques. (Adding the edges in $P \setminus E$ and deleting the edges in $E \cap P$ results in a disjoint union of cliques.)

A graph that consists of a disjoint union of cliques is called a *cluster graph*. Set P is called *cluster editing set*. Cluster graphs can be characterized by a forbidden induced subgraph, P_3. According to the scheme given in Section 9.3, we first describe a simple algorithm in Section 10.1 solving Cluster Editing in $O(3^k \cdot |V|^3)$ time. A more refined search tree strategy with improved search tree size 2.27^k is given in Section 10.2. The technically most intricate part is to prove that we need only a branching into two subcases for some special subgraphs (Lemma 10.2). Based on an automated generation of branching rules Gramm et al. [85, 107] improved the search tree size to 1.92^k.

10.1 Basic Branching Strategy

First, we formulate the forbidden subgraph characterization for cluster graphs in the following easy to prove lemma.

Lemma 10.1. *A graph $G = (V, E)$ consists of disjoint cliques iff there are no three vertices $u, v, w \in V$ which induce a P_3 in G.*

We call three vertices which induce a P_3 in G a *conflict triple*.

In other words, if G does not consist of disjoint cliques, then we can find a conflict triple between which we either have to insert or to delete an edge in order to destroy the forbidden induced P_3. In the following, we describe the straightforward branching rule according to P_3.

Let (G, k) denote the input instance with at most k edge modifications allowed. The branching rule first determines whether there exists a conflict triple in G and, if existing, finds one. Let u, v, and w denote the three vertices of the conflict triple and $\{u, v\}$ and $\{u, w\}$ be the two edges of the P_3 induced by the conflict triple. The branching rule specifies three smaller new instances as follows:

(B1) $G_1 = (V, E_1)$ with $E_1 := E \setminus \{u, v\}$ and $k_1 := k - 1$.

(B2) $G_2 = (V, E_2)$ with $E_2 := E \setminus \{u, w\}$ and $k_2 := k - 1$.

(B3) $G_3 = (V, E_3)$ with $E_3 := E \cup \{v, w\}$ and $k_3 := k - 1$.

The two termination conditions are:

- $k \leq 0$?
- Is G $\{P_3\}$-free?

The search tree algorithm starts with the input graph G and, if the termination conditions are not satisfied, applies the branching rule specifying three smaller instances shown above. Then, the branching rule is recursively applied to the three smaller instances. The algorithm reports a solution if and only if, at one search tree node, the second termination condition is satisfied but the first is not.

Proposition 10.1. CLUSTER EDITING *can be solved in* $O(3^k \cdot k^2 + |V|^3)$ *time.*

Proof. The branching rule and the resulting search tree algorithm suggested above is obviously correct. Concerning the runtime, we observe the following. We can first apply the kernelization process shown in Chapter 7 which can be done in $O(|V|^3)$ time (Theorem 7.1). After that, we employ the search tree with size clearly bounded by $O(3^k)$. Hence, it remains to justify the factor k^2 which stands for the computational overhead related to every search tree node. First, note that in a further preprocessing step, we can once set up a linked list of all conflict triples. This is clearly covered by the $O(|V|^3)$ term. Secondly, within every search tree node (except for the root) we deleted or added one edge and, thus, we have to update the conflict list accordingly. Due to Theorem 7.1, we only have $O(k^2)$ graph vertices now. Moreover, if the addition or deletion of an edge $\{u, v\}$ creates new conflict triples, then each of these new conflict triples has to contain at least one of the vertices u and v. Thus, at most $O(k^2)$ new conflict triples can be created by the addition or deletion of edge $\{u, v\}$. With the same argument, the addition or deletion of an edge $\{u, v\}$ can make at most $O(k^2)$ conflict triples disappear. Using a doubly-linked list of all conflict triples, one can update the list, after adding or deleting an edge of the graph,

in $O(k^2)$ time: after adding or deleting edge $\{v_i, v_j\}$, $v_i, v_j \in V$, we iterate over all $O(k^2)$ many vertices $v_l \in V$, $v_l \neq v_i$ and $v_l \neq v_j$. Only the status of the vertex triples $\{v_i, v_j, v_l\}$ can be changed by this modification, either by causing a new conflict (then, the triple has to be added to the conflict list) or by being a conflict solved by the modification (then, the triple has to be deleted from the conflict list). This update for one vertex triple can be done in constant time, by employing a hash table or by using a size-$|V|^3$ array to store, for every vertex triple, pointers to possible entries in the conflict list. Summarizing, the conflict list can be updated in $O(|V|) = O(k^2)$ time. □

The interleaving technique introduced by Niedermeier and Rossmanith [142] can be applied to improve Proposition 10.1:

Corollary 10.1. Cluster Editing *can be solved in* $O(3^k + |V|^3)$ *time.*

Proof. Niedermeier and Rossmanith [142] show that, in case of a polynomial size problem kernel, by doing the "kernelization" repeatedly during the course of the search tree algorithm whenever possible, the polynomial factor in parameter k can be replaced by a constant factor. □

10.2 Refined Branching Strategy

We improve the runtime of the search tree algorithm from Section 10.1 by making a case distinction with three cases and giving for each case a branching rule. Each of the new branching rules achieves a branching number smaller than three. Hence the search tree size is decreased.

Given a graph $G = (V, E)$, we start with distinguishing three main cases that may apply when considering the conflict triple u, v, and w with edges $\{u, v\}$ and $\{u, w\}$.

(C1) Vertices v and w do not share a common neighbor, i.e. $\nexists x \in V, x \neq u$: $\{v, x\} \in E$ and $\{w, x\} \in E$.

(C2) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \in E$.

(C3) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \notin E$.

Regarding case (C1), we show in the following lemma that, here, a branching into two cases (B1) and (B2) as described in Section 10.1 suffices.

Lemma 10.2. *Given a graph $G = (V, E)$, a nonnegative integer k, and a conflict triple $u, v, w \in V$, if v and w do not share a common neighbor besides u, then branching subcase (B3) cannot lead to a better solution than both subcases (B1) and (B2), and can therefore be omitted.*

Proof. We prove this claim by a counting argument. Consider a cluster graph G' which is obtained by applying a cluster editing set to G. Herein, we did add

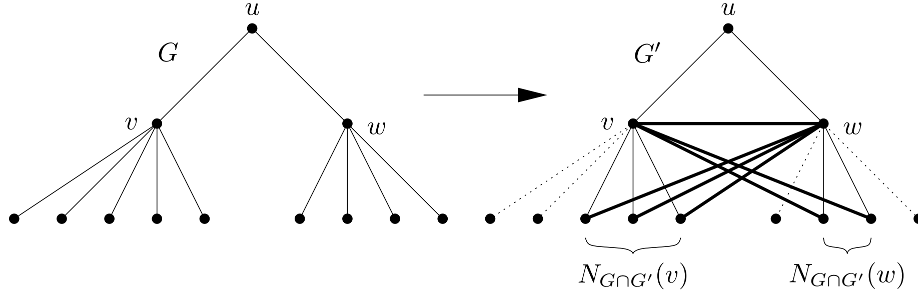

Figure 10.1: In case (C1), adding edge $\{v, w\}$ does not need to be considered. Here, G is the given graph and G' is a cluster graph resulting by applying a cluster editing set that contains insertion of edge $\{v, w\}$ to G. The dashed lines denote the edges being deleted to transform G into G', and the bold lines denote the edges being added. Observe that the drawing only shows that parts of the graphs (in particular, edges) which are relevant for our argumentation.

$\{v, w\}$ (see Figure 10.1). We use $N_{G\cap G'}(v)$ to denote the set of vertices which are neighbors of v in G and in G'. Without loss of generality, assume that

$$|N_{G\cap G'}(w)| \overset{(*)}{\leq} |N_{G\cap G'}(v)|.$$

We then construct a new graph G'' from G' by deleting all edges incident to w. Deleting all edges incident to w results in a new clique containing only vertex w. Together with the fact that G' is a cluster graph, we can infer that G'' is also a cluster graph. We compare the size of the cluster editing set P'' transforming G into G'' to the size of the cluster editing set P' transforming G into G':

- P' inserts $\{v, w\}$ while P'' does not;
- P'' deletes $\{u, w\}$ while P' does not;
- P' adds $|N_{G\cap G'}(v)|$ edges $\{w, x\}$, $x \in N_{G\cap G'}(v)$, while P'' does not;
- P'' deletes $|N_{G\cap G'}(w)|$ edges $\{w, x\}$, $x \in N_{G\cap G'}(w)$, while P' does not;

Herein, we omitted possible vertices which are neighbors of w in G' but not neighbors of w in G because they would only increase the size of P'.

Since P' and P'' make the same modifications with respect to the other parts of G, we obtain

$$|P'| - |P''| \geq |N_{G\cap G'}(v)| + 1 - |N_{G\cap G'}(w)| - 1 \overset{(*)}{\geq} 0.$$

Therefore, the size of P'' is not greater than the size of P', i.e., we do not need more edge additions and deletions to obtain G'' from G than to obtain G' from G. Note that P'' does not contain the insertion of the edge $\{v, w\}$. This

means that there always exists an optimal cluster editing set which, for a conflict triple u, v, w where v and w share no common neighbor besides u, does not insert edge $\{v, w\}$. This is why we can omit (B3) for such a conflict triple. □

Lemma 10.2 shows that in case (C1) a branching into two subcases is sufficient, namely to recursively consider graphs $G_1 = (V, E \setminus \{u, v\})$ and $G_2 = (V, E \setminus \{u, w\})$, each time decreasing the parameter value by one.

In order to achieve branching rules for cases (C2) and (C3), we introduce an annotation Ψ mapping any vertex pair to a three-element set {"permanent", "forbidden", "none"}. The semantics of the mapping is as follows:

$\Psi(\{u, v\})$ = "permanent" means that $\{u, v\} \in E$ and the edge $\{u, v\}$ cannot be deleted.

$\Psi(\{u, v\})$ = "forbidden" means that $\{u, v\} \notin E$ and the edge $\{u, v\}$ cannot be inserted.

$\Psi(\{u, v\})$ = "none" means that there is no information available about the edge $\{u, v\}$ and it can be edited.

We apply the following data reduction rule to an annotated graph G:

Annotation Rule For every three vertices $u, v, w \in V$:

1. If $\Psi(\{u, v\}) = \Psi(\{u, w\})$ = permanent, then the edge $\{v, w\}$, if it is not in E, has to be added to E and $\Psi(\{v, w\}) :=$ permanent.
2. If $\Psi(\{u, v\})$ = permanent and $\Psi(\{u, w\})$ = forbidden, then the edge $\{v, w\}$, if it is in E, has to be deleted from E and $\Psi(\{v, w\}) :=$ forbidden.

The correctness and the runtime of Annotation Rule follows directly from its description.

Lemma 10.1. *Annotation Rule is correct and can be done in $O(|V|^3)$ time.*

With the annotation mapping Ψ and Annotation Rule, we are ready to describe the branching rules for cases (C2) and (C3).

For case (C2) where the vertices v and w of a conflict triple u, v, w have a common neighbor $x \neq u$ with $\{u, x\} \in E$, we make the following branching: In the first branch, we add edge $\{v, w\}$. In the second and third branches, we delete edges $\{u, v\}$ and $\{u, w\}$, respectively, as illustrated by Figure 10.2:

- Add $\{v, w\}$ as labeled by ② in Figure 10.2. We have only one new smaller instance with a parameter value decreased by one.
- Set $\Psi(\{v, w\})$ = forbidden and delete $\{u, v\}$, as labeled by ③. This creates the new conflict triple u, v, x. Since adding $\{u, v\}$ is forbidden, in order to destroy the P_3 induced by u, v, x, there are only two possibilities to consider:
  - Delete $\{v, x\}$, as labeled by ⑤.

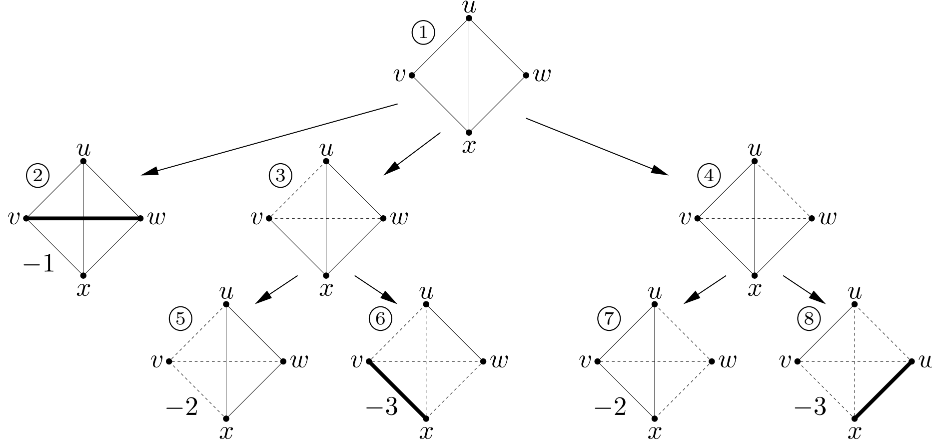

Figure 10.2: The branching rule for case (C2). Bold lines denote edges annotated as permanent, dashed lines edges annotated as forbidden.

  - Set $\Psi(\{v,x\}) =$ permanent and delete $\{u,x\}$. With Annotation Rule, we then delete $\{w,x\}$, too, as labeled by ⑥.

  By combining these two possibilities into the branching subcase where we delete the edge $\{u,v\}$, we obtain two new smaller instances, one with the deletion of edges $\{u,v\}$ and $\{v,x\}$ and a parameter value decreased by two; the other with the deletion of edges $\{u,v\}$, $\{u,x\}$, and $\{w,x\}$ and a parameter value decreased by three.

- Set $\Psi(\{v,w\}) =$ forbidden and delete $\{u,w\}$ (④). Because this case is symmetric to the previous one, we have also two new smaller instances with a parameter value decreased by two or three, respectively.

In summary, the branching rule for case (C2) specifies five new smaller instances and its branching vector is $(1,2,3,2,3)$.

For case (C3), we derive a branching rule as illustrated by Figure 10.3:

- Delete $\{u,v\}$, as labeled by ②. The obtained smaller instance has a parameter value decreased by one.
- Set $\Psi(\{u,v\}) =$ permanent and delete $\{u,w\}$, as labeled by ③. With Annotation Rule, we can additionally set $\Psi(\{v,w\})$ as forbidden. We then identify a new conflict triple u,v,x. Not being allowed to delete $\{u,v\}$, we have only two possibilities to destroy the P_3 induced by u,v,x:
  - Delete $\{v,x\}$, as labeled by ⑤.
  - Set $\Psi(\{v,x\}) =$ permanent. This implies that $\{u,x\}$ needs to be added and $\{w,x\}$ needs to be deleted due to Annotation Rule, as labeled by ⑥.

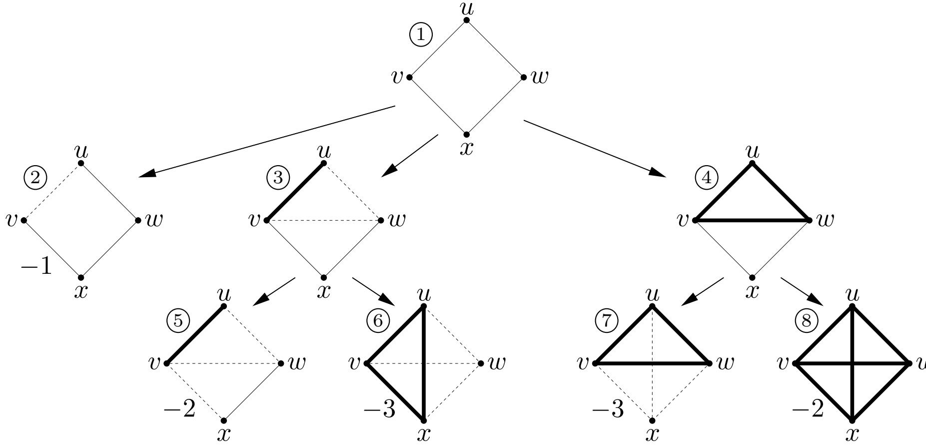

Figure 10.3: The branching rule for case (C3).

By combining these two possibilities into the branching subcase where we delete the edge $\{u, w\}$, we have two new smaller instances, one with the deletion of edges $\{u, w\}$ and $\{v, x\}$ and a parameter value decreased by two; the other with the deletion of edges $\{u, w\}$ and $\{w, x\}$ and the insertion of edge $\{u, x\}$ and a parameter value decreased by three.

- Set $\Psi(\{u, v\}) = \Psi(\{u, w\}) =$ permanent and add $\{v, w\}$, as labeled by ④. The vertices u, w, and x induce a P_3. To destroy this P_3 without deleting $\{u, w\}$, we have only two possibilities:

  – Delete $\{w, x\}$ as labeled by ⑦. We then also need to delete $\{v, x\}$.

  – Add $\{u, x\}$, as labeled by ⑧.

  By combining these two possibilities into the branching subcase that adds the edge $\{v, w\}$, we obtain two new smaller instances, one with the deletion of edges $\{v, x\}$ and $\{w, x\}$ and the insertion of edge $\{v, w\}$ and a parameter value decreased by three; the other with the insertion of edges $\{v, w\}$ and $\{u, x\}$ and a parameter value decreased by two.

It follows that the branching rule for case (C3) specifies five new smaller instances and its branching vector is $(1, 2, 3, 3, 2)$.

In summary, this leads to a refinement of the branching with the new worst-case branching vector $(1, 2, 2, 3, 3)$, yielding branching number 2.27. The algorithm stops whenever the parameter value has reached 0 or below or when G is P_3-free. We obtain a search tree size of $O(2.27^k)$. This results in the following theorem.

Theorem 10.1. Cluster Editing *can be solved in* $O(2.27^k + |V|^3)$ *time.*

10.3 Concluding Remarks

In [85, 107], by using an automated generation of more intricate branching rules based on more complicated base distinctions, an at least theoretical improvement over the search tree size given here has been achieved. More precisely, the improved upper bound on the search tree size is $O(1.92^k)$. To what extent this smaller worst-case bound also has practical significance remains an issue of future research. Note that the computer-generated search tree has a significantly increased number of branching rules which causes increased overhead in the implementation etc.

We feel that the whole field of data clustering problems might benefit from more studies on the parameterized complexity of the many problems related to this field. One possible subject of future research could be the so-called p-CLUSTER EDITING problem introduced by Shamir et al. [163]. The parameterized complexity of this problem is still open when we consider the combined parameter consisting of the number of allowed edge modifications and the number p of clusters. Note that, parameterized only by the number p of clusters, there is no hope for fixed-parameter tractability because Shamir et al. [163] showed that p-CLUSTER EDITING is NP-complete for $p \geq 2$.

Chapter 11

Closest 3-Leaf Power

Most depth-bounded search tree algorithms solving graph modification problems work directly on the input graph. Branching rules specify new instances consisting of subgraphs of the input graph. In this chapter we present a fixed-parameter algorithm solving the Closest 3-leaf power problem. This search tree algorithm does not directly work on the input graph but on a simplified special graph. The technically most difficult part of this chapter is the proof of Lemma 11.3 which justifies the strategy of working on the simplified graph. In addition, we also give a further example for a forbidden subgraph characterization. We follow partly [59].

11.1 Problem Definition and Previous Results

Problem Definition. In order to give a formal definition of the problem studied here, we need the following notion which was introduced by Nishimura et al. [144]. Herein, we use $d_G(u, v)$ to denote the length of a shortest path between vertices u and v in graph G.

Definition 11.1. *For an unrooted tree T with leaves one-to-one labeled by the elements of a set V, the* ℓ-leaf power *of T is a graph, denoted T^ℓ, with $T^\ell := (V, E)$, where*

$$E := \{\{u, v\} \mid u, v \in V \text{ and } d_T(u, v) \leq \ell\}.$$

The tree T is called an ℓ-leaf root *of T^ℓ.*

See Figure 11.1 for two examples of leaf powers. The recognition problem for ℓ-tree powers is defined as follows:

ℓ-Leaf Power Recognition (LPℓ)

Input: A graph G.

Question: Is there a tree T such that $T^\ell = G$?

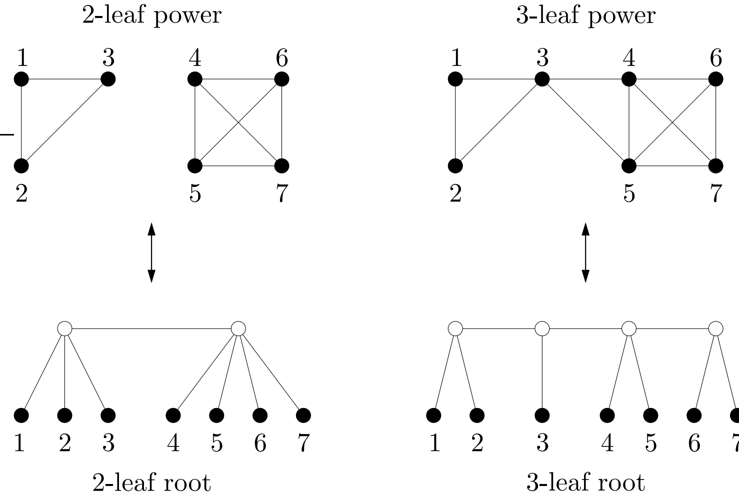

Figure 11.1: Leaf powers and leaf roots. The leaves of a leaf root stand in one-to-one correspondence to the vertices of its leaf power.

The main problem of this chapter is an "approximate" version of the recognition problem:

CLOSEST ℓ-LEAF POWER (CLPℓ)

Input: A graph $G = (V, E)$ and a nonnegative integer k.

Task: Find a set X of at most k vertex pairs, i.e., $X \subseteq V \times V$ and $|X| \leq k$, such that the graph $G' = (V, E')$ with $E' := E \ominus X$ is an ℓ-leaf power. (Adding the edges in $X \setminus E$ and deleting the edges in $E \cap X$ result in an ℓ-leaf power.)

More precisely, this problem is denoted as CLPℓ EDGE EDITING. In the literature there exist two variations of this problem:

- CLPℓ EDGE INSERTION: Only inserting edges into G is allowed to obtain T^ℓ (that is, $E(T^\ell) \supseteq E(G)$);
- CLPℓ EDGE DELETION: Only deleting edges from G is allowed to obtain T^ℓ (that is, $E(T^\ell) \subseteq E(G)$).

In addition, we examine the problem that considers deleting vertices instead of edges.

- CLPℓ VERTEX DELETION: Find a set of at most k vertices whose removal transforms G into an ℓ-leaf power.

We remark that CLP2 is equivalent to CLUSTER EDITING. In other words, each cluster graph is a 2-leaf power: We construct for each clique in a cluster graph a star with the leaves labeled by the vertices of the clique and then insert

edges between the center vertices of the stars such that the center vertices induce a path. The resulting tree is clearly a 2-leaf root of the cluster graph. Conversely, every two adjacent vertices of a 2-leaf power G one-to-one correspond to two leaves having the same parent in the corresponding 2-leaf root T. Thus, all leaves in T having the same parent correspond to vertices of a clique in the leaf power G. From the definition of 2-leaf powers, two leaves having distinct parents correspond to two non-adjacent vertices in G. Furthermore, because each leaf has only one parent, each vertex in G can be in only one clique. Then, G is a cluster graph.

Motivation. Motivated by applications in computational biology, Nishimura et al. [144] introduced the notion of *leaf powers*: A fundamental problem in computational biology is to reconstruct the evolutionary history of a set of species from biological data. This evolutionary history is typically modelled by a *phylogenetic tree*. Each leaf of a phylogenetic tree represents a distinct species and each internal vertex represents a speciation event. Modelling the interspecies similarity by a graph where the vertices are the species and the edges represent the evolutionary similarity between the vertices, the problem of forming a phylogenetic tree can be framed as the problem of constructing a leaf root from a given graph.

Note that the input graphs are derived from some evolutionary similarity data which is usually inexact in practice and may have erroneous edges. Such errors may result in graphs which have no leaf root. Hence, it is natural to consider the "error correction setting", namely Closest ℓ-Leaf Power.

Previous Results. Graph powers are a classical concept in graph theory [30, Section 10.6] with recently increased interest from an algorithmic point of view. The *ℓ-power* of a graph $G = (V, E)$ is the graph $G^\ell = (V, E')$ with $\{u, v\} \in E'$ iff there is a path of length at most ℓ between u and v in G. We say G is the *ℓ-root* of G^ℓ and G^ℓ is the *ℓ-power* of G. It is NP-complete to decide whether a given graph is an ℓ-power [135]. By way of contrast, one can decide in $O(|V|^3)$ time whether a graph is an ℓ-power of a tree for any fixed ℓ [116]. In particular, it can be decided in linear time whether a graph is a square of a tree [130, 127]. Kearney and Corneil [116] introduced the Closest ℓ-Tree Power problem determining whether a given graph can be modified by adding or deleting at most k edges such that the resulting graph has an ℓ-tree root. Unfortunately, this problem turns out to be NP-complete for $\ell \geq 2$ [116, 110], even for the special case that the vertex degree of the ℓ-tree root is upper-bounded by four [61].

In addition, Lin et al. [129], Chen et al. [42], and Chen and Tsukiji [43] examined the variant of leaf powers where all inner vertices of the root tree have degree at least three. The corresponding algorithmic problems to decide whether a graph has such an ℓ-root are called ℓ-Phylogenetic Root (PRℓ) [129]. PRℓ is solvable in polynomial time for $\ell \leq 4$ [129]. For $\ell \geq 5$, its complexity is open. Moreover, Chen et al. [42] and Chen and Tsukiji [43] showed that, under the assumption that the maximum vertex degree of the phylogenetic root

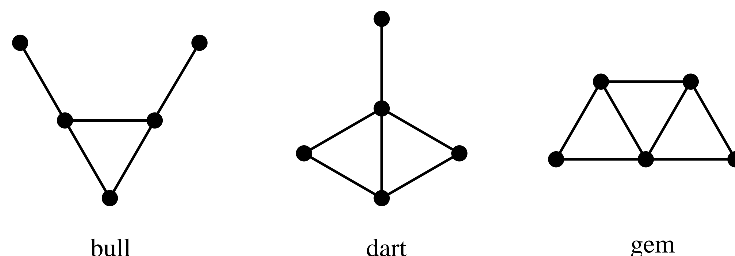

Figure 11.2: 5-vertex graphs that occur as forbidden induced subgraphs for 3-leaf powers.

is bounded from above by a constant, there is a linear-time algorithm that determines whether a graph has an ℓ-phylogenetic root (that is, an ℓ-leaf root with minimum vertex degree three) for arbitrary ℓ.

Concerning the recognition problem of leaf powers, Nishimura et al. [144] showed that, for $\ell \leq 4$, LPℓ is solvable in $O(|V|^3)$ time. The complexity of this problem for $\ell \geq 5$ is still open. In contrast, CLOSEST ℓ-LEAF POWER for $\ell \geq 2$ is NP-complete [126, 59]. The NP-completeness of CLPℓ VERTEX DELETION for $\ell \geq 2$ follows directly from a result by Lewis and Yannakakis [128], who showed that the vertex deletion problem is NP-complete for any nontrivial hereditary graph property. To the best of our knowledge, except for the case $\ell = 2$ (for more results of CLP2 which is equivalent to CLUSTER EDITING we refer to Chapters 7 and 10) these graph modification problems so far have only led to complexity hardness results. We are not aware of any results concerning polynomial-time approximation or nontrivial exact algorithms.

In Section 11.3, we show the fixed-parameter tractability with respect to the number k of edge modifications for CLOSEST 3-LEAF POWER by giving a search tree algorithm. To this end, we develop a novel forbidden subgraph characterization of 3-leaf powers in the next section. We mention in passing that the fixed-parameter tractability of CLOSEST 4-LEAF POWER with the number of edge modifications as parameter is shown in a similar way, that is, also using a search tree algorithm based on a forbidden subgraph characterization [57, 58].

11.2 Forbidden Subgraph Characterization for 3-Leaf Powers

In this section we derive an *induced* forbidden subgraph characterization of 3-leaf powers: They are chordal graphs that contain no induced bull, dart, or gem (see Figure 11.2). Note that chordal graphs forbid all induced chordless cycles with length four or higher. In other words, this characterization of 3-leaf powers admits an infinite set of forbidden subgraphs.

As we will see, 3-leaf powers are closely connected to the concept of *critical cliques* and *critical clique graphs*, which were introduced by Lin et al. [129]. For

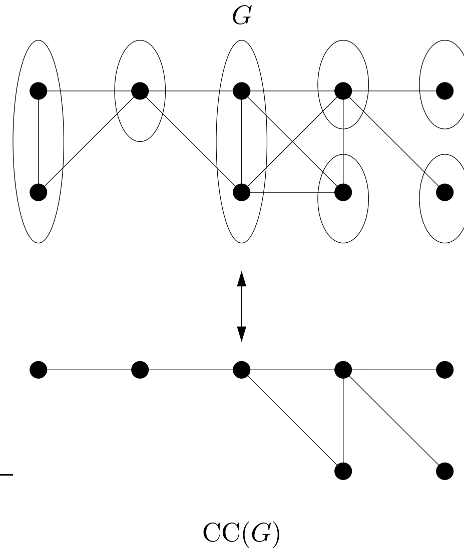

Figure 11.3: A graph G and its critical clique graph $\mathrm{CC}(G)$. *Ovals* denote the critical cliques of G.

easier distinction from the elements of G, we use the term *nodes* for vertices in the critical clique graph.

Definition 11.2. *A* critical clique *of a graph G is a clique K where the vertices of K all have the same set of neighbors in $G \setminus K$, and K is maximal under this property.*

Definition 11.3. *Given a graph $G = (V, E)$. Let C be the collection of its critical cliques. Then the* critical clique graph $\mathrm{CC}(G)$ *is a graph (C, E_C) with*

$$\{K_i, K_j\} \in E_C \iff \forall u \in K_i, v \in K_j : \{u, v\} \in E.$$

That is, the critical clique graph has the critical cliques as nodes, and two nodes are connected iff the corresponding critical cliques together form a larger clique.

An example of a graph and its critical clique graph is shown in Figure 11.3. Due to Definition 11.3, every vertex of G belongs to exactly one critical clique. The critical clique graph of G can be constructed in $O(|V| \cdot |E|)$ time: As long as there are two adjacent vertices u and v which have the same closed neighborhood, i.e., $N[u] = N[v]$, we "merge" them into a new vertex, that is, we delete u and v and create a new vertex with the neighborhood equal to $N(u)$. After iterating through all pairs of adjacent vertices, the resulting graph is then the critical clique graph.

The following connection to 3-leaf powers can be shown:

Lemma 11.1. *If a graph G is a 3-leaf power, then every clique in G contains vertices of at most two critical cliques.*

Proof. If two vertices of G are adjacent to the same inner vertex in a 3-leaf root T of G, then they have identical neighborhoods in G, since their distances to other leaves in T are identical. Therefore, vertices from different critical cliques cannot be adjacent to the same inner vertex of T. For the purpose of contradiction, we assume that there is a clique K in G that contains vertices of at least three critical cliques. Then the vertices of K must be adjacent to at least three different inner vertices of T. Two of three inner vertices in a tree have distance at least 2, which already yields a distance of 4 between their leaf children. This implies that their leaf children cannot be in a clique, contradicting the assumption that their leaf children are part of clique K. □

The following lemma is decisive for the proof of the forbidden subgraph characterization and the fixed-parameter algorithms in the next section. Herein, a C_4 is a cycle induced by four vertices.

Lemma 11.2. *For a* $\{C_4\}$*-free graph* G*, the following are equivalent:*

(1) $\mathrm{CC}(G)$ *contains a triangle.*

(2) G *contains a bull, dart, or gem (see Figure 11.2) as an induced subgraph.*

Proof. **(1) ⇒ (2):** Consider three nodes in $\mathrm{CC}(G)$ that form a triangle and take one vertex of each of the corresponding critical cliques in G. These three vertices are pairwise connected by an edge and must have pairwise distinct neighborhoods. We make a case distinction based on whether there exists a non-common neighbor that is connected to exactly one of the three vertices or not. In each of these cases, we can get a bull, dart, or gem in G. For details on the case distinction refer to [59].

(2) ⇒ (1): Assume that G contains a forbidden subgraph. Let u, v, w be the vertices of a triangle in the forbidden subgraph (in the case of the gem, the triangle which contains both degree-3 vertices). Then u, v, w form a clique. Let x and y be the remaining two vertices in the subgraph. Since each of u, v, w is adjacent to a different combination of x and y, they belong to three different critical cliques. By Definition 11.3 the triangle formed by vertices u, v, w exists in $\mathrm{CC}(G)$.

□

Utilizing Lemmas 11.1 and 11.2, we obtain a forbidden subgraph characterization of 3-leaf powers.

Theorem 11.1. *For a graph* G*, the following are equivalent:*

(1) G *is a 3-leaf power.*

(2) G *is chordal and contains no bull, dart, or gem as an induced subgraph.*

(3) $\mathrm{CC}(G)$ *is a forest.*

Proof. **(1) ⇒ (2):** If G is a leaf power, then G must be chordal [129]. Moreover, it is easy to observe that each of bull, dart, and gem contains a triangle whose three vertices have distinct neighborhoods and have to belong to three distinct critical cliques. Thus, by Lemma 11.1, if G is a 3-leaf power, then G contains none of bull, dart, and gem as an induced subgraph.

(2) ⇒ (3): If G is chordal, then so is $\mathrm{CC}(G)$, since if $\mathrm{CC}(G)$ contained a hole, we could also find a hole in G by taking one arbitrary vertex from each critical clique on the cycle. By Lemma 11.2, $\mathrm{CC}(G)$ contains no triangle. A triangle-free, chordal graph is clearly acyclic. Thus, it follows that $\mathrm{CC}(G)$ is a forest.

(3) ⇒ (1): We prove this by constructing a 3-leaf root of G from the given critical clique graph $\mathrm{CC}(G)$ of G: First, we consider every connected component of the forest $\mathrm{CC}(G)$ individually. For a connected component which is a tree, we construct a new tree by attaching to each node of the tree one new leaf node for each graph vertex belonging to the corresponding critical clique. Then we join the trees newly constructed for the connected components of $\mathrm{CC}(G)$ together by creating a new node and connecting this node to an arbitrary inner node of each of these trees. We show in the following that the resulting tree T is a 3-leaf root of G.

On the one hand, consider two adjacent vertices u and v with $u \neq v$ of G. They have to be in the same critical clique or in two adjacent critical cliques. The former case implies that the leaf nodes corresponding to u and v have the same parent node in T and, thus, the distance in T between them is two. In the latter case, the leaf nodes corresponding to u and v are attached to two adjacent nodes in T. Therefore, the distance in T between these two leaf nodes is three.

On the other hand, consider two leaf nodes which have a distance of two or three in T. They have the same parent node or their parent nodes are adjacent. This implies that the two vertices in G corresponding to these two leaf nodes have to belong to the same critical clique or two adjacent critical cliques and they, thus, are adjacent in G. This shows that T is a 3-leaf root of G.

□

11.3 Algorithms

First, we show fixed-parameter tractability results with respect to the number of editing operations k for CLP3 Edge Insertion, CLP3 Edge Deletion, and CLP3 Edge Editing (Section 11.3.1). Then, we reuse the ideas derived for the edge versions to solve CLP3 Vertex Deletion (Section 11.3.2).

11.3.1 Edge Modifications (Overview)

Since the characterization of 3-leaf powers from Theorem 11.1 contains infinitely many forbidden subgraphs, including all induced chordless cycles of length at least four, we cannot directly derive a search tree algorithm from this characterization. It seems also hard to give a data reduction to a base subset with respect to the infinite set of forbidden subgraphs. Lemma 11.2 provides the main idea behind the algorithms given in the following: Compared with a $\{C_4$, bull, dart, gem$\}$-free graph G, its corresponding critical clique graph $\mathrm{CC}(G)$ is much more simple, that is, $\mathrm{CC}(G)$ is triangle-free. Editing a triangle-free graph into a forest is a much easier task than editing an arbitrary graph into a 3-leaf power. Following this idea, our algorithms consist of two steps. The first step is working on the input graph G and it gets rid of all induced C_4s, bulls, darts, and gems in G by edge deletions and insertions. In order to destroy these four forbidden subgraphs, we can clearly apply Cai's result [32] with respect to designing depth-bounded search tree algorithms based on a finite set of forbidden subgraphs, see Section 9.3. The size of the search tree is upperbounded by N^k where N denotes the maximum size of the four forbidden subgraphs. The runtime for finding one of the four subgraphs is clearly $O(|V|^5)$ since they contain at most five vertices. Thus, the overall runtime of the first step is $O(N^k \cdot |V|^5)$. The second step works then on the critical clique graphs of the resulting graphs output by the first step. Depending on whether only edge deletions, only edge insertions, or both of them are allowed, this step applies different methods to transform the triangle-free critical clique graphs into forests.

The following central lemma of this section guarantees the correctness of working on the critical clique graphs in the second step of the algorithms.

Lemma 11.3. *There is always an optimal solution for* CLP3 EDGE EDITING, CLP3 EDGE DELETION, *or* CLP3 EDGE INSERTION *that is represented by edge editing operations on* $\mathrm{CC}(G)$. *That is, one can find an optimal solution that does not delete any edges within a critical clique; furthermore, in this optimal solution, between two critical cliques either all or no edges are inserted or deleted.*

Proof. (idea) We consider here only CLP3 EDGE EDITING, but the same argumentation holds for CLP3 EDGE DELETION and CLP3 EDGE INSERTION. Observe that deleting an edge within a critical clique or inserting or deleting some but not all edges between two critical cliques results in splitting one critical clique in G into at least two critical cliques in the resulting graph, that is, the vertices of a critical clique of G belong to different critical cliques of the resulting graph. Thus, if we can construct an optimal solution for CLP3 EDGE EDITING on G which splits no critical clique of G, then we have shown the lemma. Let F_{opt} be an arbitrary optimal solution for CLP3 EDGE EDITING on $G = (V, E)$, and $G_{\mathrm{opt}} := (V, E \ominus F_{\mathrm{opt}})$. If F_{opt} splits no critical clique of G, then we are done; otherwise, for a critical clique K of G, there are at least two critical cliques K_1 and K_2 in G_{opt} which both have common vertices with K. We claim that there

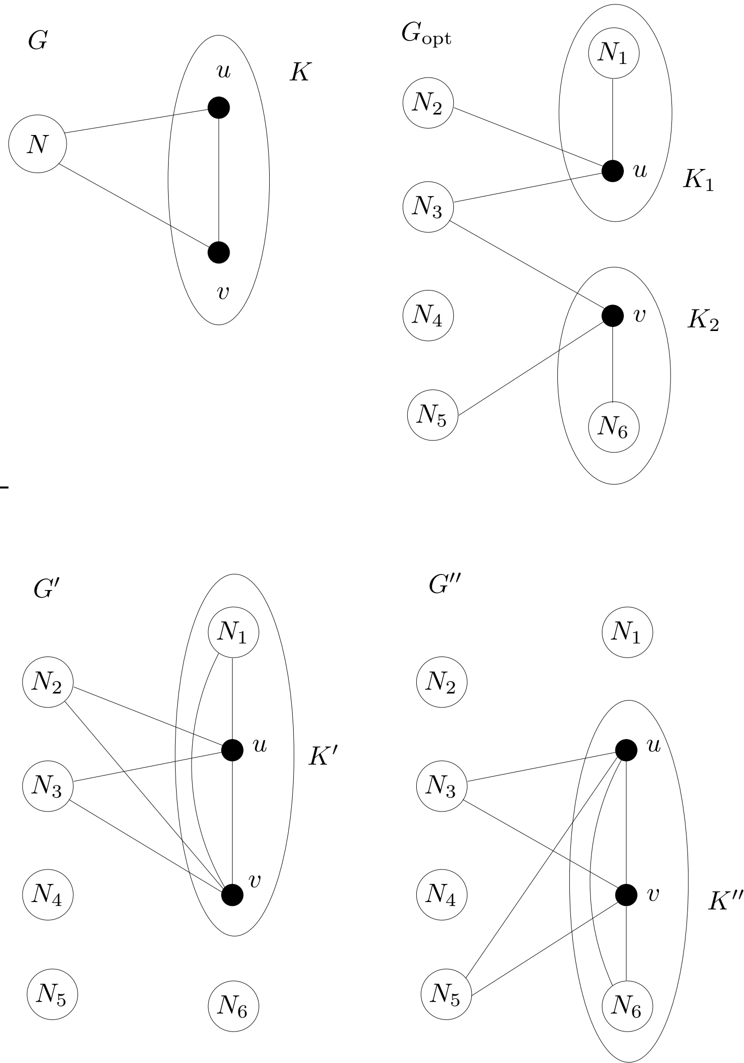

Figure 11.4: Illustration of the proof of Lemma 11.3. *Circles* denote vertex subsets and *ovals* denote critical cliques. Lines denote the edge $\{u, v\}$ and edges between u, v and the vertex subsets. Only the edges having u or v as one endpoint are displayed. Set N contains all neighbors of u and v in G. The vertices in N_1 (or N_6) are contained in the critical clique K_1 (or K_2) in G_{opt}. Set N_2 (or N_5) contains the vertices only adjacent to u (or v) in G_{opt}. Set N_3 contains the common neighbors of u and v in G_{opt} and $N_4 := N \setminus (N_1 \cup N_2 \cup N_3 \cup N_5 \cup N_6)$. Obviously, sets $N_1, \ldots, N_6$ are pairwise disjoint. Graphs G' and G'' are obtained by editing edges incident to u or v in G_{opt}.

exists a solution which does not split K and is at least as good as F_{opt}. In the following we show the claim for the simple case that $K = \{u, v\}$. For $|K| > 2$ the claim can be shown in basically the same way.

Consider a critical clique $K = \{u, v\}$ with N denoting K's neighborhood in G as shown in the top left part in Figure 11.4. The optimal solution F_{opt} splits K into two critical cliques K_1 and K_2. Suppose that $K_1 = \{u\} \cup N_1$ and $K_2 = \{v\} \cup N_6$ as shown in G_{opt} in Figure 11.4. Moreover, suppose that the neighborhood of K_1 contains $N_2 \cup N_3$ and the neighborhood of K_2 contains $N_3 \cup N_5$. We set $N_4 := N \setminus (N_1 \cup N_2 \cup N_3 \cup N_5 \cup N_6)$. Comparing G_{opt} and G we can identify the edge modifications in F_{opt} concerning the vertices u and v:

- Insertion of the edges between vertex u and the vertices in $(N_1 \cup N_2 \cup N_3) \setminus N$ and the edges between vertex v and the vertices in $(N_3 \cup N_5 \cup N_6) \setminus N$;
- Deletion of the edge $\{u, v\}$ and the edges between vertex u and the vertices in $(N_4 \cup N_5 \cup N_6) \cap N$ and the edges between vertex v and the vertices in $(N_1 \cup N_2 \cup N_4) \cap N$.

In order to show the claim, consider the two graphs G' and G'', shown in the lower part of Figure 11.4. These two graphs are obtained by editing edges in G_{opt} which are incident to u or v. Note that in G' and G'' the critical clique K is not split. Obviously, the only difference between the sets of critical cliques of G_{opt} and G' (or G'') is that the critical clique K_2 (or K_1) in G_{opt} has in G' (or G_2) one vertex less while the critical clique K_1 (or K_2) in G_{opt} has in G' (or K_2) one vertex more. Thus, due to the forest structure of $\text{CC}(G_{\text{opt}})$, it is easy to observe that $\text{CC}(G')$ (or $\text{CC}(G'')$) is also a forest and G' (or G'') is a 3-leaf power (Theorem 11.1). Let F' (or F'') denote the set of edge modifications which transform G into G' (or G''). In other words, F' (or F'') is a solution of CLP3 EDGE EDITING on G.

The set F' contains the following edge modifications concerning the vertices u and v:

- Insertion of the edges connecting u or v with the vertices in $(N_1 \cup N_2 \cup N_3) \setminus N$;
- Deletion of the edges between u or v and the vertices in $(N_4 \cup N_5 \cup N_6) \cap N$.

Similarly, F'' contains the following edge modifications concerning the vertices u and v:

- Insertion of the edges connecting u or v with the vertices in $(N_3 \cup N_5 \cup N_6) \setminus N$;
- Deletion of the edges between u or v and the vertices in $(N_1 \cup N_2 \cup N_4) \cap N$.

Since the transformation from G_{opt} to G' and the one from G_{opt} to G'' do not affect the edges that are not incident to u or v, we have $|F'| + |F''| \leq 2 \cdot |F_{\text{opt}}|$. Therefore, at least one of F' and F'' is at least as good as F_{opt}. This completes the proof of the claim for the critical clique K. Repeatedly applying the claim to each critical clique split by F_{opt} we obtain the lemma. For a more complete proof refer to [59]. □

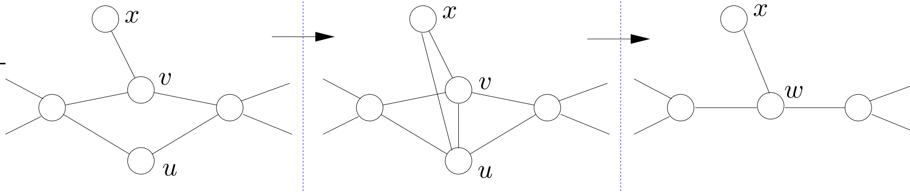

Figure 11.5: An illustration of destroying cycles in CC(G) by merge operations. By inserting edges $\{u, v\}$ and $\{u, x\}$, the two critical cliques corresponding to nodes u and v have the same neighborhood. They have to be merged into one critical clique corresponding to node w. This merge operation destroys the length-four cycle containing nodes u and v.

In the following, we give only a brief description of the second step of the algorithms. More details can be found in [59].

Note that after modifying CC(G), two or more nodes in CC(G) might obtain identical neighborhoods. Since each node in CC(G) has to represent a critical clique in G, a *merge* operation is needed, which replaces these nodes in CC(G) by a new node with the same neighborhood as the original nodes. Therefore, in the following we assume that after each modification operation, we check for every pair of nodes whether a merge operation between them is possible; this can be done in $O(n \cdot m)$ time.

In order to make CC(G) a forest, we have to destroy all cycles in it. A shortest cycle can be found in $O(n \cdot m)$ time [109]. This cycle can be destroyed by either deleting at least one edge of this cycle (CLP3 Edge Deletion), triggering a merge operation for two nodes on this cycle (CLP3 Edge Editing), or both (CLP3). See Figure 11.5 for an illustration of destroying a cycle in CC(G) by a merge operation. Recall that two nodes can be merged iff they are adjacent and they have the same neighborhood. Thus, in order to merge two nodes K_i and K_j, we have to insert an edge between them if they are not already adjacent; furthermore, we need to delete or insert edges such that K_i and K_j have the same neighborhood.

As shown in the proof of Theorem 11.1, if CC(G) has more than one connected component, we can solve the problem for each component independently, and then connect the generated leaf roots by adding a new inner vertex and connecting it to an arbitrary inner vertex of each leaf root. This allows us in the following, without loss of generality, to focus on connected graphs.

Edge Deletion. As stated above, the task is to transform CC(G) into a forest by edge deletions.

Observe that, since CC(G) contains no triangle, a merge operation between two nodes in CC(G) can only be triggered if the two nodes form a connected component. However, for a connected CC(G) with more than two nodes, no optimal solution of CLP3 Edge Deletion can produce a connected component of two nodes; otherwise, at least one edge deletion was not necessary. Thus,

no merge operation is needed when we destroy cycles by deleting edges. The following lemma follows directly from this observation.

Lemma 11.4. *For a triangle-free critical clique graph* CC(G)*, we can find an optimal solution for* CLP3 EDGE DELETION *by finding a maximum weight spanning tree for* CC(G)*, where edges are weighted by the product of the sizes of the critical cliques corresponding to their two endpoints.*

With Lemma 11.4, the second step here can be done in $O(|E| \cdot \log |V|)$ time by applying standard methods finding maximum weight spanning trees [45, 111, 179]. Since the maximum edge number of C_4, bull, dart, and gem is six, the runtime of the first step is $O(6^k \cdot |V|^5)$. Summarizing the two steps, we obtain the following theorem:

Theorem 11.2. CLP3 EDGE DELETION *with k edge deletions allowed is solvable in $O(6^k \cdot |V|^5)$ time.*

Edge Insertion Here, to modify CC(G) to a forest, we again use a depth-bounded search tree based on forbidden subgraphs. Note that to destroy cycles by edge insertions, we have to trigger merges between nodes in CC(G). Since forests have infinitely many forbidden subgraphs, we show that, if CC(G) can be transformed into a forest by at most k edge insertions, then there exists a cycle whose length is bounded from above by a function of k. A search tree algorithm can then be derived based on a branching rule with respect to this cycle. To this end, we prove a connection between the triangulation of a hole and the merge operations that turn a cycle into a tree.

A *triangulation* of a hole $C = (V_C, E_C)$, where V_C denotes the set of the vertices on this cycle and E_C the set of the edges, is a set D of chords placed into C such that there is no hole in $C' = (V_C, E_C \cup D)$. A triangulation D of a graph G is *minimal* if no proper subset of D triangulates G. The following lemma is proven in [59].

Lemma 11.5. *Each set of edges inserted into a cycle C of a critical clique graph to transform C into a tree is a triangulation of C.*

Kaplan et al. [114] showed that a minimal triangulation of an n-vertex-cycle consists of $n-3$ chords. This implies that a graph that can be triangulated by at most k edge insertions cannot have a chordless cycle of length more than $k+3$. With Lemma 11.5, we can easily derive a branching rule with respect to this length-bounded cycle: By trying all possible merge operations for a forbidden cycle with at most $k+3$ vertices, we get the following theorem. For a proof of this theorem refer to [59].

Theorem 11.3. CLP3 EDGE INSERTION *with k edge insertions allowed is solvable in $O((k+3)^k \cdot |V|^5)$ time.*

Edge Editing The algorithm for CLP3 EDGE INSERTION can be easily extend to solve CLP3 EDGE EDITING by additionally taking edge deletions into account. We distinguish two types of cycles: the *short* cycles having length at most $k+3$, and the *long* cycles having length greater than $k+3$.

We can destroy a short cycle in $\mathrm{CC}(G)$ by deleting at least one edge from it, or by merging some critical cliques. This means we have at most $k+3$ possible edge deletions and at most $(k+3)^2$ possible merge operations between a pair of nodes. However, merge operations with both edge deletion and edge insertion are more complicated than merge operations with only edge insertion: For each non-common neighbor of two critical cliques, we can either insert edges or delete edges to transform it into a common neighbor. With at most k edge modifications allowed, a merge operation between two critical cliques is possible only if they have at most k non-common neighbors. Hence we have at most 2^k different ways to merge two critical cliques. Altogether, we now have $(k+3)+(k+3)^2 \cdot 2^k$ branching subcases to transform a short cycle into a tree.

For long cycles, we must only consider edge deletions as shown by the following lemma that is proven in [59].

Lemma 11.6. *1. A long cycle in* $\mathrm{CC}(G)$ *cannot be transformed into a tree solely by edge insertions.*

2. In order to solve CLP3 EDGE EDITING *on a graph with only long cycles, there is no need to insert edges.*

By combining the algorithms for the edge deletion and insertion versions, we obtain the following theorem whose proof can be found in [59].

Theorem 11.4. CLP3 EDGE EDITING *with k edge editing operations allowed is solvable in $O(k^{2k} \cdot |V|^5)$ time.*

11.3.2 Vertex Deletion

We use a similar approach to solve CLP3 VERTEX DELETION as for the edge variants. This first step works on input graph G and gets rid of C_4s, bulls, darts, and gems. The second step transforms then the resulting $\mathrm{CC}(G)$ into forests.

There is a "vertex deletion variant" of Lemma 11.3 which can be proven by using a similar counting argument as in the proof of Lemma 11.3. A detailed proof is given in [59].

Lemma 11.7. *All optimal solutions for* CLP3 VERTEX DELETION *can be represented by node deletions on* $\mathrm{CC}(G)$*. That is, if one vertex in a critical clique is deleted, then all vertices in the critical clique are deleted.*

Observe that the second step, destroying cycles in $\mathrm{CC}(G)$ by deleting vertices, is exactly the FEEDBACK VERTEX SET (FVS) problem (for definition see Section 3.1). However, the nodes in $\mathrm{CC}(G)$ represent critical cliques in G. Then, deleting one node in $\mathrm{CC}(G)$ corresponds to deleting several vertices in G. We have to associate each node in $\mathrm{CC}(G)$ with a weight equal to the size of the

corresponding clique and to solve a weighted version of FVS. Raman et al. [152] gave a fixed-parameter algorithm which can also solve the weighted version of FVS. By applying this algorithm in the second step, we obtain the following result. For the proof of the theorem refer to [59].

Theorem 11.5. CLP3 VERTEX DELETION *with k vertex deletions allowed is solvable in $O((12k \log k)^k \cdot |V|^5)$ time.*

11.4 Concluding Remarks

We list several open questions and future challenges with respect to the CLOSEST 3-LEAF POWER problem.

- It remains open to provide a non-trivial data reduction to a problem kernel for CLOSEST 3-LEAF POWER. For more details on data reduction to problem kernels refer to Chapter 6.
- Also open is the problem to find good polynomial-time approximation algorithms for CLOSEST 3-LEAF POWER.
- One challenge is to investigate whether similar fixed-parameter tractability results can be achieved for the closely related CLOSET ℓ-PHYLOGENETIC ROOT problem studied in [42, 129]. Forbidding degree-2 nodes there in the output trees seems to make things more elusive, though.
- From a more applied point of view, it would be interesting to see how small the combinatorial explosion in the number of modifications can be made for CLP3 and its variants.

  Moreover, the algorithms for the edge editing, edge insertion, and vertex deletion variants require to know the number of modifications k in advance, whereas the algorithm for the edge deletion variant finds an optimal solution even without this knowledge. It would be interesting to see whether we can find fixed-parameter algorithms for the first three variants with this desirable property.

We mention that for CLOSEST 4-LEAF POWER we derived a forbidden subgraph characterization and similar fixed-parameter algorithms [57, 58]. However, the corresponding forbidden subgraph characterization becomes much more complicated and, therefore, causes a much increased algorithmic complexity. As long as it remains open to determine the complexity of ℓ-LEAF POWER RECOGNITION for $\ell > 4$, it seems to make little sense to study the more general CLOSEST ℓ-LEAF POWER for $\ell > 4$.

Part V

Parameterization by Structure

Chapter 12

Basic Concepts and Ideas

Until now we have presented several algorithm design methods for deriving fixed-parameter algorithms where the parameter is always the size of the solution, such as the "edge cut" size for Multicut in Trees, the number of vertex removals for Feedback Vertex Set, and the number of edge modifications for Cluster Editing and Closest 3-Leaf Power. The aim of this chapter is to investigate a different parameterization, the so-called "parameterization by structure." One of the most prominent structural parameters is the notion of treewidth developed by Robertson and Seymour [158]. From an algorithmic viewpoint, parameterizing by structure seems to provide a promising alternative to solve the problems that are hard to approximate and fixed-parameter intractable with respect to solution size. Since a problem carries usually many parameters which represent some structural aspects of the input, there are many meaningful ways to parameterize a problem by structure. In this chapter we will introduce the "distance from triviality" measurement as a prospective way of determining reasonable structural problem parameters and we present two simple examples demonstrating the potential of parameterizing by distance from triviality. The dynamic programming technique, the seemingly most commonly used method for designing fixed-parameter algorithms with respect to structural parameters, will be thoroughly discussed in the next two chapters with two case studies.

12.1 Distance From Triviality

As introduced in Chapter 1, given an instance (x, k) of a parameterized (and, as a rule, NP-hard) problem, parameterized complexity theory offers a two-dimensional framework for studying the computational complexity of the problem: the input size $n := |(x, k)|$ and the parameter value k (usually a non-negative integer). The most important issue herein is whether a problem is fixed-parameter tractable with respect to the chosen parameter k or not. From the viewpoint of complexity theory, the goal is find out whether or not the in-

herent, seemingly unavoidable "combinatorial explosion" of the corresponding problem can be restricted to the considered parameter.

The two-dimensional view on problems by means of parameterization opens new and interesting opportunities for attacking otherwise hard problems: the study of different parameterizations for one and the same problem. Many problems naturally offer a whole selection of parameters and, as a rule, these problems can be parameterized in various reasonable ways according to these parameters. For example, Fellows [74] discusses at least five different parameterizations of the MAX LEAF SPANNING TREE problem.

Roughly speaking, there are at least two fundamentally different ways of parameterization: "parameterizing by solution" and "parameterizing by structure."[1] The former one, which uses the solution size of the corresponding optimization problems as parameter, seems to be the most natural and most common parameterization. We have discussed this kind of parameterization in previous parts for MULTICUT IN TREES (Chapters 8 and 9), FEEDBACK VERTEX SET (Chapter 3), and other graph modification problems. In the following, we discuss the latter one in more detail with particular interest in how to determine a reasonable structural parameter.

We start with an example given by the NP-complete VERTEX COVER problem: Given an undirected graph with n vertices, the goal is to find a vertex subset with minimum cardinality such that each graph edge has at least one of its endpoints in this subset. Parameterized by the solution size k, the currently best fixed-parameter algorithms solve VERTEX COVER in $O(1.2745^k k^+ kn)$ time [36]. Considering special graph classes, we remark that VERTEX COVER is trivially solvable in trees: Root the tree at an arbitrary vertex and, based on the observation that taking a leaf into the cover set is never better than taking its parent, recursively prune the tree in a bottom-up manner. One would like to extend this trivial solution process of VERTEX COVER in trees to graphs which are in structure similar to trees. Now, for example, consider the parameter d defined as the number of edges that have to be deleted from a graph to transform it into a tree. In this sense parameter d measures the "distance" of a graph from a tree and one may ask whether VERTEX COVER is fixed-parameter tractable when parameterized by d. In this simple example the answer is clearly "yes": By applying standard algorithms to compute an arbitrary spanning tree of the given graph, we can find d edges whose removal from the given graph results in a tree.[2] We then enumerate all minimal vertex sets which "cover" these d edges, which can be done in $O(2^d \cdot n)$ time with n denoting the number of graph vertices. For each of the enumerated vertex sets whose number is bounded from above by 2^d, we delete the already covered edges from the input graph. The resulting graph is clearly a forest. Solving VERTEX COVER in linear time on

[1] Note, however, that in the context of approximability up to a factor $(1+\epsilon)$ a parameter k such as $k = 1/\epsilon$ would make sense as well.

[2] The problem seeking for a minimum set of edges whose removal transforms a graph into a tree is also called FEEDBACK EDGE SET, the edge-deletion version of FEEDBACK VERTEX SET, and is equivalent to the problem of computing a spanning tree. It is well-known that a spanning tree can be found in polynomial time [45, 111, 179].

the remaining forest completes the algorithm which runs in $O(2^d \cdot n)$ time.

We extend now the "distance from a tree" concept from the simple example of Vertex Cover to a broader setting, leading to a generic framework of parameterizing hard problems by structure. We call this framework parameterizing by *distance from triviality*. This framework was introduced in [95]. In general, this framework consists of two parts:

1 Identify the trivially (in most cases, polynomial-time) solvable cases for the problem—the triviality—and determine a certain measure that reflects the distance between the trivially solvable and hard problem instances—the structural parameter;

2 Design a fixed-parameter algorithm with the distance measure determined above as structural parameter.

Concerning the first part, one of the deepest examples for the distance from triviality parameterization is clearly the notion of bounded *treewidth* developed by Robertson and Seymour [158]. Without going into details, the basic motivation for considering this concept can be derived from the fact that many NP-hard graph problems (such as Vertex Cover) become easy (linear-time solvable) on trees. Treewidth then measures how close a graph is to the trivial problem instance tree and, if this parameter is small, then many otherwise hard graph problems can be solved efficiently (see [27, 28, 154] for surveys). Trees have treewidth one. In this sense treewidth measures the distance from the triviality "tree" and problems such as Vertex Cover are fixed-parameter tractable with respect to this structural parameter [169]. Comparing the treewidth ω of a given graph with the number d of edge deletions required to transform the graph into a tree, it is easy to see that $\omega \leq d+1$: To transform a graph into a tree (or forest), we need to delete at most d vertices from the graph. Adding all these d vertices to each bag of the tree decomposition of the resulting tree (or forest). We have then a tree decomposition of the input graph with width at most $d+1$. Thus, the treewidth ω is upperbounded by $d+1$. For the definitions of tree decomposition and treewidth, we refer to [27, 28, 119, 154].

We now turn our attention to the second part of the framework and will thoroughly discuss the *dynamic programming* technique, the until now most commonly used technique for designing fixed-parameter algorithms for structural parameters. For several decades dynamic programming has been used to design exact algorithms, in most cases polynomial-time algorithms, such as in the fields of sequence comparison and tree alignments [182, 118, 90, 84]. Several well-known exact algorithms for NP-hard problems are based on dynamic programming, such as the Bellman algorithm and the Held-Karp algorithm for the Traveling Salesman problem [21, 106] or the Dreyfus-Wagner algorithm solving Minimum Steiner Tree in Graphs [66].

Roughly speaking, dynamic programming works in a "bottom-up" fashion: It starts by computing solutions to the smallest subinstances, and continues to larger and larger subinstances until the input instance is solved. During the bottom-up process, the solutions to smaller subinstances are stored in some

tables and these stored solutions are then used for computing the solutions of larger subinstances. For more details on dynamic programming on its own, we refer to standard algorithm textbooks [45, 101, 121, 161, 171]. We will illustrate the benefits of dynamic programming for parameterization by structure by giving two fixed-parameter algorithms for MULTICOMMODITY DEMAND FLOW IN TREES and MULTICUT IN TREES in the next two chapters.

We conclude this section with a remark on the correlation between the complexity of the determination of a distance parameter and the "quality" of the determined distance parameter. For example, consider VERTEX COVER parameterized by treewidth ω and the number d of edges between a graph and a tree considered above. For both parameterizations there are fixed-parameter algorithms solving VERTEX COVER in $O(2^{\omega} \cdot n)$ [169] and $O(2^{d} \cdot n)$ time, respectively, where n denotes the number of graph vertices. As mentioned above, treewidth ω is always bounded from above by $d+1$. This means that the parameterization with treewidth as parameter seems to be more interesting. In contrast, however, it is NP-complete to determine the treewidth of a graph while parameter d is computable in polynomial time by applying standard spanning tree algorithms. As a consequence, in order to give an efficient fixed-parameter algorithm with respect to a structural parameter, we should not consider the two parts of the framework as separate, but rather as interrelated: The computational complexity of both parts—the determination of the parameter and the algorithm based on this parameter—has to be taken into account.

In the next two sections we provide two case studies for parameterizing by distance from triviality.

12.2 Case Study 1: Clique

The CLIQUE problem is defined as follows:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer ℓ.
>
> **Question:** Does G contain a *clique*, i.e., a complete subgraph, with at least ℓ vertices?

This problem can also be formulated as a graph modification problem.

> **Input:** A graph $G = (V, E)$ and a nonnegative integer ℓ.
>
> **Task:** Find a set V' of vertices with $|V'| \leq |V| - \ell$ whose removal transforms G into a *clique*?

CLIQUE is one of the classical NP-complete problems [115] and it is not polynomial-time approximable within a factor of $|V|^{1-\epsilon}$ for any $\epsilon > 0$, unless any problem in NP can be solved in probabilistic polynomial time [104]. Concerning parameterized complexity, CLIQUE is W[1]-complete with respect to parameter ℓ [65]. Here we exhibit fixed-parameter tractability with respect to the distance from a trivial case.

Our trivial case is the class of *cluster graphs*: graphs which are a disjoint union of cliques. Clique can be trivially solved in linear time on such graphs. We examine Clique on graphs which are "almost" cluster graphs, namely, on graphs which are cluster graphs with k edges added. Observe that determining the distance parameter, i.e., finding the added k edges, is exactly the Cluster Deletion problem defined in Chapter 7. An algorithm running in $O(1.53^k + |V|^3)$ time for this problem was given by Gramm et al. [86, 85].

It remains to show how to solve Clique for the "almost cluster graph" G after identifying the k added edges and the corresponding cluster graph G'. If the largest clique in G is not one which is already contained in G', then each of its vertices must have gained in degree by at least one compared to G'. This means it can only be formed by a subset of the up to $2k$ vertices "touched" by the added edges. Hence, we solve Clique for the subgraph of G induced by the up to $2k$ vertices which are endpoints of the added edges. This step can be done for example by using Robson's algorithm for Independent Set [159] on the complement graph in $O(1.22^{2k}) = O(1.49^k)$ time, which is dominated by the above time bound for the Cluster Deletion subproblem. The largest clique for G is simply the larger of the clique found this way and the largest clique in G'. We obtain the following theorem:

Theorem 12.1. Clique *for a graph $G = (V, E)$ which is a cluster graph with k edges added can be solved in $O(1.53^k + |V|^3)$ time.*

12.3 Case Study 2: Power Dominating Set

The Power Dominating Set problem is motivated from applications in electric networks [105]. The task is to seek for a minimum cardinality set of vertices such that, by placing monitoring devices (so-called *PMUs*) at the vertices in this set, all vertices are *observed*. The vertex set is call *power dominating set*. The rules for observation are:

- **Observation Rule 1 (OR1)**: A vertex in the power domination set observes itself and all of its neighbors.
- **Observation Rule 2 (OR2)**: If an observed vertex v of degree $d \geq 2$ is adjacent to $d - 1$ observed vertices, then the remaining unobserved neighbor vertex becomes observed as well.[3]

We can now formulate the Power Dominating Set problem:

Input: A graph $G = (V, E)$ and a nonnegative integer ℓ.

Task: Find a *power dominating set* of size at most ℓ, that is, a subset $M \subseteq V$ of vertices such that by placing a PMU in every $v \in M$, all vertices in V are observed?

[3] This is motivated by Kirchhoff's law in electrical network theory. The original definition of Haynes et al. [105] is a bit more complicated and the equivalent definition presented here is due to Kneis et al. [120].

It is shown in [99, 151] that POWER DOMINATING SET is equivalent to a graph modification problem called VALID ORIENTATION WITH MINIMUM ORIGIN.

POWER DOMINATING SET is NP-complete [105] and is not polynomial-time approximable within $(1-\epsilon)\cdot\ln|V|$ for any $\epsilon > 0$ [73, 99]. Parameterized by the size of power dominating set, POWER DOMINATING SET is W[2]-hard [99, 120]. There is a simple algorithm solving POWER DOMINATING SET in trees in linear time [99, 151].[4] Since we use this algorithm as a building block for our result, we briefly sketch how it proceeds. This algorithm works bottom-up from the leaves and adds every vertex to the power dominating set which has at least two unobserved children. Then it updates the observation status of all vertices according to the two observation rules and prunes completely observed subtrees, since they no longer affect observability of other vertices.

Our goal is now to find an efficient algorithm for input graphs that are "nearly" trees. More precisely, we aim for a fixed-parameter algorithm for graphs which are trees with k edges added.

As a first step we present a simple algorithm with quadratic running time for the case of one single added edge.

Lemma 12.1. POWER DOMINATING SET *for a graph* $G = (V, E)$ *with* $n := |V|$ *which is a tree with one edge added can be solved in* $O(n^2)$ *time.*

Proof. Graph G contains exactly one cycle and a collection of trees T_i touching the cycle at their roots.

We use the above mentioned linear-time algorithm to find an optimal solution for each T_i. When it reaches the root r_i, several cases are possible:

- The root r_i needs to be in the power dominating set M, and we can remove it. This breaks the cycle, and we can solve the remaining instance in linear time.

- The root r_i is not in M, but already observed. Then all children of r_i in T_i except for at most one are observed, or we would need to take r_i into M. Then, we can remove T_i except for r_i and except for the unobserved child, if it exists. The root r_i remains as an observed degree-2 or degree-3 vertex on the cycle.

- The root r_i still needs to be observed. This is only possible if it has exactly one child in T_i which is unobserved since otherwise r_i either would be in M, or be observed. As in the previous case, we keep r_i and the unobserved child, and the rest of T_i can again be removed.

Then, we apply OR2 to the vertices on the cycle, as long as possible. If there are still unobserved vertices, then at least one vertex on the cycle has to be added to M. We simply try each vertex. After each choice, the rest of the cycle decomposes into a tree, after adding the selected vertex to M, applying

[4] Haynes et al. [105] give a far more complicated linear-time algorithm for POWER DOMINATING SET in trees.

OR1 and OR2, and deleting the selected vertex. The remaining tree can be handled in linear time. From all possible choices, we keep the one leading to a minimal M. Since there are $O(n)$ vertices on the cycle, this takes $O(n^2)$ time. □

Lemma 12.1 is applicable whenever each vertex is part of at most one cycle. We now generalize this result.

Theorem 12.2. POWER DOMINATING SET *for a graph which is a tree with k edges added is fixed-parameter tractable with respect to k.*

Proof. We first treat all trees which are attached in single points to cycles as in the proof of Lemma 12.1. What remains are degree-2 vertices, degree-3 vertices with a degree-1 neighbor, and other vertices of degree 3 or greater, the *joints*. For a vertex v, let $\deg(v)$ denote its degree, that is, the number of its adjacent vertices. We branch into several cases for each joint:

- The joint v is in M. We can prune it and its incident edges.

- The joint v is not in M. Note that the only effect v can still have is that a neighbor of v becomes observed from application of OR2 applied to v. We branch further into $\deg(v) \cdot (\deg(v) - 1)$ cases for each pair (w_1, w_2) of neighbors of v with $w_1 \neq w_2$. In each branch, we omit the edges between v and all neighbors of v except w_1 and w_2. Clearly any solution of such an instance provides a solution for the unmodified instance. Furthermore, it is not too hard to show that if the unmodified instance has a solution of size s, then on at least one branch we will also find a solution of size s. To see this, consider a solution M for the unmodified problem. Vertex v is observed; this can only be either because a neighbor w_1 of v was put into M, or because there is a neighbor w_1 of v such that vertex v became observed by applying OR2 to w_1. Furthermore, as mentioned, there can be at most one vertex w_2 which becomes observed by OR2 applied to v. Then M is also a valid solution for the branch corresponding to the pair (w_1, w_2).

In each of the less than $(\deg(v))^2$ branches we can eliminate the joint. If we branch for all joints in parallel, we end up with an instance where every connected component is a tree or a cycle with attached degree-1 vertices, which can be solved in $O(n^2)$ time. The number of cases to distinguish is $\prod_{v \text{ is joint}} (\deg(v))^2$. Since there are at most $2k$ joints, each of maximum degree k, the total running time is roughly bounded by $O(n^2 \cdot 2^{4k \log k})$, confirming fixed-parameter tractability. □

Remark: Note that a tree with k edges added has treewidth bounded by $k+1$ [25]. We refer to [99, 151] for a fixed-parameter algorithm for POWER DOMINATING SET with treewidth ω as parameter. Given a graph G with n vertices and given a tree decomposition of G with width ω, the algorithm runs in $O(c^{\omega^2} \cdot n)$ time with a very big constant c.

12.4 Concluding Remarks

The art of parameterizing problems is of key importance to better understand and cope with computational intractability. In contrast to approximation algorithms where the approximation ratio is tied to the optimization goal, we have more than one possible parameterizations of a given problem. The parameterization with distance from triviality as parameter seems to be a natural way of parameterizing problems and opens new views on well-known hard problems. In particular, consider problems such as POWER DOMINATING SET which are fixed-parameter intractable with respect to the solution size and hard to approximate. Parameterizing by structural parameters as we have done for POWER DOMINATING SET in Section 12.3 and in [99] seems to be the only reasonable way to attack these problems.

Determining structural parameters, it is worth remarking that the tractable trivial case may refer to polynomial-time solvability as well as fixed-parameter tractability.[5] An example for the latter case is DOMINATING SET in planar graphs which is fixed-parameter tractable [7, 9]. These results were extended to graphs of bounded genus [53, 68], genus here measuring the distance from the "trivial case" (because settled) planar graphs. Also observe that with the advancement of scientific achievements also the range of triviality may be extended and new parameterizations might emerge in this process.

Furthermore, for some problems, the distance from triviality could be a combination of more than one parameters. Consider the MULTICUT IN GRAPHS problem which is the generalization of MULTICUT IN TREES (Chapter 8) on general graphs: Given a graph and a set of vertex pairs, find a minimum cardinality set of edges whose removal disconnects every input vertex pair. On the one hand, from the fact that MULTICUT IN TREES is NP-complete, we know that MULTICUT IN GRAPHS is fixed-parameter intractable with respect to treewidth. On the other hand, it was shown in [49] that this problem is NP-complete even if there are only three input vertex pairs, which excludes the fixed-parameter tractability with the number of vertex pairs as parameter. However, we can easily solve the problem in trees when there are only constantly many input vertex pairs: We need to remove at most c edges to separate c vertex pairs in trees. By trying all possibilities of removing at most c edges, we can compute the optimal solution in polynomial time. We have as the triviality a tree with constantly many vertex pairs. Using the combination of treewidth ω and the number of input vertex pairs l measuring the distance from triviality, MULTICUT IN GRAPHS can be solved in $O(l^{l+\omega+1} \cdot (|V| + |E|))$ time for a given graph $G = (V, E)$ with a given tree decomposition with width ω [94].

Finally we emphasize that not only graph problems fall into the parameterizing by distance from triviality framework. An example with SATISFIABILITY is given by Szeider [166, 167]. It is easy to observe that a boolean formula in conjunctive normal form which has a matching between variables and clauses that matches all clauses is always satisfiable. For a formula F, considered as

[5] The latter being of particular interest when attacking W[1]-hard problems as shown for the W[1]-complete CLIQUE in Section 12.2.

a set of m clauses over n variables, define the *deficiency* as $\delta(F) := m - n$. The maximum deficiency is $\delta^*(F) := \max_{F' \subseteq F} \delta(F')$. Szeider shows that the satisfiability of a formula F can be decided in $O(2^{\delta^*(F)} \cdot n^3)$ time [166]. Note that a formula F with $\delta^*(F) = 0$ has a matching as described above. Again, $\delta^*(F)$ is a structural parameter measuring the distance from triviality in our sense.

Chapter 13

Multicommodity Demand Flow in Trees

In this chapter, we study an NP-hard (and MaxSNP-hard) problem in trees—MULTICOMMODITY DEMAND FLOW—from the viewpoint of structural parameterization. This problem deals with demand flows between pairs of vertices and tries to maximize the value of routed flows. We prove the fixed-parameter tractability of this problem with respect to the maximum number of the input flows at any tree vertex. Herein, we partly follow [98].

13.1 Problem Definition and Previous Results

The MULTICOMMODITY DEMAND FLOW IN TREES (MDFT) problem is defined as follows.

> **Input:** A tree network $T = (V, E)$, where each edge e is associated with a positive integer $c(e)$ as its capacity, and a collection F of flows which is encoded as a list of pairs of vertices of T,
>
> $$F = \{f_i \,|\, f_i := (u_i, v_i), u_i \in V, v_i \in V, u_i \neq v_i, 1 \leq i \leq l\}.$$
>
> Each flow $f \in F$ has associated an integer demand value $d(f)$ and a real valued profit $p(f)$.
> **Task:** Remove a subset of the flows in F such that the set F' of the remaining flows is routable and maximizes $\sum_{f \in F'} p(f)$. A subset $F' \subseteq F$ is *routable* (in T) if the flows in F' can be simultaneously routed without violating any edge capacity of the tree.[1]

The tree T is termed the *supply tree*. Throughout this chapter let $n := |V|$ and $l := |F|$. The vertices u, v are called the two *endpoints* of the flow $f =$

[1] That is, for any edge e the sum of the demand values of the flows routed through e does not exceed $c(e)$.

(u, v). Note that, for a tree, the path between two distinct vertices is uniquely determined and can be found in linear time. Thus, we can assume that each flow $f = (u, v) \in F$ is given as a path between u and v. We use F_v to denote the set of demand flows passing through vertex v. A demand flow *passes through* vertex v if it has v as one of its endpoints or v lies on the flow's path.

Multicommodity flow problems find many applications as in telecommunication, routing, and railroad transportation [147, 174]. For example, Ahuja et al. [4] modeled a problem called "racial balancing of schools" as a multicommodity flow problem, where one seeks for an optimal assignment of students to schools such that the total distance traveled by the students is minimized and the desired ethnic composition for each school is achieved. We refer to Chekuri et al. [39] and Garg et al. [82] with respect to further relevance of studying MDFT.

It follows from the results of Garg et al. [82] that MDFT is NP-complete and MaxSNP-hard even in the case of unit demands and unit profits. For this special case of unit demands and unit profits, they gave a factor-two polynomial-time approximation algorithm. Chekuri et al. [39] showed that in the special case of MDFT with unit demands but arbitrary profits the natural linear programming relaxation yields a factor-four approximation. They further showed that, under the assumption that the maximum flow demand is at most the minimum edge capacity, in the general case with arbitrary demands and arbitrary profits the natural linear programming relaxation of MDFT has an integrality gap of at most 48, improving previous work of Chakrabarti et al. [35] which dealt with path instead of tree networks. Both papers, however, concentrate on the polynomial-time approximability of MDFT and its still NP-complete special cases.

Concerning exact algorithms, we are only aware of the recent work of Anand et al. [12] where (a special case of) MDFT is referred to as "call admission control problem." For the special case of MDFT restricted to instances with unit demands and unit profits, they presented a fixed-parameter algorithm with run time $O(2^d \cdot d! \cdot |I|^{O(1)})$, where $|I|$ denotes the size of the input instance and d denotes the number of flows to be rejected in order to "enable" all remaining flows. Hence, their version of MDFT is fixed-parameter tractable with respect to parameter d defined in this way.

By way of contrast, for the general MDFT problem, we subsequently show that it is fixed-parameter tractable with respect to the new parameter

$$k := \max_{v \in V} |F_v|.$$

That is, parameter k denotes the maximum number of the input flows passing through any vertex of the given tree network. We call the parameter k the *vertex cutwidth* of the given instance. The motivation for considering this parameterization is that, if all demand flows are vertex-disjoint, then MDFT can be solved trivially: We can determine the routability of every demand flow independently from other demand flows. If the demand of a flow exceeds the minimum capacity of the edges it is passing then this demand flow cannot be routed. Thus, the

$a := a_1 a_2 \dots a_{k_v}$	D_e
$00\dots00$	
$00\dots01$	
$\vdots$	
$11\dots11$	

Figure 13.1: Table D_e for edge $e = (u, v)$ with $F_v := \{f_1^v, f_2^v, \dots, f_{k_v}^v\}$ with $k_v \leq k$

instance settings with vertex-disjoint demand flows are a "triviality" for MDFT and the distance of an input instance from this triviality is measured by the vertex cutwidth k considered here.

Our fixed-parameter algorithm relies on the dynamic programming technique and has a runtime of $O(2^k \cdot l \cdot n)$. This fixed-parameter algorithm is superior to approximative solutions whenever k is of limited size, a realistic assumption for several applications. Without going into details, we remark that MDFT restricted to paths and unit flow demands can be solved in polynomial time [39] and if, in addition, all flows have unit profits, then there is a linear-time solving algorithm [98].

13.2 The Algorithm

The fixed-parameter algorithm is based on dynamic programming. We begin with some agreements that simplify the presentation and analysis of the algorithm. Then, we describe the algorithm in detail and after that we prove its correctness and analyze its time complexity.

13.2.1 Agreements and Basic Tools

We assume that we deal with arbitrarily rooted trees. Thus, an edge $e = (u, v)$ reflects that u is the parent vertex of v. In particular, using the rooted tree structure, we will solve MDFT in a bottom-up fashion by dynamic programming from the leaves to the root. In this context, we use $T[v]$ to denote the subtree of the input tree rooted at vertex v.

The core idea of the dynamic programming is based on the following definition of tables which are used throughout the algorithm. For each edge $e = (u, v)$ with $F_v := \{f_1^v, f_2^v, \dots, f_{k_v}^v\} \subseteq F$ we construct a table D_e as illustrated in Figure 13.1. Table D_e has 2^{k_v} rows which correspond to all possible k_v-vectors a over $\{0, 1\}$, i.e., binary vectors having k_v components. Each of these vectors represents a *route schedule* for the flows in F_v. The ith component a_i of a corresponds to flow f_i^v, and $a_i = 0$ means that we do not route flow f_i^v and $a_i = 1$ means that we do route f_i^v. Furthermore, to refer to the set of routed flows, we define $r(a) := \{i \,|\, a_i = 1, 1 \leq i \leq k_v\}$. Table entry $D_e(a)$ stores the maximum

profit which we can achieve according to the route schedule encoded by a with the flows which have at least one of their endpoints in $T[v]$.

13.2.2 Dynamic Programming Algorithm

The algorithm works bottom-up from the leaves to the root. Having computed all tables D_e for the edges e connected to the root vertex, MDFT will be solved easily as pointed out later on. The algorithm starts with "leaf edges" which are initialized as follows. For an edge $e = (u, v)$ connecting a leaf v with its parent vertex u, the table entries for the at most 2^k rows a can be easily computed as

$$D_e(a) := \begin{cases} 0, \text{ if } c(e) < \sum_{i \in r(a)} d(f_i^v); \\ \sum_{i \in r(a)} p(f_i^v), \text{ otherwise.} \end{cases}$$

Then, the main algorithm consists of distinguishing between three cases.

Case 1. Consider an edge $e = (u, v)$ connecting two non-leaf vertices u and v where v has only one child w connected to v by edge $e' = (v, w)$.

The sets of flows passing through vertices u, v, and w are denoted by $F_u := \{f_1^u, \ldots, f_{k_u}^u\}$, $F_v := \{f_1^v, \ldots, f_{k_v}^v\}$, and $F_w := \{f_1^w, \ldots, f_{k_w}^w\}$. Moreover, we use F_e and $F_{e'}$ to denote the sets of flows passing through e and e' and we have $F_e = F_u \cap F_v$ and $F_{e'} = F_v \cap F_w$.

First, if $F_e \cap F_{e'} = \emptyset$, then the given instance can be divided into two smaller instances, one consisting of subtree $T[v]$ and the flows inside it and the other consisting of the original tree without the vertices below v and the flows therein. The optimal solution for the original instance then is the sum of the optimal solutions of the two smaller instances. An optimal solution of the first smaller instance is already computed and it is obtained from a maximum entry of table $D_{e'}$. In order to compute an optimal solution of the second smaller instance, we can treat v as a leaf and proceed as for leaf edges as described above.

Second, if $F_e \cap F_{e'} \neq \emptyset$, then there are some flows passing through both e and e'. Recall that entry $D_e(a)$ for a k_v-vector a shall store the maximum profit with respect to the route schedule encoded by a that can be achieved by the flows with at least one of their endpoints in $T[v]$. We partition F_v into two sets, $F_v \cap F_w$ and $F_v \setminus F_w$. The value of $D_e(a)$ is thus the sum of the maximum of the entries of $D_{e'}$ which have the same route schedule for the flows in $F_v \cap F_w$ as encoded in a, and the profit achieved by the flows in $F_v \setminus F_w$ obeying the route schedule encoded by a. Let $B^v := \{i \,|\, f_i^v \in (F_v \cap F_w)\}$, $B^w := \{i \,|\, f_i^w \in (F_v \cap F_w)\}$, and $j := |B^v| = |B^w|$.[2] To easier obtain the maximum of the entries of $D_{e'}$ which have the same route schedule for the flows in $F_v \cap F_w$, we condense table $D_{e'}$ with respect to B^w. The *condensation of $D_{e'}$ with respect to B^w* is to keep only the components of the k_w-vector a' of $D_{e'}$

[2] Clearly, B^v and B^w refer to the same sets of demand flows. Note, however, that they may differ due to different "naming" of the same flow in the two tables D_e and $D_{e'}$.

which correspond to the demand flows in $F_v \cap F_w$. More precisely, for a route schedule $\overline{a'}$ condensed with respect to B^w, we obtain

$$D_{e'}(\overline{a'}) := \max\{D_{e'}(a') \mid \overline{a'} = \pi_{(B^w)}(a')\}.$$

Herein, $\pi_{(B^w)}(a')$ returns the projection of a' onto the j components of a' that correspond to B_w. Then, using $A := \{i \mid f_i^v \in F_e\}$ to refer to the set of the flows in F_v passing through edge e, the entries of D_e are computed as follows.

$$D_e(a) := \begin{cases} 0, \text{ if } c(e) < \sum_{i \in A} d(f_i^v); \\ D_{e'}(\pi_{(B^v)}(a)) + \sum_{i \in (r(a) \setminus B^v)} p(f_i^v), \text{ otherwise.} \end{cases}$$

Obeying the route schedule encoded by the k_v-vector a, the terms $D_{e'}(\pi_{(B^v)}(a))$ and $\sum_{i \in (r(a) \setminus B^v)} p(f_i^v)$ denote the profits achieved by the flows in $F_v \cap F_w$ and in $F_v \setminus F_w$, respectively.

Case 2. Consider an edge $e = (u, v)$ connecting two non-leaf vertices u and v where v has two children w_1 and w_2 connected to v by edges $e' = (v, w_1)$ and $e'' = (v, w_2)$.

We use $F_u := \{f_1^u, \ldots, f_{k_u}^u\}$, $F_v := \{f_1^v, \ldots, f_{k_v}^v\}$, $F_{w_2} := \{f_1^{w_1}, \ldots, f_{k_{w_1}}^{w_1}\}$, and $F_{w_2} := \{f_1^{w_2}, \ldots, f_{k_{w_2}}^{w_2}\}$ to denote the sets of flows passing through vertices u, v, w_1, and w_2. As in Case 1, $F_e = F_u \cap F_v$, $F_{e'} = F_v \cap F_{w_1}$, and $F_{e''} = F_v \cap F_{w_2}$.

With the same argument as in Case 1, if one of $F_e \cap F_{e'}$ and $F_e \cap F_{e''}$ is empty, we can divide the given instance into two smaller instances and solve them separately. Thus, we assume that they are not empty. Similar to Case 1, we "partition" F_v into $F_v \cap F_{w_1}$, $F_v \cap F_{w_2}$, and $(F_v \setminus F_{w_1}) \setminus F_{w_2}$. For a k_v-vector a, one might simply set $D_e(a)$ equal to the sum of the maximum of the entries of $D_{e'}$ which have the same route schedule as encoded in a for the flows in $F_v \cap F_{w_1}$, the maximum of the entries of $D_{e''}$ which have the same route schedule as encoded in a for the flows in $F_v \cap F_{w_2}$, and the profit achieved by the flows in $(F_v \setminus F_{w_1}) \setminus F_{w_2}$ obeying the route schedule encoded by a. However, if $(F_v \cap F_{w_1}) \cap (F_v \cap F_{w_2}) \neq \emptyset$, that is, due to the tree structure, $F_{w_1} \cap F_{w_2} \neq \emptyset$, then the edges e' and e'' have some common flows. Then, for each flow between $T[w_1]$ and $T[w_2]$ scheduled to be routed in both subtrees, we have to once subtract its profit from the sum to avoid double counting . Let

$$\begin{aligned} B_1^v &:= \{i \mid f_i^v \in (F_v \cap F_{w_1})\}; & B_1^{w_1} &:= \{i \mid f_i^{w_1} \in (F_v \cap F_{w_1})\}; \\ B_2^v &:= \{i \mid f_i^v \in (F_v \cap F_{w_2})\}; & B_2^{w_2} &:= \{i \mid f_i^{w_2} \in (F_v \cap F_{w_2})\}; \\ B_3^{w_1} &:= \{i \mid f_i^{w_1} \in (F_{w_1} \cap F_{w_2})\}; & B_3^{w_2} &:= \{i \mid f_i^{w_2} \in (F_{w_1} \cap F_{w_2})\}. \end{aligned}$$

Note that $|B_1^v| = |B_1^{w_1}|$, $|B_2^v| = |B_2^{w_2}|$, $|B_3^{w_1}| = |B_3^{w_2}|$, $B_3^{w_1} \subseteq B_1^{w_1}$, and $B_3^{w_2} \subseteq B_2^{w_2}$. As in Case 1, we condense $D_{e'}$ and $D_{e''}$. More specifically, we condense $D_{e'}$ with respect to $B_1^{w_1}$ and $D_{e''}$ with respect to $B_2^{w_2}$:

$$\begin{aligned} D_{e'}(\overline{a'}) &:= \max\{D_{e'}(a') \mid \overline{a'} = \pi_{(B_1^{w_1})}(a')\}; \\ D_{e''}(\overline{a''}) &:= \max\{D_{e''}(a'') \mid \overline{a''} = \pi_{(B_2^{w_2})}(a'')\}. \end{aligned}$$

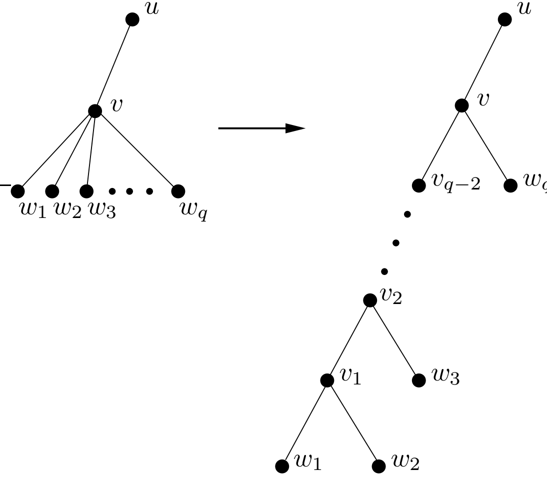

Figure 13.2: Transformation of an arbitrary tree into a binary tree.

Then, using $A := \{i \,|\, f_i^v \in F_e\}$, the entries of D_e are computed as follows.

$$D_e(a) := \begin{cases} 0, \text{ if } c(e) < \sum_{i \in A} d(f_i^v); \\ \alpha + \beta, \text{ otherwise.} \end{cases}$$

Herein,

$$\begin{aligned} \alpha &:= D_{e'}(\pi_{(B_1^v)}(a)) + D_{e''}(\pi_{(B_2^v)}(a)) - \gamma, \\ \beta &:= \sum_{i \in (r(a) \setminus (B_1^v \cup B_2^v))} p(f_i^v), \\ \gamma &:= \sum_{i \in B_3^{w_1}, \pi_{(\{i\})}(a) = 1} p(f_i^{w_1}). \end{aligned}$$

In order to avoid double counting of common flows of edges e' and e'', γ has been subtracted in the above determination of D_e.

Case 3. Consider an edge $e = (u, v)$ connecting two non-leaf vertices u and v where v has $q > 2$ children $w_1, w_2, \ldots, w_q$.

We add $q - 2$ new vertices $v_1, v_2, \ldots, v_{q-2}$ to T and we transform T into a binary tree as illustrated in Figure 13.2. Each of these new vertices has exactly two children. Vertex v_1 is the parent vertex of w_1 and w_2, v_2 is the parent vertex of v_1 and w_3, and so on. Thus, v becomes the parent vertex of v_{q-2} and w_q. The edge between w_i and its new parent vertex is assigned the same capacity as the edge between w_i and v in the original tree. The edges $(v, v_{q-2}), (v_{q-2}, v_{q-1}), \ldots, (v_2, v_1)$ between the new vertices obtain unbounded capacity. The flows have the same endpoints as in the original instance. It is easy to see that the solutions for the new and the old tree are the same, and thus Case 1 and Case 2 suffice for handling the new binary tree. This concludes the description of the dynamic programming algorithm.

13.2.3 Main Result

The above described dynamic programming algorithm leads to the following.

Theorem 13.1. MDFT *can be solved in* $O(2^k \cdot l \cdot n)$ *time, where* k *denotes the maximum number of demand flows passing through any vertex of the given supply tree, i.e.,* $k := \max_{v \in V} |F_v|$.

Proof. The correctness of the algorithm basically follows directly from its description. To this end, however, note that when the D-tables of all edges of the supply tree are computed, we may easily determine the final optimum by comparing the tables of the without loss of generality at most two edges leading to the root. Moreover, by means of a top-down traversal from the root to the leaves we can easily determine the actual subset of routable flows that led to the overall maximum profit.

Concerning the algorithm's runtime, observe that table D_e for edge $e = (u, v)$ has at most $O(2^k)$ entries. For a vertex v with more than two children, as described in Section 13.2.2, we add some new vertices to construct a binary tree. The resulting tree at most doubles in size. Moreover, since $F_{v'} \subseteq F_v$ for each of the new vertices v', the D-tables of the new edges have at most $O(2^k)$ entries. Assuming that all basic set operations such as union, set minus, etc. between two sets having at most l elements can be done in $O(l)$ time, the computation of a new table from its at most two child tables takes $O(2^k \cdot l)$ time.

Altogether, the algorithm then takes $O(2^k \cdot l \cdot n)$ time. □

13.3 Concluding Remarks

Employing dynamic programming, we obtained a fixed-parameter tractability result for Multicommodity Demand Flow in Trees with respect to the vertex cutwidth parameter. This result complements previous work mainly dealing with the polynomial-time approximability of this and related problems. Moreover, this exact algorithm is conceptually simple enough to allow easy implementation and may appear as profitable alternative to existing approximation algorithms. Clearly, the immediate challenge is to significantly improve the exponential time bound of our algorithm.

Note that the vertex cutwidth parameter in our fixed-parameter tractability result relates to the maximum number of flows passing through any *vertex* of the input tree. The natural question arises what happens when we consider the *edge cutwidth* by replacing "vertex" by "*edge*." Somewhat surprisingly, a simple adaption of the reduction used in [82, Theorem 4.2] to show the NP-completeness of MDFT shows that MDFT is NP-complete even when there are at most six demand flows passing through an edge. Hence there is no hope for fixed-parameter tractability with respect to the edge cutwidth parameter defined as "maximum number of demand flows passing through any edge of the tree network." Finally, it seems worth studying the complexity of MDFT when parameterized by the number of the demand flows through any vertex

in a *solution* instead of the vertex cutwidth parameter, namely the number of demand flows through any vertex in the *input* (as we did here). We leave it as an open question to determine the parameterized complexity for this modified parameterization.

Chapter 14

Weighted Multicut in Trees

In Chapters 8 and 9 we have shown the fixed-parameter tractability of the MULTICUT IN TREES problem with respect to the number of removed edges by two different means, kernelization and a depth-bounded search tree. Here, we will provide a new parameterization for this problem with respect to a structural parameter which is defined in a similar way as the *vertex cutwidth* parameter for MULTICOMMODITY FLOW IN TREES in Chapter 13. This fixed-parameter algorithm solves actually the *weighted* version of MULTICUT IN TREES which is defined as follows.

> WEIGHTED MULTICUT IN TREES
> **Input:** An undirected tree $T = (V, E)$, $n := |V|$, and a collection H of h pairs of vertices in V, $H = \{(v_i, u_i) \mid v_i, u_i \in V, v_i \neq u_i, 1 \leq i \leq h\}$. Each edge $e \in E$ has a positive real weight $w(e) > 0$.
> **Task:** Find a subset E' of E with minimum total weight whose removal separates each pair of vertices in H.

We call the uniquely determined path in T between vertices u and v of a vertex pair $(u, v) \in H$ the *demand path* of (u, v).

Since (unweighted) MULTICUT IN TREES is a special case of WEIGHTED MULTICUT IN TREES where $w(e) = 1$ for all $e \in E$, we can infer that the weighted case is also NP-complete and MaxSNP-hard [82]. With arbitrary edge weights, the search tree algorithm and the data reduction given in Chapters 8 and 9 cannot be directly applied to the weighted case. We now will present a fixed-parameter algorithm solving the weighted case with respect to the following parameter:

$$k := \text{the maximum number of demand paths passing through some tree vertex.}$$

The algorithm runs in $O(3^k \cdot h \cdot n)$ time. Both of this parameter and the vertex cutwidth parameter in Chapter 13 measure the number of paths over some vertex. Thus, we call this parameter k vertex cutwidth as well. Observe that this

parameterization is also motivated by the observation that we can trivially solve the problem if all demand paths in H are pairwise vertex-disjoint. Therefore, this is a "triviality" for the WEIGHTED MULTICUT IN TREES problem and the vertex cutwidth parameter measures the distance of a given instance from this triviality.

Partly following [96], we first introduce a special variant of the SET COVER problem, so-called TREE-LIKE WEIGHTED SET COVER (TWSC), which is more general than WEIGHTED MULTICUT IN TREES. In Section 14.1 we present some complexity results for this problem. Then, we show that there is a parameterized reduction from WEIGHTED MULTICUT IN TREES with vertex cutwidth as parameter to TWSC with subset size as parameter. Thus, the fixed-parameter algorithm given in Section 14.2 for TWSC can be applied to solve WEIGHTED MULTICUT IN TREES as well. Besides its relation to MULTICUT IN TREES, TWSC is motivated by applications in tree decomposition based computing [22] and bioinformatics [148]. In Section 14.2, we present a fixed-parameter algorithm solving TWSC with respect to the parameter subset size. Together with the parameterized reduction this algorithm implies the above mentioned $O(3^k \cdot h \cdot n)$ time algorithm for WEIGHTED MULTICUT IN TREES.

14.1 Multicut in Trees and Tree-Like Set Cover

We introduce an NP-complete special case of WEIGHTED SET COVER and then show its relation to WEIGHTED MULTICUT IN TREES.

14.1.1 Tree-Like Weighted Set Cover (TWSC)

Basic Definitions. (WEIGHTED) SET COVER is one of the most prominent NP-complete problems [80]. The basic SET COVER problem (optimization version) is defined as follows:

> **Input:** A ground set $S = \{s_1, s_2, \ldots, s_n\}$ and a collection C of subsets of S, $C = \{c_1, c_2, \ldots, c_m\}$, $c_i \subseteq S$ for $1 \leq i \leq m$.
> **Task:** Find a subset C' of C with minimal cardinality which covers all elements in S, i.e., $\bigcup_{c \in C'} c = S$.

Assigning weights to the subsets and minimizing total weight of the collection C' instead of its cardinality, one naturally obtains the WEIGHTED SET COVER problem. We call C' the *minimum set cover* of S resp. the *minimum weight set cover*. Define the *occurrence* of an element $s \in S$ in C as the number of the subsets in C which contain s. An element with occurrence of one is called *unique*. SET COVER remains NP-complete even if the occurrence of each element is bounded from above by two [149].

Definition 14.1 (Tree-like subset collection).
Given a ground set $S = \{s_1, s_2, \ldots, s_n\}$ and a collection C of subsets of S, $C = \{c_1, c_2, \ldots, c_m\}$, $c_i \subseteq S$ for $1 \leq i \leq m$, we say that C is a tree-like *subset*

collection of S if we can organize the subsets in C in an unrooted tree T such that every subset one-to-one corresponds to a node of T and, for each element $s_j \in S$, $1 \leq j \leq n$, all nodes in T corresponding to the subsets containing s_j induce a subtree of T.

We call T the underlying *subset tree* and the property of T that, for each $s \in S$, the nodes containing s induce a subtree of T is called the *consistency property* of T. Observe that the consistency property also is of central importance in Robertson and Seymour's famous notion of tree decompositions of graphs [158, 119, 27, 28]. By results of Tarjan and Yannakakis [168], we can test whether a subset collection is a tree-like subset collection and, if yes, we can construct a subset tree for it in linear time. Therefore, in the following, we always assume that the subset collection is given in form of a subset tree. For convenience, we denote the nodes of the subset tree by their corresponding subsets. We define the *height* of a subset tree as the minimum height of all rooted trees which can be obtained by taking one node of the subset tree as the root.

Example 14.1. For $S = \{s_1, s_2, s_3\}$, the subset collection $C = \{c_1, c_2, c_3\}$ where $c_1 = \{s_1, s_2\}$, $c_2 = \{s_2, s_3\}$, and $c_3 = \{s_1, s_3\}$ is not a tree-like subset collection. These three subsets can only be organized in a triangle. By way of contrast, if $c_1 = \{s_1, s_2, s_3\}$ instead, then we can construct a subset tree (actually a path) with these three nodes and two edges, one between c_1 and c_2 and one between c_1 and c_3.

We now define the special variant of Set Cover considered here.

Tree-like Weighted Set Cover (TWSC):
Input: A ground set $S = \{s_1, s_2, \ldots, s_n\}$ and a tree-like collection C of subsets of S, $C = \{c_1, c_2, \ldots, c_m\}$, $c_i \subseteq S$ for $1 \leq i \leq m$, and $\bigcup_{1 \leq i \leq m} c_i = S$. Each subset in C has a positive real weight $w(c_i) > 0$ for $1 \leq i \leq m$. The weight of a subset collection is the sum of the weights of all subsets in it.
Task: Find a subset C' of C with minimum weight which covers all elements in S, i.e., $\bigcup_{c \in C'} c = S$.

Motivation and Previous Results. TWSC is motivated by the following two concrete applications in practice. The first application deals with dynamic programming on tree decompositions. It is well-known that graphs with small treewidth allow for efficient solutions of otherwise hard graph problems [27]. The core tool is dynamic programming on tree decompositions. As discussed in [22], the main bottleneck of this approach is memory space consumption which is exponential with respect to the treewidth. To attack this problem, one can try to minimize the redundancy of information stored by avoiding to keep *all* dynamic programming tables in memory. This can be formulated as TWSC, where the tree decomposition serves as the subset collection and each tree node is assigned a positive weight equal to the size of the dynamic programming table associated with it. With the help of the TWSC formalization, experiments

showed memory savings of around 80 to 90 % [22]. The second application arises in computational molecular biology. In their work on vertebrate phylogenomics, Page and Cotton [148] formulate the problem of locating gene duplications as TREE-LIKE UNWEIGHTED SET COVER: given a so-called species tree in which each node has a set of gene duplications associated with it, find the smallest set of nodes whose union includes all gene duplications. Assigning each node a positive weight equal to the minimum number of distinct "episodes" of duplications at this node and solving TREE-LIKE WEIGHTED SET COVER on this tree seems to be a prospective way to answer one of their open questions dealing with minimizing the number of episodes of all covered duplications.

SET COVER appears to be very hard from an algorithmic point of view. Unless NP has slightly super-polynomial time algorithms, the best polynomial-time approximation algorithm achieves approximation factor $\Theta(\ln n)$ [73]. In addition, from the viewpoint of parameterized complexity, the problem is known to be W[2]-complete [65] (with respect to the parameter "number of chosen covering sets") which excludes fixed-parameter tractability in this respect. The special case of TWSC where T should be a path instead of a tree has been well-studied under the name SET COVER WITH CONSECUTIVE ONES PROPERTY (SCC1P): A SET COVER instance can also be represented by a binary coefficient matrix M over $\{0, 1\}$ where the rows correspond to the elements in S, the columns correspond to the subsets in C, and an entry is set to "1" if the corresponding element is contained in the corresponding subset; otherwise, it is set to "0." A matrix M has the *consecutive ones property* if there is a permutation of its columns that leaves the "1"s appearing consecutively in every row. In the SCC1P problem, we deal with SET COVER instances whose coefficient matrices have the consecutive ones property. It is well-known that SCC1P can be solved in polynomial time [139, 173].

A Simple Solution for the Unweighted Case TREE-LIKE UNWEIGHTED SET COVER can be solved by a simple polynomial-time algorithm as described in the following. The following lemma will be helpful in developing our algorithm.

Lemma 14.1. *Given a tree-like subset collection C of S together with its underlying subset tree T, then each leaf of T is either a subset of its parent node or it has a unique element.*

Proof. Consider a leaf $c = \{s_1, s_2, \ldots, s_r\}$ of the subset tree. Let c' be its parent. We show that, if c is not a subset of c', i.e., $(c \setminus c') \neq \emptyset$, then c contains a unique element. Assume that c contains no unique element. Thus, for each $s_i \in (c \setminus c')$, $1 \leq i \leq k$, there is a subset c'' different from c which contains s_i. According to the consistency property of the subset tree, all subsets on the path from c to c'' must also contain s_i. Since the uniquely determined path from c to c'' must pass through c', s_i has to be in c'. This contradicts $s_i \in (c \setminus c')$. □

We process the subset tree T in a bottom-up manner, i.e., we begin with the leaves. By Lemma 14.1, each leaf of T is either a subset of its parent node or it contains a unique element from S. If a leaf c contains a unique element,

the only choice to cover this element is to put c into the set cover. Then, we delete c from T and c's elements from other subsets. If c is completely contained in its parent node, it is never better to take c into the set cover than to take its parent node. Hence, we can safely delete it from the subset tree. After processing its children, an internal node becomes a leaf and we can iterate the described process. Thus, we obtain the following result.

Proposition 14.1. Tree-like Unweighted Set Cover *can be solved in* $O(m \cdot n)$ *time.*

Page and Cotton [148] in their work on phylogenomics implicitly dealt with Tree-like Unweighted Set Cover only giving a heuristic solution seemingly unaware of the simple polynomial-time solvability.

Complexity Results for TWSC. The key idea of the above algorithm for Tree-like Unweighted Set Cover is that we never put a set into the desired subset collection C' which is a subset of other subsets in the collection C. However, if we associate each subset with an arbitrary positive weight and ask for the set cover with minimum weight, then this strategy is no longer valid.

Proposition 14.2. *The decision version of* Tree-like Weighted Set Cover *is NP-complete.*

Proof. TWSC is clearly in NP. To show its NP-hardness, we reduce the general (unweighted) Set Cover problem to it. Given a Set Cover instance with the ground set S and the subset collection C, we construct a new subset collection $C' := C \cup \{c_{m+1}\}$ with an additional subset $c_{m+1} := S$. All subsets in C' except c_{m+1} have unit weight, $w(c_i) = 1$ for $1 \leq i \leq m$, and $w(c_{m+1}) = m + 1$. Then, the ground set S and the new subset collection C' form an instance of TWSC, the underlying tree being a star with center c_{m+1}. □

Since this reduction is gap-preserving, by Feige's result for Set Cover [73] it directly follows that the best polynomial-time approximation for TWSC is $\Theta(\ln n)$ unless NP has slightly super-polynomial time algorithms. Moreover, because this reduction also preserves the total weight of the optimal solution in the sense of parameterized complexity theory [65], we also can infer W[2]-hardness for TWSC with respect to the parameter "total weight of the solution" of the set cover. This excludes fixed-parameter tractability for this parameterization [65]. The reduction above shows also that TWSC remains NP-complete even if the subset tree has height one. In the following, we will show that several other relevant variations of TWSC are also NP-complete. The first corollary follows from the NP-completeness of the variant of Set Cover where the occurrence of elements is bounded from above by 2 [149] and the reduction used above.

Corollary 14.1. *The decision version of* Tree-like Weighted Set Cover *is NP-complete even if the occurrence of each element from* S *in the subsets of the collection* C *is at most* 3.

Corollary 14.2. *The decision version of* TREE-LIKE WEIGHTED SET COVER *is NP-complete even if the underlying subset tree is a balanced binary tree.*

For more complexity results for TWSC and a proof of Corollary 14.2, we refer to [96]. A structural parameterization for TWSC with bounded occurrence has been studied in [95].

14.1.2 Weighted Multicut in Trees and TWSC

In the following, we show a parameterized reduction from WEIGHTED MULTICUT IN TREES to TREE-LIKE WEIGHTED SET COVER. Together with the fixed-parameter algorithm in Section 14.2 for TWSC, this parameterized reduction implies that WEIGHTED MULTICUT IN TREES is fixed-parameter tractable with respect to the vertex cutwidth parameter.

Theorem 14.1. *There is parameterized reduction from* WEIGHTED MULTICUT IN TREES *to* TREE-LIKE WEIGHTED SET COVER *running in $O(h \cdot n)$ time such that the maximum subset size of the* TREE-LIKE WEIGHTED SET COVER *instance is equal to the vertex cutwidth parameter of the* WEIGHTED MULTICUT IN TREES *instance.*

Proof. Given an instance of WEIGHTED MULTICUT IN TREES, we create a new tree T' by adding some new vertices to the input tree $T = (V, E)$, $n := |V|$. We replace each edge $e = \{u, v\} \in E$ with a new vertex w_e and we connect it by two edges with u and v. The set of these new vertices is denoted as V'. The new tree $T' = (V \cup V', E')$ has $2n-1$ vertices and $2n-2$ edges. Then, we create a set P containing the paths in T which connect the vertex pairs in H. For a vertex pair $(u_i, v_i) \in H$, there is a unique path p_i in T connecting u_i and v_i. We can determine p_i in $O(n)$ time and we put it into P. Furthermore, we create a set P_e for each $e \in E$ which contains the paths in P passing through e, and a set P_v for each $v \in V$ containing the paths in P passing through v. Note that P_v contains the paths starting or ending at v as well. The TREE-LIKE WEIGHTED SET COVER instance then consists of the ground set P and the subset collection $C := C_1 \cup C_2$, where $C_1 := \{P_e \,|\, e \in E\}$ and $C_2 := \{P_v \,|\, v \in V\}$. We have $\bigcup_{P_e \in C_1} P_e = P$. Each subset P_e in C_1 is defined to have the same weight as its corresponding edge e, i.e., $w(P_e) := w(e)$. Since WEIGHTED MULTICUT IN TREES asks for a subset of the edge set, we have to give the subsets P_v in C_2 a weight such that none of them will be in the minimum weight set cover: $w(P_v) := \sum_{e \in E} w(e) + 1$ for all $v \in V$. It is clear that C is a tree-like subset collection: The underlying subset tree is T' by associating the subsets $P_e \in C_1$ with the vertices $w_e \in V'$ and the subsets $P_v \in C_2$ with the vertices $v \in V$. The maximum subset size corresponds to the vertex cutwidth of the WEIGHTED MULTICUT IN TREES instance, the maximum number of paths passing through a vertex or an edge.

It is easy to see that an optimal solution $\{P_{e'_1}, P_{e'_2}, \ldots, P_{e'_l}\}$ for the TWSC instance corresponds to an optimal solution $\{e'_1, e'_2, \ldots, e'_l\}$ for the WEIGHTED MULTICUT IN TREES instance and vice versa. The runtime of the reduction is clearly $O(h \cdot n)$. □

14.2 Algorithm for TWSC

We show that TWSC is fixed-parameter tractable with respect to the parameter maximal subset size k, i.e., $k := \max_{c \in C}\{\,|c|\,\}$. This implies that the problem can be efficiently solved for small values of k. To facilitate the presentation of the algorithm, we will describe, in the first subsection, how to solve the problem for binary subset trees, an also NP-complete special case (cf. Corollary 14.2), and then, in the second subsection, we extend the described algorithm to arbitrary trees.

14.2.1 TWSC with Binary Subset Tree

The dynamic programming processes the underlying subset tree bottom-up, i.e., first the leaves, then the nodes having leaves as their children, and finally the root. For a given tree-like subset collection C with its underlying subset tree T, we define for each node c_i of T a set $A(c_i)$ which contains all elements occurring in the nodes of the subtree with c_i at the root:

$$A(c_i) := \bigcup_{c \in T[c_i]} c\,,$$

where $T[c_i]$ denotes the node set of the subtree of T rooted at c_i.

Moreover, we associate with each node c of T a table D_c. Table D_c has three columns, the first two corresponding to the two children of c and the third to c. The rows of the table correspond to the elements of the power set of c, i.e., there are $2^{k'}$ rows if $c = \{s_1, s_2, \ldots, s_{k'}\}$, $k' \leq k$. Figure 14.1 illustrates the structure of table D_c for a node c having two children c' and c''. Table D_c has $3 \cdot 2^{k'} = O(2^k)$ entries. Entry $D_c(x, y)$ is defined as follows:

$$\begin{aligned} D_c(x,y) \quad := \quad & \text{the minimum weight to cover the elements in} \\ & x \cup (A(y) \setminus c\,) \text{ by using the subsets in the} \\ & \text{subtree } T[y] \text{ for } y \in \{c, c', c''\} \text{ and } x \subseteq c. \end{aligned}$$

During the bottom-up process, the algorithm fills out such a table for each node. For an internal node c, the entries of the columns corresponding to c' and c'' can be directly retrieved from $D_{c'}$ and $D_{c''}$, which have been already computed before we arrive at node c.[1] Using the values from the first two columns, we can then compute the entries in the column of c. After D_r for the root r of the subset tree is computed, we can find the minimum weight to cover all elements in S in the entry $D_r(r, r)$. In the following, we describe the subtle details how to fill out the table for a node in the tree. We distinguish three cases:

Case 1: Node $c := \{s_1, s_2, \ldots, s_{k'}\}$ is a leaf:

[1] Note that these two columns are only needed to make the description of the computation of the last column more simple. For the purpose of implementation, the table D_c needs only the column corresponding to c.

D_c	c'	c''	c
$\emptyset$			
$\{s_1\}$			
$\{s_2\}$			
$\vdots$			
$c := \{s_1, s_2, \ldots, s_{k'}\}$			

Figure 14.1: Table D_c for node $c := \{s_1, s_2, \ldots, s_{k'}\}$ with $k' \leq k$ having two children c' and c''.

Since c has no child, columns c' and c'' are empty. We can easily compute the third column:

$$D_c(x, c) := \begin{cases} 0, \text{ if } x = \emptyset; \\ w(c), \text{ otherwise.} \end{cases}$$

Case 2: Node $c := \{s_1, s_2, \ldots, s_{k'}\}$ has only one child c':

The column c'' of D_c is empty. The first step to fill out the table is to get the values of the first column from the table $D_{c'}$. If there is one element s_j, $1 \leq j \leq k'$, in set x which does not occur in $T[c']$, i.e., $x \nsubseteq A(c')$, then it is impossible to cover $x \cup (A(c') \setminus c)$ by using only the subsets in $T[c']$. The entry $D_c(x, c')$ is then set to ∞. Otherwise, i.e., $x \subseteq A(c')$, in order to get the value of $D_c(x, c')$, we have to find the (uniquely determined) row in table $D_{c'}$ which corresponds to the subset x' of c' satisfying $x' \cup (A(c') \setminus c') = x \cup (A(c') \setminus c)$. Due to the consistency property of tree-like subset collections, each element in c also occurring in $T[c']$ is an element of c'. Hence we get

$$x \cup (A(c') \setminus c) = x \cup (c' \setminus c) \cup (A(c') \setminus c').$$

We set $x' := x \cup (c' \setminus c)$. Since $x \subseteq A(c')$ and $x \subseteq c$, it follows that $x \subseteq c'$. Therefore, also $x' \subseteq c'$ and there is a row in $D_{c'}$ corresponding to x'. Thus, $D_c(x, c')$ is set equal to $D_{c'}(x', c')$. Altogether, we have:

$$D_c(x, c') := \begin{cases} \infty, \text{ if } x \nsubseteq c'; \\ D_{c'}(x \cup (c' \setminus c), c'), \text{ if } x \subseteq c'. \end{cases}$$

The second step is to compute the last column of D_c using the values from the column for c'. For each row corresponding to a subset x of c, we have to compare the two possibilities to cover the elements of $x \cup (A(c) \setminus c)$, either using c to cover elements in x and using some subsets in $T[c']$ to cover the remaining elements or using solely subsets in $T[c']$ to cover all elements:

$$D_c(x, c) := \min\{w(c) + D_c(\emptyset, c'),\ D_c(x, c')\}.$$

Case 3: Node $c := \{s_1, s_2, \ldots, s_{k'}\}$ has two children c' and c'':

In this case, the first step can be done in the same way as in Case 2, i.e., retrieving the values of the columns c' and c'' of D_c from tables $D_{c'}$ and $D_{c''}$.

In order to compute the value of $D_c(x, c)$, for a row x corresponding to a subset of c, we also compare the two possibilities to cover $x \cup (A(c) \setminus c)$, either using c to cover x or not. In this case, however, we have two subtrees $T[c']$ and $T[c'']$ and, hence, we have more than one alternative to cover $x \cup (A(c) \setminus c)$ by only using subsets in $T[c']$ and $T[c'']$. As a simple example consider a subset $x \subseteq c$ that has only two elements, i.e., $x = \{s'_1, s'_2\}$. We can cover it by using only subsets in $T[c']$, only subsets in $T[c'']$, or a subset in $T[c']$ to cover $\{s'_1\}$ and a subset in $T[c'']$ to cover $\{s'_2\}$ or vice versa. Therefore, for $x := \{s'_1, s'_2, \dots, s'_{k''}\} \subseteq c$ with $k'' \leq k'$,

$$D_c(x, c) := \min \left\{ \begin{array}{l} w(c) + D_c(\emptyset, c') + D_c(\emptyset, c''), \\ D_c(\emptyset, c') + D_c(x, c''), \\ D_c(\{s'_1\}, c') + D_c(x \setminus \{s'_1\}, c''), \\ D_c(\{s'_2\}, c') + D_c(x \setminus \{s'_2\}, c''), \\ \vdots \\ D_c(x \setminus \{s'_2\}, c') + D_c(\{s'_2\}, c''), \\ D_c(x \setminus \{s'_1\}, c') + D_c(\{s'_1\}, c''), \\ D_c(x, c') + D_c(\emptyset, c'') \end{array} \right\}.$$

With these three cases, we can fill out D_c for all nodes c. The entry $D_r(r, r)$ stores the minimum weight to cover all elements where r denotes the root of the subset tree. In order to construct the minimum weight set cover, we can, using table D_r, find out whether the computed minimum weight is achieved by taking r into the minimum weight set cover or not. Then, doing a traceback, we can recursively, from the root to the leaves, determine the subsets in the minimum weight set cover. Note that, if we only want to know the minimum weight, we can discard the tables $D_{c'}$ and $D_{c''}$ after filling out D_c, for each internal node c with children c' and c'', to reduce the required memory space from $O(2^k \cdot m)$ to $O(2^k)$.

Theorem 14.1. Tree-like Weighted Set Cover *with an underlying binary subset tree can be solved in* $O(3^k \cdot m \cdot n)$ *time, where* k *denotes the maximum subset size, i.e.,* $k := \max_{c \in C} |c|$.

Proof. The correctness of the above algorithm directly follows from the above description.

Concerning the runtime of the algorithm, the size of table D_c is bounded from above by $3 \cdot 2^k$ for each node c since $|c| \leq k$. Using a proper data structure, such as a hash table, the retrieval of a value from one of the tables corresponding to the children can be done in constant time. Thus, the two columns of D_c corresponding to the two children c' and c'' can be filled out in $O(2^k)$ time. To compute an entry in the column c, which corresponds to a subset x of c, the algorithm compares all possibilities to cover some elements of x by the subsets in $T[c']$. There can be only $2^{|x|}$ such possibilities. Hence, it needs $O(2^{|x|})$ steps to compute $D_c(x, c)$ for each subset x of c. Since all set operations needed between two sets with maximum size of n can be done in $O(n)$ time, the runtime for

computing D_c is

$$n \cdot (\sum_{j=1}^{|c|} \binom{|c|}{j} O(2^j)) + O(2^{|c|}) = O(3^{|c|} \cdot n).$$

Therefore, the computation of the tables of all nodes can be done in $O(3^k \cdot m \cdot n)$ time. During the traceback, we visit, from the root to leaves, each node only once and, at each node, can in constant time find out whether or not to put this node into the set cover and with which entries in the tables of the children to continue the traceback. Thus, the traceback works in $O(m)$ time. □

14.2.2 TWSC with Arbitrary Subset Tree

Our subsequent main result gives a fixed-parameter algorithm that solves TWSC on *arbitrary* subset trees.

Theorem 14.2. TREE-LIKE WEIGHTED SET COVER *can be solved in* $O(3^k \cdot m \cdot n)$ *time, where* k *denotes the maximum subset size, i.e.,* $k := \max_{c \in C} |c|$.

Proof. Using the same construction as illustrated in Figure 13.2, we transform an arbitrary tree into a binary tree. For an internal node c with $l > 2$ children nodes, c^1, c^2, ..., c^l, we add $l - 2$ new nodes c^{12}, c^{123}, ..., and $c^{1 \cdots l-1}$ into the subset tree. All newly added nodes are set equal to c and have weight $w(c) + 1$. Observe that the newly added nodes $c^{12}, c^{123}, \ldots, c^{1 \cdots l-1}$ can never be in an optimal set cover, since they cover the same elements as c but have higher weight. Hence, there is a one-to-one correspondence between the solution for the arbitrary subset tree and the solution for the binary subset tree. Then, we can apply the algorithm in Section 14.2.1 to the binary tree which contains at most $2n$ nodes. □

Together with Theorem 14.3, we get our main result of this chapter.

Theorem 14.3. WEIGHTED MULTICUT IN TREES *can be solved in* $O(3^k \cdot h \cdot n)$ *time, where* k *denotes the vertex cutwidth, i.e., the maximum number of paths passing through a vertex or an edge,* h *denotes the number of vertex pairs, and* n *denotes the number of tree vertices.*

14.3 Concluding Remarks

Garg et al. [82] have shown that MULTICUT IN TREES is equivalent to the so-called TREE-REPRESENTABLE SET COVER problem. A (weighted) SET COVER instance (S, C) is called a *tree-representable set system* if there is a tree T in which each edge is associated with a subset in C such that, for each element $s \in S$, the edges corresponding to the subsets containing s induce a *path* in T. There are polymomial-time algorithms to decide whether a given SET COVER instance is tree-representable [23]. The problem of deciding whether a given SET COVER

instance is tree-representable has been extensively studied in different contexts such as testing whether a given binary matroid is graphic [170].

Compare Tree-representable Set Cover with Tree-like Set Cover. Both problems have an underlying tree and, in both problems, the subsets containing an element should induce a connected substructure. In the tree-representable case, however, these subsets induce only a path whereas in the tree-like case they induce a tree. Furthermore, the subsets in the subset collection are associated with the edges of the tree in the tree-representable case and the subsets are associated with the nodes of the tree in the tree-like case. Concerning their complexity, for the unweighted case, Tree-representable Set Cover is NP-complete, since Multicut in Trees is NP-complete [82], while Tree-like Set Cover can be easily solved in polynomial time as shown in Proposition 14.1. By way of contrast, due to the equivalence between Multicut in Trees and Tree-representable Set Cover, Tree-representable Weighted Set Cover can be reduced to TWSC. The reverse reduction, if valid, seems to be hard to show. In this sense, we have the "paradoxical situation" that, whereas the *unweighted* case of TWSC is much easier than Tree-representable Set Cover, the *weighted* case of TWSC seems harder than Tree-representable Weighted Set Cover. This observation might be an interesting research subject for future research.

The application of TWSC, namely the reduction of the space requirement of dynamic programming (for problems such as Vertex Cover, Dominating Set etc.) on (nice) tree decompositions of graphs, where parameter k (which corresponds to treewidth there) is typically between 5 and 20, underpins the practical relevance of the parameterization by subset size. Using TWSC, memory savings of up to 90 and more have been achieved in this way [22]. An interesting potential for further applications appears in recent work of Mecke and Wagner [133] in the context of railway optimization problems. They studied a special case of TWSC where the subset collection is almost "path-like." Other applications of TWSC are conceivable with respect to acyclic hypergraphs and their applications for relational databases [168], and several other fields with set covering applications to be explored in future research. Finally, it is an interesting task for future research to investigate the relation between tree-like set covering and set covering with *almost* consecutive ones property as introduced by Ruf and Schöbel [160].

Part VI

Conclusion

Chapter 15

Conclusion

There is a long list of graph modification problems arising in various fields of applications. A very natural and promising approach to attack these, in most cases NP-complete, problems is to design fixed-parameter algorithms. In this thesis we have investigated four general techniques for designing fixed-parameter algorithms, namely, iterative compression, data reduction, depth-bounded search trees, and parameterization by structure. We gave a brief description of the general scheme behind each technique and demonstrated with several case studies how to adapt the general scheme to individual problems.

Summary. In contrast to Section 1.4, we organize the following summary of results according to the considered problems. We use n and m to denote the number of the graph vertices and the graph edges, respectively.

- Feedback Vertex Set. We gave a fixed-parameter algorithm running in $O(c^k \cdot m)$ time where k denotes the size of the feedback vertex set and c is a constant (Chapter 3 and Chapter 4). Moreover, we modified this algorithm to obtain an enumeration algorithm for Feedback Vertex Set with the same runtime (Chapter 5).
- Edge Bipartization. An $O(2^k \cdot m^2)$ time algorithm was described in Chapter 4 where k is the size of the edge bipartization set.
- Cluster Editing. We provided two data reduction rules and showed a problem kernel of size $O(k^2)$ with k denoting the size of the cluster editing set (Chapter 7). A search tree algorithm running in $O(2.27^k + n^3)$ time was presented in Chapter 10.
- Closest 3-Leaf Power. A forbidden subgraph characterization and a search tree algorithm based on this characterization have been derived in Chapter 11.
- Multicut in Trees. An exponential-size problem kernel was shown by means of eight data reduction rules (Chapter 6). In addition, we gave a

simple $O(2^k \cdot n^2)$ time algorithm by applying the search tree technique (Chapter 9). It turned out that parameterization by a structural parameter, the vertex cutwidth, provides an efficient algorithm for the weighted case (Chapter 13).

- MULTICOMMODITY DEMAND FLOW IN TREES. We presented a fixed-parameter algorithm with respect to the maximum number k of flows passing through a vertex in the tree network. The runtime is $O(2^k \cdot n^2)$.

Future research. The obvious next step is to try to improve the results achieved in this work. The exponential-size problem kernel for MULTICUT IN TREES, the fixed-parameter algorithm for FEEDBACK VERTEX SET with a relatively high exponential term due to the big constant, and the search tree algorithms for CLOSEST 3-LEAF POWER seem to be the prime candidates for future improvements. Implementation of the algorithms and experiments with real-world data would be necessary to make a fair judgement of the performance of the algorithms in practice. Additional heuristical tuning methods should be taken into account when the algorithms are applied in some applications. For instance, several encouraging results of algorithm engineering in the realm of fixed-parameter algorithms have been presented for VERTEX COVER [2, 3, 8, 38].

Further specializing and deepening the techniques discussed in this thesis seems to be a challenging task. While data reduction and search trees are already considered to be "well-established" methods, iterative compression and parameterization by structure are relatively new and have not been "sufficiently" studied. For example, a constant-factor approximation algorithm has been used to replace the iteration procedure as a speed-up method for iterative compression (Chapter 4). A speed-up method for the compression procedure could be to use approximation algorithms to "steer" the compression procedure. For instance, instead of the brute-force partition, a smarter and more efficient partition of size-$(k+1)$ solutions could be achieved. Attacking (in particular, W[1]-hard with respect to some natural parameters) problems using new parameters might be an interesting subject for future research. Estivill-Castro et al. [70] have started some work in this direction. They show, among other things, that the W[2]-complete DOMINATING SET problem can be solved in $O(103^k \cdot n^{O(1)})$ time with the maximum number of leaves of the spanning trees of the input graph as parameter k.

We have shown in this thesis several problem kernels, in particular, the exponential-size problem kernels for MULTICUT IN TREES and MINIMUM CLIQUE COVER. On the one hand, we felt that it might not be easy to derive polynomial- or even linear-size kernels for these two problems. On the other hand, we are not aware of any tools to show that the exponential-size problem kernels are the best possible for MULTICUT IN TREES and MINIMUM CLIQUE COVER with respect to polynomial-time data reductions. It is a fundamental challenge to derive a framework for proving (relative) lower bounds for problem kernel sizes (see [40] for a fist result on linear bounds on problem kernel size). Little is known here.

Bibliography

[1] K. A. Abrahamson, R. G. Downey, and M. R. Fellows. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogs. *Annals of Pure and Applied Logic*, 73:235–276, 1995. 6

[2] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the Vertex Cover problem: theory and experiments. In *Proc. of 6th ACM-SIAM ALENEX*, pages 62–69. ACM-SIAM, 2004. 44, 142

[3] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons. Scalable parallel algorithms for FPT problems. To appear in *Algorithmica*, 2005. 142

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Springer, 1994. 3, 120

[5] J. Alber. *Exact Algorithms for NP-hard Problems on Networks: Design, Analysis, and Implementation.* PhD thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2003. 40

[6] J. Alber, N. Betzler, and R. Niedermeier. Experiments on data reduction for optimal domination in networks. In *Proc. of 1st INOC*, pages 1–6, 2003. Long version to appear in *Annals of Operations Research.* 40, 54, 55

[7] J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier. Fixed parameter algorithms for Dominating Set and related problems on planar graphs. *Algorithmica*, 33(4):461–493, 2002. 116

[8] J. Alber, F. Dorn, and R. Niedermeier. Experimental evaluation of a tree decomposition based algorithm for Vertex Cover on planar graphs. *Discrete Applied Mathematics*, 145(2):219–231, 2005. 142

[9] J. Alber, H. Fan, M. R. Fellows, H. Fernau, R. Niedermeier, F. Rosamond, and U. Stege. A refined search tree technique for Dominating Set on planar graphs. *Journal of Computer and System Sciences*, 71(4):385–405, 2005. 116

[10] J. Alber, M. R. Fellows, and R. Niedermeier. Polynomial-time data reduction for Dominating Set. *Journal of the ACM*, 51:363–384, 2004. 40, 51, 54, 55

[11] J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Computing the similarity of two sequences with nested arc annotations. *Theoretical Computer Science*, 312:337–358, 2004. 9, 75

[12] R. S. Anand, T. Erlebach, A. Hall, and S. Stefanakos. Call control with k rejections. *Journal of Computer and System Sciences*, 67:707–722, 2003. 120

[13] T. Asano and T. Hirata. Edge-deletion and edge-contraction problems. In *Proc. of 14th ACM STOC*, pages 245–254, 1982. 4

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

[15] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics*, 3(2):289–297, 1999. 21, 32

[16] M. O. Ball, T. L. Magnanti, C. L. Monma, and G. L. Nemhauser, editors. *Handbooks in Operations Research and Management Science (Volume 7): Network Models*. Elsevier, 1995. 3

[17] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Machine Learning*, 56(1):89–113, 2004. 4, 47, 48

[18] R. Bar-Yehuda, D. Geiger, J. Naor, and R. M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and Bayesian inference. *SIAM Journal on Computing*, 27(4):942–959, 1998. 4, 22, 32

[19] A. Becker, R. Bar-Yehuda, and D. Geiger. Randomized algorithms for the Loop Cutset problem. *Journal of Artificial Intelligence Research*, 12:219–234, 2000. 4, 21, 42

[20] A. Becker, D. Geiger, and A. A. Schäffer. Automatic selection of loop breakers for genetic linkage analysis. *Human Heredity*, 48:49–60, 1998. 4

[21] R. Bellman. Dynamic programming treatment of the Traveling Salesman Problem. *Journal of the ACM*, 9(1):61–63, 1962. 111

[22] N. Betzler, R. Niedermeier, and J. Uhlmann. Tree decompositions of graphs: saving memory in dynamic programming. In *Proc. of 2nd CTW*, pages 56–80, 2004. Long version to appear in *Discrete Optimization.* 8, 10, 128, 129, 130, 137

[23] R. E. Bixby and D. K. Wagner. An almost linear time algorithm for graph realization. *Mathematics of Operations Research*, 13:99–123, 1988. 136

[24] M. Bläser. Computing small partial coverings. *Information Processing Letters*, 85(6):327–331, 2003. 15

[25] H. L. Bodlaender. Classes of graphs with bounded treewidth. Technical Report RUU-CS-86-22, Department of Computer Science, Utrecht University, The Netherlands, 1986. 115

[26] H. L. Bodlaender. On disjoint cycles. *International Journal of Foundations of Computer Science*, 5:59–68, 1994. 21

[27] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proc. of 22nd MFCS*, volume 1295 of *LNCS*, pages 19–36. Springer, 1997. 111, 129

[28] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998. 111, 129

[29] J. M. Bower and H. Bolouri, editors. *Computational Modeling of Genetic and Biochemical Networks (Computational Biology)*. The MIT Press, 2001. 3

[30] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: a Survey*. SIAM Monographs on Discrete Mathematics and Applications, 1999. 8, 76, 95

[31] N. H. Bshouty and L. Burroughs. Massaging a linear programming solution to give a 2-approximation for a generalization of the Vertex Cover problem. In *Proc. of 15th STACS*, volume 1373 of *LNCS*, pages 298–308. Springer, 1998. 15

[32] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996. 10, 48, 79, 100

[33] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84:119–138, 1997. 43

[34] P. J. Carrington, J. Scott, and S. Wasserman. *Models and Methods in Social Network Analysis*. Cambridge University Press, 2005. 3

[35] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar. Approximation algorithms for the unsplittable flow problem. In *Proc. of 5th APPROX*, volume 2462 of *LNCS*, pages 51–66. Springer, 2002. 120

[36] L. S. Chandran and F. Grandoni. Refined memorization for vertex cover. *Information Processing Letters*, 93(3):123–131, 2005. 15, 19, 81, 110

[37] M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. *Journal of Computer and System Sciences*, 71(3):360–383, 2005. 48

[38] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon. Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003. 142

[39] C. Chekuri, M. Mydlarz, and F. B. Shepherd. Multicommodity demand flow in a tree (extended abstract). In *Proc. of 30th ICALP*, volume 2719 of *LNCS*, pages 410–425. Springer, 2003. 120, 121

[40] J. Chen, H. Fernau, I. A. Kanj, and G. Xia. Parametric duality and kernelization: Lower bounds and upper bounds on kernel size. In *Proc. of 22nd STACS*, volume 3404 of *LNCS*, pages 269–280. Springer, 2005. 142

[41] J. Chen, I. A. Kanj, and W. Jia. Vertex Cover: Further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001. 44, 51

[42] Z.-Z. Chen, T. Jiang, and G. Lin. Computing phylogenetic roots with bounded degrees and errors. *SIAM Journal on Computing*, 32(4):864–879, 2003. 95, 106

[43] Z.-Z. Chen and T. Tsukiji. Computing bounded-degree phylogenetic roots of disconnected graphs. *Journal of Algorithms*, 59(2):125–148, 2004. 95

[44] J. Chuzhoy and J. S. Naor. Covering problems with hard capacities. In *Proc. 43rd IEEE FOCS*, pages 481–489, 2002. Long version to appear in *SIAM Journal on Computing*. 15

[45] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. 31, 104, 110, 112

[46] D. G. Corneil, Y. Perl, and L. Stewart. Cographs: recognition, application and algorithms. *Congressus Numerantium*, 43:249–258, 1984. 76

[47] M. Costa, L. Létocart, and F. Roupin. Minimal multicut and maximal integer multiflow: A survey. *European Journal of Operational Research*, 162:55–69, 2004. 54

[48] G. Călinescu, C. G. Fernandes, and B. A. Reed. Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width. *Journal of Algorithms*, 48:333–359, 2003. 54

[49] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994. 116

[50] P. Damaschke. On the fixed-parameter enumerability of Cluster Editing. In *Proc. of 31st WG*, volume 3787 of *LNCS*, pages 283–294. Springer, 2005. 7, 9, 33, 48

[51] P. Damaschke. Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theoretical Computer Science*, 351(3):337–350, 2006. 33, 36

[52] F. Dehne, M. R. Fellows, M. A. Langston, F. A. Rosamond, and K. Stevens. An $\mathcal{O}^*(2^{O(k)})$ FPT algorithm for the undirected feedback vertex set problem. In *Proc. of 11th COCOON*, volume 3595 of *LNCS*, pages 859–869. Springer, 2005. 6, 9, 20, 21, 22

[53] E. D. Demaine, F. V. Fomin, M. T. Hajiaghayi, and D. M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs. *Journal of the ACM*, 52(6):866–893, 2005. 116

[54] E. D. Demaine and N. Immorlica. Correlation clustering with partial information. In *Proc. of 6th APPROX*, volume 2764 of *LNCS*, pages 1–13. Springer, 2003. 48

[55] R. Diestel. *Graph Theory*. Springer, 3rd edition, 2005. 8

[56] I. Dinur and S. Safra. The importance of being biased. In *Proc. of 34th ACM STOC*, pages 33–42, 2002. 15

[57] M. Dom. *Error Compensation in Leaf Root Problems* (in German). Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2004. 96, 106

[58] M. Dom, J. Guo, F. Hüffner, and R. Niedermeier. Extending the tractability border for closest leaf powers. In *Proc. of 31st WG*, volume 3787 of *LNCS*, pages 397–408. Springer, 2005. 9, 76, 96, 106

[59] M. Dom, J. Guo, F. Hüffner, and R. Niedermeier. Error compensation in leaf power problems. *Algorithmica*, 44(4):363–381, 2006. 9, 10, 76, 93, 96, 98, 102, 103, 104, 105, 106

[60] M. Dom, J. Guo, F. Hüffner, R. Niedermeier, and A. Truß. Fixed-parameter tractability results for feedback set problems in tournaments. In *Proc. of 6th CIAC*, volume 3998 of *LNCS*, pages 321–332. Springer, 2006. 9

[61] M. Dom, J. Guo, and R. Niedermeier. Bounded degree Closest k-Tree Power is NP-complete. In *Proc. of 11th COCOON*, volume 3595 of *LNCS*, pages 757–766. Springer, 2005. 9, 76, 95

[62] M. Dom, J. Guo, R. Niedermeier, and S. Wernicke. Minimum membership set covering and the consecutive ones property. In *Proc. of 10th SWAT*, LNCS. Springer, 2006. 9

[63] R. G. Downey. Parameterized complexity for the skeptic. In *Proc. of 18th IEEE CCC*, pages 147–169, 2003. 6

[64] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness. *Congressus Numerantium*, 87:161–187, 1992. 21

[65] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 5, 6, 22, 76, 112, 130, 131

[66] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972. 111

[67] E. S. El-Mallah and C. J. Colbourn. The complexity of some edge deletion problems. *IEEE Transactions on Circuits and Systems*, 35(3):354–362, 1988. 4

[68] J. Ellis, H. Fan, and M. R. Fellows. The Dominating Set problem is fixed parameter tractable for graphs of bounded genus. *Journal of Algorithms*, 52(2):152–168, 2004. 116

[69] D. Emanuel and A. Fiat. Correlation clustering – minimizing disagreements on arbitrary weighted graphs. In *Proc. of 11th ESA*, volume 2832 of *LNCS*, pages 208–220. Springer, 2003. 48

[70] V. Estivill-Castro, M. Fellows, M. Langston, and F. Rosamond. FPT is P-time extremal structure I. In *Proc. 1st ACiD*, pages 1–41, 2005. 142

[71] G. Even, J. Naor, B. Schieber, and L. Zosin. Approximating minimum subset feedback sets in undirected graphs with applications. *SIAM Journal on Computing*, 13(2):255–267, 2000. 4

[72] S. Fedin and A. S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. In *Proc. of 1st IWPEC*, volume 3162 of *LNCS*, pages 248–259. Springer, 2004. 83

[73] U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45:634–652, 1998. 114, 130, 131

[74] M. R. Fellows. Blow-ups, win/win's, and crown rules: Some new directions in FPT. In *Proc. of 29th WG*, volume 2880 of *LNCS*, pages 1–12. Springer, 2003. 39, 110

[75] M. R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized. In *Proc. of 8th WADS*, volume 2748 of *LNCS*, pages 505–520. Springer, 2003. 6

[76] H. Fernau. On parameterized enumeration. In *Proc. of 8th COCOON*, volume 2383 of *LNCS*, pages 564–573. Springer, 2002. 33, 36

[77] P. Festa, P. M. Pardalos, and M. G. C. Resende. Feedback set problems. In D. Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization, Vol. A*, pages 209–258. Kluwer, 1999. 22

[78] S. Fiorini, N. Hardy, B. A. Reed, and A. Vetta. Planar graph bipartization in linear time. In *Proc. of 2nd GRACO*, volume 19 of *Electronic Notes in Discrete Mathematics*, pages 226–232, 2005. 31

[79] R. Gandhi, E. Halperin, S. Khuller, G. Kortsarz, and A. Srinivasan. An improved approximation algorithm for Vertex Cover with hard capacities. In *Proc. of 30th ICALP*, volume 2719 of *LNCS*, pages 164–175. Springer, 2003. 15

[80] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 4, 75, 128

[81] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996. 28

[82] N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18:3–30, 1997. 53, 54, 58, 62, 120, 125, 127, 136, 137

[83] J. Gramm. *Fixed-Parameter Algorithms for the Consensus Analysis of Genomic Sequences*. PhD thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2003. 75

[84] J. Gramm. A polynomial-time algorithm for the matching of crossing contact-map patterns. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):171–180, 2004. 111

[85] 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. 9, 83, 85, 92, 113

[86] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Exact algorithms for clique generation. *Theory of Computing Systems*, 38(4):373–392, 2005. 10, 47, 49, 50, 85, 113

[87] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction, exact, and heuristic algorithms for Clique Cover. In *Proc. of 8th ACM-SIAM ALENEX*, pages 86–94. ACM-SIAM, 2006. 9, 39, 43

[88] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier, H.-P. Piepho, and R. Schmid. A better algorithm for compact letters displays—a rendezvous between theoretical computer science and applied statistics. Manuscript, Institut für Informatik, FSU Jena, Germany, April 2006. 9, 42

[89] J. Gramm, J. Guo, and R. Niedermeier. On exact and approximation algorithms for distinguishing substring selection. In *Proc. of 14th FCT*, volume 2751 of *LNCS*, pages 195–209. Springer, 2003. Long version to appear under the title "Parameterized intractability of distinguishing substring selection" in *Theory of Computing Systems*. 9, 75

[90] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. *ACM Transactions on Algorithms*, 2(1):44–65, 2006. 9, 111

[91] J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for Closest String and related problems. *Algorithmica*, 37(1):25–42, 2003. 75

[92] S. Guha, R. Hassin, S. Khuller, and E. Or. Capacitated vertex covering. *Journal of Algorithms*, 48(1):257–270, 2003. 15

[93] J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Improved fixed-parameter algorithms for two feedback set problems. In *Proc. of 9th WADS*, volume 3608 of *LNCS*, pages 158–168. Springer, 2005. Long version to appear under the title "Compression-based fixed-parameter algorithms for Feedback Vertex Set and Edge Bipartization" in *Journal of Computer and System Sciences*. 9, 21

[94] J. Guo, F. Hüffner, E. Kenar, R. Niedermeier, and J. Uhlmann. Complexity and exact algorithms for Multicut. In *Proc. of 32nd SOFSEM*, volume 3831 of *LNCS*, pages 303–312. Springer, 2006. 9, 61, 116

[95] J. Guo, F. Hüffner, and R. Niedermeier. A structural view on parameterizing problems: distance from triviality. In *Proc. of 1st IWPEC*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004. 10, 111, 132

[96] J. Guo and R. Niedermeier. Exact algorithms and applications for Tree-Like Weighted Set Cover. To appear in *Journal of Discrete Algorithms*, 2005. 10, 128, 132

[97] J. Guo and R. Niedermeier. Fixed-parameter tractability and data reduction for Multicut in Trees. *Networks*, 46(3):124–135, 2005. 10, 53, 83

[98] J. Guo and R. Niedermeier. A fixed-parameter tractability result for Multicommodity Demand Flow in Trees. *Information Processing Letters*, 97(3):109–114, 2006. 10, 119, 121

[99] J. Guo, R. Niedermeier, and D. Raible. Improved algorithms and complexity for power domination in graphs. In *Proc. of 15th FCT*, volume 3623 of *LNCS*, pages 172–184. Springer, 2005. 9, 114, 115, 116

[100] J. Guo, R. Niedermeier, and S. Wernicke. Parameterized complexity of generalized Vertex Cover problems. In *Proc. of 9th WADS*, volume 3608 of *LNCS*, pages 36–48. Springer, 2005. Long version to appear under the title "Parameterized complexity of Vertex Cover variants" in *Theory of Computing Systems*. 9, 15

[101] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. 112

[102] B. V. Halldórsson, M. M. Halldórsson, and R. Ravi. On the appxoximability of the Minimum Test Collection problem. In *Proc. of 19th ESA*, volume 2161 of *LNCS*, pages 158–169. Springer, 2001. 45

[103] F. Harary. *Graph Theory*. Addison-Wesley, 1969. 3

[104] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999. 112

[105] T. W. Haynes, S. M. Hedetniemi, S. T. Hedetniemi, and M. A. Henning. Domination in graphs: applied to electric power networks. *SIAM Journal on Discrete Mathematics*, 15(4):519–529, 2002. 113, 114

[106] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10:196–210, 1962. 111

[107] F. Hüffner. *Graph Modification Problems and Automated Search Tree Generation*. Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2003. 83, 85, 92

[108] F. Hüffner. Algorithm engineering for optimal graph bipartization. In *Proc. of 4th WEA*, volume 3503 of *LNCS*, pages 240–252. Springer, 2005. 20, 25, 31

[109] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978. 103

[110] T. Jiang, G. Lin, and J. Xu. On the closest tree kth root problem. Manuscript, Department of Computer Science, University of Waterloo, 2000. 95

[111] D. Jungnickel. *Graphs, Networks and Algorithms*. Springer, 1999. 3, 8, 104, 110

[112] A. B. Kahng, S. Vaya, and A. Zelikovsky. New graph bipartizations for double-exposure, bright field alternating phase-shift mask layout. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 133–138, 2001. 28

[113] I. Kanj, M. Pelsmajer, and M. Schaefer. Parameterized algorithms for feedback vertex set. In *Proc. of 1st IWPEC*, volume 3162 of *LNCS*, pages 235–247. Springer, 2004. 22

[114] H. Kaplan, R. Shamir, and R. E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999. 80, 104

[115] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 15, 21, 112

[116] P. E. Kearney and D. G. Corneil. Tree powers. *Journal of Algorithms*, 29(1):111–131, 1998. 95

[117] E. Kenar and J. Uhlmann. Multicut in graphs. Study work, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2005. 61

[118] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995. 111

[119] T. Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994. 111, 129

[120] J. Kneis, D. Mölle, S. Richter, and P. Rossmanith. Parameterized power domination complexity. *Information Processing Letters*, 98(4):145–149, 2006. 113, 114

[121] D. E. Knuth. *The Art of Computer Programming, Volume 1 (Fundamental Algorithms)*. Addison-Wesley, 3rd edition, 1997. 112

[122] J. Könemann, G. Konjevod, O. Parekh, and A. Sinha. Improved approximations for tour and tree covers. *Algorithmica*, 38(3):441–449, 2004. 15

[123] L. T. Kou, L. J. Stockmeyer, and C. K. Wong. Covering edges by cliques with respect to keyword conflicts and intersection graphs. *Communications of the ACM*, 21:135–138, 1978. 42

[124] A. S. Kulikov. Automated generation of simplification rules for SAT and MAXSAT. In *Proc. of 8th SAT*, volume 3569 of *LNCS*, pages 430–436. Springer, 2005. 83

[125] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999. 78

[126] M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, 1986. 47, 96

[127] L. C. Lau. Bipartite roots of graphs. In *Proc. of 15th ACM-SIAM SODA*, pages 952–961. ACM-SIAM, 2004. 95

[128] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980. 4, 96

[129] G. Lin, P. E. Kearney, and T. Jiang. Phylogenetic k-root and Steiner k-root. In *Proc. of 11th ISAAC*, volume 1969 of *LNCS*, pages 539–551. Springer, 2000. 95, 96, 99, 106

[130] Y. L. Lin and S. S. Skiena. Algorithms for square roots of graphs. *SIAM Journal on Discrete Mathematics*, 8(1):99–118, 1995. 95

[131] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In *Proc. of 20th ICALP*, volume 700 of *LNCS*, pages 40–51. Springer, 1993. 21

[132] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41:960–981, 1994. 42

[133] S. Mecke and D. Wagner. Solving geometric covering problems by data reduction. In *Proc. of 12th ESA*, volume 3221 of *LNCS*, pages 760–771. Springer, 2004. 137

[134] B. Monien. The bandwidth minimization problem for caterpillars with hair length 3 is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 7:505–512, 1986. 53

[135] R. Motwani and M. Sudan. Computing roots of graphs is hard. *Discrete Applied Mathematics*, 54(1):81–88, 1994. 95

[136] A. Nagurney, editor. *Innovations in Financial and Economic Networks.* Edward Elgar Publishing, 2003. 3

[137] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001. 4, 76

[138] G. L. Nemhauser and L. E. Trotter. Vertex packing: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975. 44

[139] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1988. 130

[140] R. Niedermeier. Ubiquitous parameterization—invitation to fixed-parameter algorithms. In *Proc. of 29th MFCS*, volume 3153 of *LNCS*, pages 84–103. Springer, 2004. 6

[141] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006. 6, 33, 40

[142] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125–129, 2000. 39, 43, 44, 78, 81, 87

[143] S. Nikolenko and A. Sirotkin. Worst-case upper bounds for SAT: automated proof, 2003. *Presented at 15th European Summer School in Logic Language and Information (ESSLLI 2003), Student Session.* 83

[144] N. Nishimura, P. Ragde, and D. M. Thilikos. On graph powers for leaf-labeled trees. *Journal of Algorithms*, 42(1):69–108, 2002. 93, 95, 96

[145] J. Orlin. Contentment in graph theory: covering graphs with cliques. *Indigationes Mathematicae*, 80:406–424, 1977. 42

[146] J. Ott, editor. *Analysis of Human Genetic Linkage: Revised*. The John Hopkins University Press, 1991. 4

[147] A. E. Ozdaglar and D. P. Bertsekas. Routing and wavelength assignment in optical networks. *IEEE/ACM Transactions on Networking*, 11(2):259–272, 2003. 120

[148] R. D. M. Page and J. A. Cotton. Vertebrate phylogenomics: reconciled trees and gene duplications. In *Proc. of 7th Pacific Symposium on Biocomputing*, pages 536–547, 2002. 8, 10, 128, 130, 131

[149] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991. 8, 28, 128, 131

[150] M. Pop, D. S. Kosack, and S. L. Salzberg. Hierarchical scaffolding with Bambus. *Genome Research*, 14:149–159, 2004. 28

[151] D. Raible. *Algorithms and Complexity Results for Power Domination in Networks* (in German). Master's thesis, Wilhelm-Schickard Institut für Informatik, Universität Tübingen, Germany, 2005. 114, 115

[152] V. Raman, S. Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for undirected feedback vertex set. In *Proc. of 13th ISAAC*, volume 2518 of *LNCS*, pages 241–248. Springer, 2002. 22, 42, 106

[153] V. Raman, S. Saurabh, and C. R. Subramanian. Faster algorithms for feedback vertex set. In *Proc. of 2nd GRACO*, volume 19 of *Electronic Notes in Discrete Mathematics*, pages 273–279, 2005. 22

[154] B. A. Reed. Algorithmic aspects of tree width. In B. A. Reed and C. L. Sales, editors, *Recent Advances in Algorithms and Combinatorics*, pages 85–107. Springer, 2003. 111

[155] B. A. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32:299–301, 2004. 8, 9, 13, 18, 19, 20, 25, 28

[156] F. Roberts, editor. *Applications of Combinatorics and Graph Theory to the Biological and Social Sciences*. Springer, 1989. 3

[157] F. Roberts, editor. *Graph Theory and Its Applications to Problems of Society*. SIAM, 1993. 3

[158] N. Robertson and P. D. Seymour. Graph minors. II: Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986. 109, 111, 129

[159] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986. 113

[160] N. Ruf and A. Schöbel. Set covering with almost consecutive ones property. *Discrete Optimization*, 1(2):215–228, 2004. 137

[161] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, 1983. Reprinted in 1999 by CSLI Publications. 112

[162] B. Schwikowski and E. Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1–3):253–265, 2002. 34

[163] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144(1–2):173–182, 2004. 47, 48, 92

[164] S. C. Shapiro. *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, 2nd edition, 1992. 83

[165] R. Sharan and R. Shamir. Algorithmic approaches to clustering gene expression data. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 269–300. The MIT Press, 2002. 47

[166] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, 2004. 75, 116, 117

[167] S. Szeider. On fixed-parameter tractable parameterizations of SAT. In *Proc. of 6th SAT*, volume 2919 of *LNCS*, pages 188–202. Springer, 2004. 116

[168] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984. Addendum in *SIAM Journal on Computing*, 14(1): 254-255, 1985. 129, 137

[169] J. A. Telle and A. Proskurowski. Practical algorithms on partial k-trees with an application to domination-like problems. In *Proc. of 3rd WADS*, volume 709 of *LNCS*, pages 610–621. Springer, 1993. 111, 112

[170] W. T. Tutte. An algorithm for determining whether a given binary matroid is graphic. *Proc. Amer. Math. Soc.*, 11:905–917, 1960. 137

[171] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002. 112

[172] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003. 8

[173] A. F. Veinott and H. M. Wagner. Optimal capacity scheduling. *Operations Research*, 10:518–532, 1962. 130

[174] A. M. Verweij, K. Aardal, and G. Kant. On an integer multicommodity flow problem from the airplane industry. Technical Report UU-CS-1997-38, Utrecht University, The Netherlands, 1997. 120

[175] K. Weihe. Covering trains by stations or the power of data reduction. In *Proc. of 1st ALEX*, pages 1–8, 1998. http://rtm.science.unitn.it/alex98/proceedings.html. 40, 70

[176] K. Weihe. On the differences between "practical" and "applied". In *Proc. of 4th WEA*, volume 1982 of *LNCS*, pages 1–10. Springer, 2000. 40, 70

[177] S. Wernicke. *On the Algorithmic Tractability of Single Nucleotide Polymorphism (SNP) Analysis and Related Problems*. Master's thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany, 2003. 28

[178] S. Wernicke, J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Avoiding forbidden submatrices by row deletions. In *Proc. of 30th SOFSEM*, volume 2932 of *LNCS*, pages 349–360. Springer, 2004. Long version to appear under the title "The computational complexity of avoiding forbidden submatrices by row deletions" in *International Journal of Foundations of Computer Science*. 9

[179] D. B. West. *Introduction to Graph Theory*. Prentice-Hall, 2nd edition, 2000. 8, 104, 110

[180] M. Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981. 4

[181] M. Yannakakis. Node-deletion problems on bipartite graphs. *SIAM Journal on Computing*, 10(2):310–327, 1981. 4

[182] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989. 111

Tabellarischer Lebenslauf

Name:	Jiong Guo
Geburtsdatum :	28.11.1970
Geburtsort:	Chengdu, V. R. China
Staatsangehörigkeit:	Chinesisch
Anschrift:	Markt 14 07743 Jena
Familienstand:	verheiratet, 1 Kind

09.1977 - 07.1982	11. Grundschule von Chengdu.
09.1982 - 07.1985	21. Gymnasium von Chengdu.
09.1985 - 07.1988	7. Gymnasium von Chengdu.
09.1988 - 07.1992	Studium des Fachs Informatik an der University of Electronic Science and Technology of China.
30.07.1992	Abschluss des Studiums als Bachelor of Science.
08.1992 - 09.1995	Systemverwalter und Softwareentwickler an der Industrial and Commercial Bank of China in Chengdu.
10.1995 - 07.1996	Systemverwalter und Softwareentwickler an der Guotai Securities Ltd. in Chengdu.
08.1996 - 02.2002	Studium des Fachs Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität Tübingen.
20.02.2002	Abschluss des Studiums als Diplom-Informatiker.
03.2002 - 10.2004	wissenschaftlicher Angestellter am Institut für Informatik der Universität Tübingen.
10.2004 - 04.2006	wissenschaftlicher Angestellter am Institut für Informatik der Universität Jena.

Jena, 28. April 2006