Algorithms and Experiments for Parameterized Approaches to Hard Graph 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. Falk Hüffner
geboren am 25. 02. 1976 in Oldenburg (Oldenburg)

Gutachter

1. Prof. Dr. Rolf Niedermeier (Friedrich-Schiller-Universität Jena)
2. Prof. Dr. Michael R. Fellows (The University of Newcastle, Australien)
3. Prof. Dr. Peter Rossmanith (Rheinisch-Westfälische Technische Hochschule Aachen)

Tag der letzten Prüfung des Rigorosums: 17. Dezember 2007

Tag der öffentlichen Verteidigung: 19. Dezember 2007

Zusammenfassung

Diese Dissertation beschäftigt sich mit Entwurf, Analyse, Implementierung und experimenteller Bewertung von Algorithmen für schwere Graphprobleme. Das Ziel ist es zu belegen, dass das Konzept der *parametrisierten Komplexität*, und insbesondere neuartige algorithmische Techniken, deren Entwicklung auf dieses Konzept zurückgeht, zu einsetzbaren Programmen für die exakte Lösung von Praxisinstanzen führt.

Insbesondere untersuchen wir die drei Graphprobleme Clique Cover, Balanced Subgraph und Minimum-Weight Path. Beim Clique Cover-Problem ist die Aufgabe, eine kleinstmögliche Menge von Cliquen zu finden, die alle Kanten abdeckt. Es hat Anwendungen in Compileroptimierung, geometrischen Berechnungsproblemen und angewandter Statistik. Beim Balanced Subgraph-Problem sucht man nach einer Zweifärbung der Knotenmenge, die die Inkonsistenzen mit Kantenbeschriftungen minimiert. Es hat Anwendungen in sozialen Netzwerken, Systembiologie und integriertem Schaltungsentwurf. Das Minimum-Weight Path-Problem besteht darin, einen einfachen Pfad einer gegebenen Länge mit minimalem Gewicht zu finden. Es hat Anwendungen in Systembiologie und Textanalyse.

Alle drei Probleme sind, wie viele andere praktisch relevante Graphrobleme, NP-schwer. Deshalb braucht vermutlich jeder exakte Algorithmus eine Laufzeit, die exponentiell mit der Eingabegröße wächst. Die Idee der parametrisierten Komplexität ist eine zweidimensionale Sichtweise, d. h. zusätzlich zur Eingabegröße n betrachten wir einen Parameter k; ein typischer Parameter ist die Größe der Lösung. Ein Problem ist *festparameter-handhabbar*, wenn eine Instanz der Größe n in Zeit $f(k) \cdot n^{O(1)}$ gelöst werden kann, wobei f eine beliebige berechenbare Funktion ist, die den exponentiellen Anteil der Laufzeit absorbiert. Somit können wir gute Laufzeiten erwarten, wenn der Parameter in der Anwendung klein ist.

Diverse Techniken zum Entwurf von Festparameter-Algorithmen sind vorgeschlagen worden. Wir untersuchen drei solche Techniken: Datenreduktion, iterative Kompression und Farbkodierung.

Datenreduktion. Datenreduktion ist eine klassische Methode, mit schweren Problemen umzugehen: bevor der eigentliche Lösungsprozess gestartet wird, versucht man, die Größe der Eingabe zu verringern, indem man Teile entfernt oder vereinfacht, zum Beispiel weil man sofort sehen kann, dass sie irrelevant für die Lösung sind. Wir geben effiziente und effektive Datenreduktionsregeln für Clique Cover an und zeigen so insbesondere, dass Clique Cover festparameterhandhabbar ist. Zusammen mit einem einfachen Suchbaumalgorithmus ist es so möglich, exakte Lösungen mit vergleichsweise kurzer Laufzeit zu bekommen.

Dies wird durch Experimente mit Anwendungsdaten und generierten Daten bestätigt.

Wir zeigen weiter eine neuartige Datenreduktion für Balanced Subgraph basierend auf kleinen Graphseparatoren und einem neuen Konstruktionsschema für verkleinerte äquivalente Graphkomponenten. Das Datenreduktionsschema kann für eine Reihe von Graphproblemen angewendet werden, bei denen eine Färbung oder eine Teilmenge der Knoten gesucht wird. Unsere Implementierung, die die Datenreduktion mit iterativer Kompression verbindet, kann biologische Praxisinstanzen exakt lösen, für die bisher nur Approximationen bekannt waren. Wir bestimmen die Grenzen der Anwendbarkeit durch Experimente mit generierten Daten.

Iterative Kompression. Iterative Kompression ist eine 2004 entwickelte Technik, die auf struktureller Induktion und Kompression von Zwischenlösungen basiert. Mithilfe von iterativer Kompression entwerfen wir die schnellsten zur Zeit bekannten Festparameter-Algorithmen für eine Reihe von Problemen. Das erste ist Cluster Vertex Deletion, das Problem, eine minimale Anzahl von Knoten aus einem Graphen zu löschen, sodass danach jede Zusammenhangskomponente eine Clique ist. Wir geben einen Algorithmus mit Laufzeit $O(2^k \cdot k^9 + n^3)$ an, wobei n die Anzahl Knoten im Eingabegraphen und k die Lösungsgröße ist. Dies ist die erste nichttriviale Anwendung von iterativer Kompression für ein Problem, das kein Kreiszerstörungsproblem ist, und eine der ersten Anwendungen, die auch mit gewichteten Daten umgehen können. Als nächstes betrachten wir das Feedback Vertex Set-Problem in Turniergraphen, also das Problem, einen Turniergraphen durch Knotenlöschungen azyklisch zu machen, und erhalten eine Laufzeit von $O(2^k \cdot n^2(\log n + k))$. Im Kern benutzt der Algorithmus auf überraschende Weise eine Prozedur zur Bestimmung einer längsten gemeinsamen Teilzeichenkette.

Wir zeigen weiter, dass das Edge Bipartization-Problem festparameterhandhabbar ist, indem wir einen auf iterativer Kompression beruhenden Algorithmus mit Laufzeit $O(2^k \cdot m^2)$ angeben, wobei m die Anzahl der Kanten im Eingabegraphen ist. Dieses Resultat kann auf Balanced Subgraph erweitert werden. Wir zeigen weiter einige heuristische Beschleunigungen, die sich in Experimenten als sehr effektiv herausstellen. Als nächstes geben wir einen neuartigen Beweis für einen auf iterativer Kompression beruhenden Algorithmus für Vertex Bipartization und verbessern die Laufzeit um einen Faktor von k auf $O(3^k \cdot mn)$. Mit einer heuristischen Beschleunigung kann unsere Implementierung alle Instanzen einer Problemsammlung aus der Bioinformatik in Minuten lösen, während konventionelle Methoden nur etwa die Hälfte der Instanzen überhaupt in akzeptabler Zeit lösen können. Weitere Experimente mit

generierten und zufälligen Eingaben werden gezeigt.

Farbkodierung. Farbkodierung ist eine randomisierte Methode zum Finden kleiner Teilgraphen der Größe k in einem Graph. Sie wurde bereits benutzt, um Kandidaten für Signalpfade in Proteininteraktionsnetzwerken zu finden, was als Minimum-Weight Path-Problem modelliert werden kann. Wir verbessern die Laufzeit der Farbkodierungsmethode für Minimum-Weight Path von $O(5.44^k \cdot m \cdot (-\ln \varepsilon))$ auf $O(4.32^k \cdot m \cdot (-\ln \varepsilon))$, wobei ε die Fehlerwahrscheinlichkeit ist. Wir zeigen weiter, wie heuristische Evaluationsfunktionen benutzt werden können, um den Suchraum einzuschränken. Unsere Implementierung, die sorgfältig abgestimmte Datenstrukturen benutzt, kann für praktisch alle für die Suche nach Signalpfaden relevanten Parametereinstellungen Resultate in Sekunden liefern, was die interaktive Untersuchung solcher Pfade ermöglicht. Für diese Aufgabe haben wir die graphische Benutzeroberfläche Faspad entwickelt.

Zusammengefasst haben wir eine Reihe neuer Festparameter-Algorithmen angegeben und Implementierungen für vier Probleme angegeben, die nach unserem Stand des Wissens die jeweils schnellsten Löser sind. Dies unterstreicht die praktische Brauchbarkeit des parametrisierten Ansatzes, und insbesondere seine Nützlichkeit als Leitfaden für die Entwicklung neuartiger algorithmischer Techniken wie iterativer Kompression oder Farbkodierung.

Die Implementierungen für Clique Cover, Balanced Subgraph, Vertex Bipartization und Minimum-Weight Path sind als freie Software unter der *GNU Public License* erhältlich bei http://theinf1.informatik.uni-jena.de/~hueffner/.

Abstract

This thesis is about the design, analysis, implementation, and experimental evaluation of algorithms for hard graph problems. The aim is to establish that the concept of *fixed-parameter tractability*, and in particular novel algorithmic techniques whose development was driven by this concept, can lead to practically useful programs for exactly solving real-world problem instances.

In particular, we examine the three graph problems Clique Cover, Balanced Subgraph, and Minimum-Weight Path. In the Clique Cover problem, the task is to find a minimum-cardinality set of cliques that covers all edges. It has applications in compiler optimization, computational geometry, and applied statistics. The Balanced Subgraph problem asks for a 2-coloring of a graph that minimizes the inconsistencies with given edge labels. It has applications in social networks, systems biology, and integrated circuit design. The Minimum-Weight Path problem is to find a minimum-weight simple path of a given length. It has applications in systems biology and text mining.

All three problems, as many other practically relevant graph problems, are NP-hard, implying that presumably any exact algorithm requires time exponential in the input size. The idea of parameterized complexity is to take a two-dimensional view, where in addition to the input size we consider a parameter k; a typical parameter is the size of the solution. A problem is called *fixed-parameter tractable* if an instance of size n can be solved in $f(k) \cdot n^{O(1)}$ time, where f is an arbitrary computable function that absorbs the exponential part of the running time. Thus, whenever the parameter turns out to be small in practice, we can expect good running times.

Various techniques to design fixed-parameter algorithms have been suggested. We examine three of them: data reduction, iterative compression, and color-coding.

Data reduction. Data reduction is a classic way of dealing with hard problems: before starting the actual solving process, one tries to reduce the size of the instance by removing or simplifying parts, for example because we can immediately infer that they are irrelevant to the solution. We present efficient and effective data reduction rules for Clique Cover, which in particular allow to prove that Clique Cover is fixed-parameter tractable. Combined with a simple search tree algorithm, this allows for exact problem solutions in competitive time. This is confirmed by experiments with real-world and synthetic data.

We then present a novel data reduction for Balanced Subgraph based on finding small separators and a novel gadget construction scheme. The data reduction scheme can be applied to a large number of graph problems where a coloring or a subset of the vertices is sought. Our implementation, which

combines the data reduction with iterative compression, can solve biological real-world instances exactly for which previously only approximations were known. We chart the borders of tractability by experiments with synthetic data.

Iterative compression. Iterative compression is a technique developed in 2004 that is based on structural induction and compression of intermediary solutions. Based on iterative compression, we present the fastest currently known fixed-parameter algorithms for a number of problems. The first is Cluster Vertex Deletion, the problem of deleting a minimum number of vertices from a graph to obtain a disjoint union of cliques. We give an algorithm running in $O(2^k \cdot k^9 + n^3)$ time, where n is the number of vertices in the input graph, and k is the solution size. This is the first nontrivial application of iterative compression to a problem that is not a feedback set problem and one of the first applications that can also deal with weighted instances. We continue with Feedback Vertex Set in tournaments, the problem of deleting a minimum number of vertices from a tournament graph to make it acyclic, for which we give an $O(2^k \cdot n^2(\log n + k))$ time algorithm. At its core, it makes surprising use of a longest common substring procedure.

We go on to show that the Edge Bipartization problem is fixed-parameter tractable by giving an iterative compression-based algorithm running in $O(2^k \cdot m^2)$ time, where m is the number of edges in the input graph. This result can be extended to Balanced Subgraph. We further present several heuristic speedups, which proved very effective in experiments. Next, we give a novel proof of an iterative compression algorithm for Vertex Bipartization and improve the running time by a factor of k to $O(3^k \cdot mn)$. With an additional heuristic speedup, our implementation can solve all instances from a testbed from computational biology within minutes, whereas established methods are only able to solve about half of the instances within reasonable time. Further experiments with synthetic and random inputs are shown.

Color-coding. Color-coding is a randomized method for finding small subgraphs of size k in a graph. It has been used to find candidates for signaling pathways in protein interaction networks, which can be modeled as a Minimum-Weight Path problem. We improve the worst-case running time of color-coding for Minimum-Weight Path from $O(5.44^k \cdot m \cdot (-\ln \varepsilon))$ to $O(4.32^k \cdot m \cdot (-\ln \varepsilon))$, where ε is the error probability. We further show how to use heuristic evaluation functions to prune the search space. Our implementation, using carefully tuned data structures, can for basically all parameter settings relevant to the search for signaling pathways obtain results within seconds, whereas previous tools used hours or longer. This allows the interactive exploration of such pathways. For

this, the graphical user interface FASPAD was implemented.

In summary, we have developed a number of new fixed-parameter algorithms and given implementations for four important problems, all of which are to the best of our knowledge the currently fastest solvers. This underlines the viability of the parameterized approach, and in particular its usefulness as a guide to develop novel algorithmic techniques such as iterative compression and color-coding.

The implementations for CLIQUE COVER, BALANCED SUBGRAPH, VERTEX BIPARTIZATION, and MINIMUM-WEIGHT PATH are available as free software under the GNU Public License at http://theinf1.informatik.uni-jena.de/~hueffner/.

Preface

This thesis covers a substantial part of my research on fixed-parameter algorithms, focusing on algorithms and experiments for hard graph problems. My research was part of the project "PIAF" funded by the Deutsche Forschungsgemeinschaft (DFG), which supported me starting in March 2004. I owe sincere thanks to Rolf Niedermeier for giving me the opportunity to work in his group and for his support in my research and the development of this thesis. Further, I want to thank my colleagues Nadja Betzler, Michael Dom, Jens Gramm, Jiong Guo, Christian Komusiewicz, Hannes Moser, and Sebastian Wernicke for productive cooperation and many interesting discussions. Finally, I am grateful to Tamara Steijger and Thomas Zichner for helping with implementation and experiments, and to Hans-Peter Piepho (Universität Hohenheim), Ramona Schmid (Universität Bielefeld), and Anke Truß (Universität Jena) for the interesting collaborations.

Much of the results in this thesis have been achieved in collaborations with some of the above mentioned people. In this thesis, I present only results that concern fixed-parameter algorithms for hard graph problems using three central techniques (data reduction, iterative compression, and color-coding) to which I have made substantial contributions. My further research concerns structural parameterization [Guo et al. 2004], leaf power problems [Dom et al. 2006a, 2005], multicut problems [Guo et al. 2007a], matrix problems [Brosemann et al. 2006], feedback set problems [Guo et al. 2006, 2007b], heuristics [Gramm et al. 2006, 2007b], and clustering [Komusiewicz et al. 2007, Guo et al. 2008, Hüffner et al. 2008a, Ponta et al. 2008]. I further collaborated on a number of book chapters and surveys [Dom et al. 2007, Hüffner 2007, Helwig et al. 2007, Hüffner et al. 2007c,b, 2008b].

This thesis is structured into six chapters. After a brief introduction in Chapter 1, I describe three central problems and their applications in Chapter 2. Then, one chapter is dedicated to each of the parameterized techniques data reduction (Chapter 3), iterative compression (Chapter 4), and color-coding (Chapter 5). Finally, I summarize the findings in Chapter 6. In the following, I briefly sketch my contributions.

Section 3.1 describes an exact algorithm for Clique Cover based on data reduction and a search tree algorithm. Jiong Guo found the equivalence of this problem to the Compact Letter Display problem. I came up with some of the data reduction rules for Clique Cover (Section 3.1.1); in particular Rule 3.4, which subsumes several weaker rules. The kernel (Theorem 3.2) was found by Jiong Guo. Further, I designed the search tree algorithm and did all of the implementation and experiments (Section 3.1.3). These results were presented at the *8th Workshop on Algorithm Engineering and Experiments (ALENEX '06)* [Gramm

et al. 2006] and are to appear in the *ACM Journal of Experimental Algorithmics* [Gramm et al. 2007a]. Further experimental results and comparisons to heuristics are to appear in *Computational Statistics & Data Analysis* [Gramm et al. 2007b].

Section 3.2 describes a data reduction scheme for the Balanced Subgraph problem. I initiated the study of this problem, designed the data reduction scheme, and proved its correctness and power. I further did the implementation. Experiments (Section 3.2.4) were designed by Nadja Betzler and me and carried out by Nadja Betzler with help from our students Tamara Steijger and Thomas Zichner. The results were presented at the *6th Workshop on Experimental Algorithms (WEA '07)* [Hüffner et al. 2007a].

Section 4.3 gives an exact algorithm for Cluster Vertex Deletion based on iterative compression. I started the examination of this problem, designed the algorithm, and proved its correctness (Hannes Moser found and fixed an error in the algorithm). These results are presented at the *8th Latin American Theoretical Informatics Symposium (LATIN '08)* [Hüffner et al. 2008a].

Section 4.4 applies iterative compression to the Feedback Vertex Set in tournaments problem. I had the central idea of using a polynomial-time string problem as a subroutine and proved the correctness of the algorithm. The results were presented at the *6th Conference on Algorithms and Complexity (CIAC '06)* [Dom et al. 2006b]; they can also be found in the diploma thesis of Anke Truß [2005].

Section 4.5 shows how to solve Edge Bipartization by iterative compression. The idea for the algorithm is due to Jiong Guo, and Jens Gramm found an initial, very complicated proof. I came up with the vastly simpler proof given here. These results were presented at the *9th Workshop on Algorithms and Data Structures (WADS '05)* [Guo et al. 2005] and appeared in the *Journal of Computer and System Sciences* [Guo et al. 2006]. The simpler proof further allowed me to find and prove the improvements presented in Sections 4.5.2 and 4.5.3. The applicability to Balanced Subgraph was also found by me. I further did the implementation of the algorithm. These results were sketched in the WEA '07 paper [Hüffner et al. 2007a], which focuses on the data reduction.

Section 4.6 describes an iterative compression algorithm for Vertex Bipartization. My first contribution is a simpler, more intuitive proof for this result originally due to Reed et al. [2004]. I further found the algorithmic improvements described in Section 4.6.2 and did the implementation and experiments (Section 4.6.3). These results were presented at the *4th International Workshop on Experimental and Efficient Algorithms (WEA '05)* [Hüffner 2005].

Chapter 5 is on the application of the color-coding method to the problem of finding signaling pathways in protein interaction networks. Sebastian Wernicke devised the improvement of using more colors in the random coloring process (Section 5.3.1). I developed the speedup by heuristic evaluation func-

tions (Section 5.3.2), designed the data structures (Section 5.3.3), and found the improvements for the PATHWAY QUERY variant (Section 5.3.4). I further did the implementation of the algorithm. The experiments (Section 5.4) were designed by Sebastian Wernicke and mostly carried out by Thomas Zichner. The results were presented at the *5th Asia–Pacific Bioinformatics Conference (APBC '07)* [Hüffner et al. 2007d] and are to appear in *Algorithmica* [Hüffner et al. 2007e]. Thomas Zichner also implemented the graphical user interface (Section 5.5), which was drafted by Sebastian Wernicke and me. An applications note describing the interface was published in *Bioinformatics* [Hüffner et al. 2007f].

Jena, October 2007 Falk Hüffner

Contents

Chapter 1

Introduction

1.1 NP-hard problems

Many combinatorial problems that come up in the real world have been classified as NP-hard [Garey and Johnson 1979]. An example is the following task: After a series of experiments, certain pairs of experiments are found to have conflicting results. We therefore know that for each pair, at least one of the experiments was faulty. We now want to find the minimum number of experiments to discard such that there are no more conflicts. This problem, in its abstract form as a graph problem, is known as Vertex Cover, where given a graph, we search for a minimum number of vertices to delete to get rid of all edges. It is by now an established assumption that NP-hardness implies an inherent combinatorial explosion in the solution space that leads to running times growing exponentially with the input size. This means that large instances of NP-hard problems cannot always be solved efficiently to optimality.

There are several approaches to circumvent this problem in practice. *Heuristics* drop the demand for useful running time guarantees or for useful quality guarantees, and are tuned to run fast with good results on typical instances [Michalewicz and Fogel 2005]. *Approximation algorithms* trade the demand for optimality for a provably efficient running time behavior, while still providing provable bounds on the solution quality [Ausiello et al. 1999, Vazirani 2001]. However, these methods have serious drawbacks. In many applications, it is not acceptable that there is a chance that the algorithm might take exceptionally long in corner cases; and the approximation guarantees that are typically obtained are rather weak (a guarantee such as 10 % error is often not attainable [Ausiello et al. 1999]). In particular, most of the problems we study here are MaxSNP-hard, which implies that one cannot get arbitrarily good approximation factors.

In the next section, we present the approach of *parameterized complexity*, which in many cases can show a way out of this quandary.

1.2 Parameterized complexity and fixed-parameter algorithms

The central observation that motivates parameterized complexity is that often even very large instances of NP-hard problems can be easy. The reason is that they might contain structure that can be exploited. The idea is to measure the structural complexity by a *parameter*, which is usually a nonnegative integer denoted by k. We can then do a two-dimensional complexity analysis, expressing the growth of the running time both in terms of the input size n and in terms of k. The hope is to be able to confine the combinatorial explosion to the parameter, such that we can solve instances fast whenever the parameter is small.

Consider for example the detection of faulty experiments (Vertex Cover) introduced at the start of Section 1.1. Here, it is quite reasonable to assume that not very many experiments are faulty; otherwise, the whole series is useless. Therefore, if we can restrict the combinatorial explosion to the number of faulty experiments k, we can solve even large instances. This concept is made more precise in the following definitions introduced by Downey and Fellows [1999].

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.*

In all examples in this thesis, the parameter is a nonnegative integer, and thus we will assume $L \subseteq \Sigma^* \times \mathbb{N}$. For $(x, k) \in L$, the two dimensions of the parameterized complexity analysis are then the input size $n := |(x, k)|$ and the parameter k. Since in all our applications meaningful parameter values are upper-bounded by $|x|$, we can simply assume $n := |x|$ in asymptotic considerations. The central notion of tractability is then the following.

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

Note that the concept of fixed-parameter tractability is different from the notion of "polynomial-time solvable for fixed k"; an algorithm running in $O(n^k)$ time demonstrates that a problem is polynomial-time solvable for any fixed k, but does not show fixed-parameter tractability.

As an example, we can show that Vertex Cover is fixed-parameter tractable by giving a simple *fixed-parameter algorithm*, that is, an algorithm that has a

running time bound as in Definition 1.2. This algorithm is based on recursive branching. For any conflict involving two experiments a and b, branch into the two cases "a is faulty" and "b is faulty"; clearly, at least one of these assumptions is correct. If we have decided that e. g. a is faulty, then we can remove a from the instance, thereby getting rid of all conflicts involving a. If there are no more conflicts to branch on while we have deleted at most k experiments, then we have found a solution. The size of the thus defined search tree is bounded by $O(2^k)$, since we always branch into two cases and the height of the search tree is bounded by k. A branching can be easily done in $O(n)$ time, and we arrive at an overall running time of $O(2^k n)$. Thus, we can for example solve quite large instances with $k = 30$ and $n = 1000$ within reasonable time.

An important goal in parameterized complexity research is to further bring down the combinatorial explosion. For Vertex Cover, algorithms are known that solve Vertex Cover in $O(1.274^k + kn)$ time [Chen et al. 2006], which even allows to solve instances with $k = 120$ and $n = 1000$.

Unfortunately, parameterized complexity is no silver bullet. There are parameterized problems for which there is good evidence that they are not fixed-parameter tractable. Analogously to the concept of NP-hardness, Downey and Fellows [1999] developed the concept of W[1]-hardness, along with corresponding reduction and completeness notions. For our purposes, it will suffice to know that once a parameterized problem has been classified as W[1]-hard, there is no hope for a fixed-parameter algorithm.

This drawback is much mitigated by the fact that for a single problem, there are many parameters to choose from. The canonical choice is for optimization problems the value that is optimized, such as the number of experiments to omit in our running example. Another possibility is the distance of the instance to some tractable case. For instance, the problem of finding a longest common subsequence for a set of strings is W[1]-hard with respect to several natural parameters such as length of the common subsequence and number of strings, but is FPT when the parameter is the maximum number of occurrences of a letter in a string [Guo et al. 2004].

The classic monograph on parameterized complexity is the one by Downey and Fellows [1999]. Recently, two new books have appeared: one focusing on algorithmic aspects [Niedermeier 2006], and the other on complexity theory [Flum and Grohe 2006].

1.3 Fundamental fixed-parameter techniques

In this section, we briefly survey some FPT techniques that have been used in previous experimental and practical works (see also Hüffner et al. [2008b]).

Clearly, we cannot list all such implementations here, in particular since many examples exist where people employed fixed-parameter algorithms and enjoyed their benefits without being aware of the concept, in particular when the parameter is nonobvious. A famous example [Downey and Fellows 1999] is a type inference algorithm used in a compiler for the ML programming language. Even though it was proved that type inference for ML is EXPTIME-complete and thus in a sense as hard as any problem that requires exponential time, the algorithm could deal easily even with very large inputs. The explanation is that the algorithm is a fixed-parameter algorithm with linear polynomial part, where the parameter is the nesting depth of *let*-constructs, which is very small in real-world programs.

We here omit the three techniques data reduction, iterative compression, and color-coding, since each of them is given its own chapter, where previous work will be described.

1.3.1 Depth-bounded search trees

Search trees are probably the most immediate approach to hard problems; the algorithm we sketched in Section 1.2 to show the fixed-parameter tractability of Vertex Cover is a search tree algorithm. They are a standard technique e. g. for satisfiability solvers, where they are known as DPLL (Davis–Putnam–Logemann–Loveland) algorithms [Davis et al. 1962]; other names used in the literature are "splitting algorithms" or "backtracking algorithms".

The general idea is to examine a small part of the input, and then branch into several cases depending on how the solution for this local structure might look like. If we can bound both the number of cases and the height of the search tree by a function of the parameter k, then we obtain a fixed-parameter algorithm.

Much research in the FPT community has focused on improving the running time of branching algorithms. For example, in a series of papers, the running time of the above mentioned Vertex Cover algorithm was improved to $O(1.274^k + kn)$ time [Chen et al. 2006]. This is mostly done by case distinction. However, it is often found (e. g. Böcker et al. [2007]) that this does not always pay in practice, because of the overhead introduced by the case distinction.

In combination with data reduction (Chapter 3), the use of depth-bounded search trees has proven itself quite useful in practice, allowing to find vertex covers of more than ten thousand vertices in some dense graphs of biological origin [Abu-Khzam et al. 2006]. Search trees also trivially allow for a parallel implementation: when branching into subcases, each process in a parallel setting can further explore one of these branches with no additional communication required. Cheetham et al. [2003] exposed this in their parallel Vertex Cover solver to achieve a near-optimum (i.e., linear with the number of processors

employed) speedup on multiprocessor systems, solving instances with $n \geqslant 700$ and $k \geqslant 400$ in mere hours.

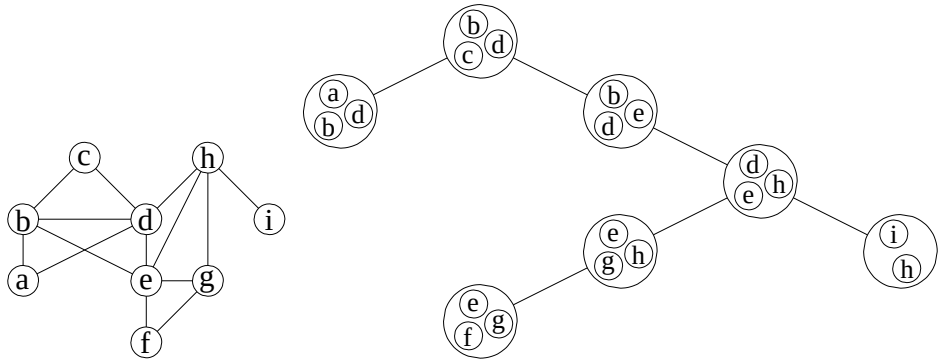
Another success story for search trees is Cluster Editing, the problem of adding and deleting a minimum number of edges of a graph such that every connected component becomes a clique. Dehne et al. [2006] evaluated a search tree algorithm by Gramm et al. [2005] that runs in $O(2.270^k + n^3)$ time, where k is the number of edge modifications. Rahmann et al. [2007] adapted the approach to weighted instances and reported further positive results. Recently, Böcker et al. [2007] improved the worst-case running time to $O(1.83^k + n^3)$ by using a different branching method, thereby even beating out an algorithm running in $O(1.920^k + n^3)$ time that was obtained by a computer-generated search tree algorithm [Gramm et al. 2004]. Böcker et al. [2007] reported that they can compute exact solutions for biological instances of medium size in a matter of minutes.

1.3.2 Dynamic programming

Dynamic programming is another classic algorithm technique with a vast number of applications; see e. g. Cormen et al. [2001]. The running time of a dynamic programming algorithm is usually a function of the table size times a polynomial of the input size. Therefore, if we can restrict the size of the table to a function of the parameter, we can obtain a fixed-parameter algorithm.

An important example is the NP-hard Steiner Tree problem, where the task is, given an edge-weighted graph and a set of vertices called *terminals*, to find a minimum-weight tree that connects all terminals (see Prömel and Steger [2002] for a monograph on Steiner problems). The Dreyfus–Wagner algorithm [Dreyfus and Wagner 1972] can solve the Steiner Tree problem in $O(3^k n^3)$ time, where k is the number of terminals. It thus shows fixed-parameter tractability with the number of terminals as parameter. Its basic idea is to compute Steiner trees for subsets of the terminal set and combine them by dynamic programming to form the solution Steiner tree.

Scott et al. [2005] used Steiner Tree to model the task of finding regulatory subnetworks in protein interaction network (see Betzler [2006] for more details and related problems). Since in this application the number of terminals is small, they could successfully use the Dreyfus–Wagner algorithm to find exact solutions. Ganley [1999] noted that for the special case of rectilinear Steiner trees, the Dreyfus–Wagner algorithm is probably the most popular method for computing optimal solutions in practice. Recently, improvements of the running time to $(2+\varepsilon)^k \cdot n^{O(1)}$ for any fixed ε [Fuchs et al. 2007] and, for the case that weights are integers bounded by M, to $O(2^k \cdot n^2 M + nm \log M)$ [Björklund et al. 2007] were given. It is not clear that these are practical, though, since the



(a) Graph G (b) Tree decomposition of G

Figure 1.1: A graph together with a tree decomposition of width 2

former suffers from a large polynomial time overhead, and the latter from a large polynomial space overhead.

A major problem with dynamic programming for NP-hard problems is that it typically requires exponential amounts of memory. Therefore, the problem is often not that the algorithms run too slow, but rather that they run out of memory after only a few seconds or minutes. We show some ways of dealing with this problem in Section 5.3, where dynamic programming is used in a subroutine of the color-coding method.

1.3.3 Tree decompositions

Many NP-hard graph problems such as Vertex Cover become polynomial-time solvable when the input is restricted to trees. Based on this, Coppersmith and Vishkin [1985] gave an algorithm for Vertex Cover that runs fast on "almost trees"; more precisely, they give an algorithm running in $O(2^k \cdot |G|)$ time, where k is the number of edges that needs to be deleted in the input graph G to obtain a tree. Treewidth is a more general measure of "tree-likeness", defined via the concept of a *tree decomposition*.

Definition 1.3. *Let* $G = (V, E)$ *be a graph. A* tree decomposition *of* G *is a pair* $\langle \{X_i \mid i \in I\}, T \rangle$*, where each* X_i *is a subset of* V *called a* bag*, and* T *is a tree with the elements of* I *as nodes. The following three properties must hold:*

1. $\bigcup_{i \in I} X_i = V$*;*
2. *for every edge* $\{u, v\} \in E$*, there is an* $i \in I$ *such that* $\{u, v\} \subseteq X_i$*; and*

3. *for all* $i, j, k \in I$, *if* j *lies on the path between* i *and* k *in* T, *then* $X_i \cap X_k \subseteq X_j$.

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

Note that subtracting one from the maximum bag size to obtain the width of a tree decomposition is mostly for aesthetic reasons, so that trees have a treewidth of 1 and not 2. An equivalent, more intuitive definition is given by Seymour and Thomas [1993] using a "cops-and-robber game".

Figure 1.1 shows a graph together with an optimal tree decomposition of width two. In a tree decomposition $\langle B, T \rangle$ of width k, each bag is a *separator* with at most $k+1$ vertices, that is, after its deletion the graph decomposes into at least two connected components. This allows to solve many problems using a table that uses only $f(k)$ space for some function f not depending on n and doing dynamic programming bottom-up from the leaves of T. Thus, these problems are fixed-parameter tractable with respect to the parameter k. For example for Vertex Cover, we can for a bag X_i use a table of size 2^{k+1} that stores for each possible allocation of the vertices in X_i to one of the two states "in the cover" and "not in the cover" the minimum cost of a vertex cover for the subgraph of G corresponding to the subtree below i in T. Further examples are Independent Set and Dominating Set, which can be solved in $O(2^k n)$ and $O(4^k n)$ time, respectively [Alber and Niedermeier 2002]. For these two problems, this is a particularly appealing result, since for their most natural parameter (the solution size), one get W[1]-hardness results.

Positive experimental results with tree decomposition approaches have been reported by Alber et al. [2005] for Vertex Cover. Another practical application was given by Xu et al. [2005], who proposed a tree decomposition based algorithm to solve two problems encountered in protein structure prediction (a survey covering related results is given by Cai et al. [2007]).

A drawback of treewidth-based algorithms is that finding an optimal tree decomposition is already NP-hard; however, various exact or heuristic approaches exist (see e. g. Bodlaender and Koster [2007]). Finding the optimal treewidth is also fixed-parameter tractable with respect to k [Bodlaender 1996]; however, the known algorithms have impractical running times.

1.4 Algorithm engineering

In recent times, there is a growing gap between algorithm theory and the algorithms actually used in practice. The reason is a difference in methodology and goals. In algorithm theory, one considers simple, abstract problem models and simple, abstract machine models. The solutions are sophisticated, and the

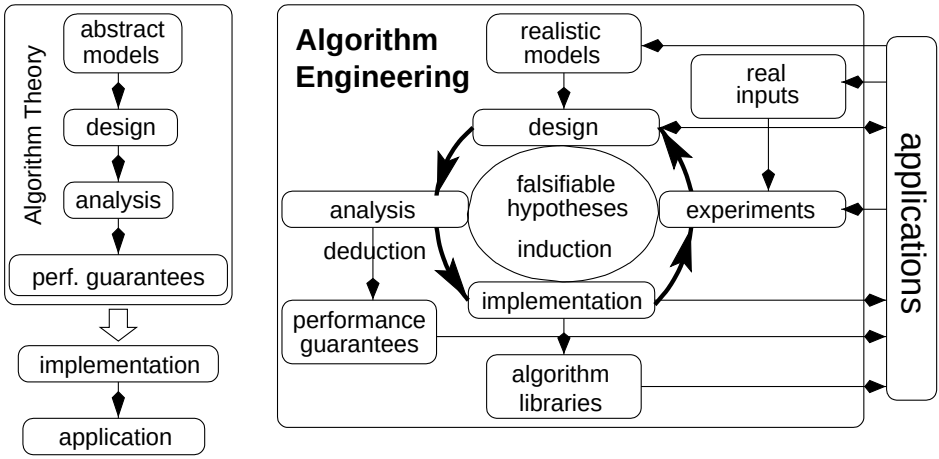


Figure 1.2: Classic algorithm theory versus algorithm engineering (from Mehlhorn et al. [2006])

main goal is provable performance guarantees. In practice, one has to deal with complex requirements and real machines. Much preferred are simple implementations, and the goal is good observed performance. In the worst case, these differences in viewpoint lead to unusable solutions from algorithm theorists and to badly scaling, unreliable solutions from the practitioners.

The goal of algorithm engineering is to bridge this gap. A central demand is that research in algorithmics should also be concerned with implementations and experimental evaluations. In fact, algorithm engineering has recently become synonymous with a change in the overall methodology of algorithm research, as illustrated in Figure 1.2. Classic algorithm theory works in a linear fashion: From the application, an abstract model is distilled and an algorithm is designed. Then this algorithm is mathematically analyzed, leading to performance guarantees. At this point, the work is often considered done; implementation and application are typically done as an afterthought, or by other groups.

Initially, the approach of algorithm engineering is the same: from the application, a model is extracted, although more emphasis is put on realistic models. An algorithm is designed and analyzed, providing performance guarantees. After this, however, implementation and experimental evaluation are not considered optional, but as an integral component of the process. In particular, real inputs from the application are to be used. The main difference to classic algorithm theory is that the results from experiments are used in a feedback loop to influence the design of updated algorithms. This cycle of design, analysis, implementation,

and experiments is driven by falsifiable hypotheses about how the algorithms behave, and thus differs fundamentally from the strictly deductive approach that aims only for performance guarantees. Further, the development of algorithm libraries that are directly usable by practitioners is seen as another important research goal.

A successful example of algorithm engineering is route planning in road networks. In classic algorithm theory, it is clear what to do: this is a shortest path problem in a graph, and the best algorithm we have for it is Dijkstra's. However, for the huge road networks we have to deal with, it is just too slow. The first idea is to adapt the model: we allow preprocessing of the graph to be able to quickly answer many queries. But the obvious solution of storing all shortest paths again does not work: it requires too much memory. Therefore, one has to exploit properties of real street graphs. For example, Bast et al. [2007] used the observation that for large distances, one leaves the current location via one of only a few important traffic junctions. From this, they developed the concept of *transit nodes*, which in their experiments allowed to execute queries up to six orders of magnitude faster than Dijkstra's algorithm.

In my research, I have tried to initiate research in several areas, following the guidelines of algorithm engineering. Probably the best example is Chapter 5 dealing with color-coding: here, we have improved the running times for real-world instances by several orders of magnitude, and provided a user-friendly tool with a graphical interface. However, since algorithm engineering encompasses and extends classic algorithm theory, it is an ambitious undertaking that takes many resources and usually requires the cooperation of several people; therefore, some parts of this thesis need to be seen as an invitation for further research following the concept of algorithm engineering.

More information on algorithm engineering can be found in the proceedings of the *Workshop on Algorithm Engineering and Experiments (ALENEX)*, the *European Symposium on Algorithms* (Track B), and the *Workshop on Experimental Algorithms (WEA)*, and further the *ACM Journal of Experimental Algorithmics* or the website http://www.algorithm-engineering.de/ of the DFG priority program 1307 on algorithm engineering.

1.5 Notation

In this section, we introduce some basic notation used throughout this work.

For a set S and an element x, we write $S - x$ for the set $S \setminus \{x\}$. An *optimization problem* is the problem of finding the best solution from all feasible solutions. More formally, an optimization problem A is a quadruple (I, f, m, g), where

- I is a set of instances;

- given an instance $x \in I$, $f(x)$ is the set of feasible solutions;
- given an instance x and a feasible solution y of x, $m(x, y)$ denotes the *measure* of y, which is usually a positive real number;
- g is the goal function, and is either min or max.

The goal is then to find for some instance x an *optimal solution*, that is, a feasible solution y with

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}. \tag{1.1}$$

For a minimization problem (that is, an optimization problem with $g = \min$), a feasible solution is called *minimal* if it does not contain another feasible solution as a proper subset and *minimum* if there is no other feasible solution with better measure. We extend weight functions $\omega : X \to \mathbb{Q}$ to sets in the natural way, that is, we set $\omega(S) := \sum_{x \in S} \omega(x)$ for $S \subseteq \mathcal{P}(X)$.

Undirected graphs. For a general introduction to graph theory, see Diestel [2005]. An undirected graph G is a tuple (V, E), where V is a finite set of *vertices* and E is a set of *edges*, which are unordered tuples of vertices, that is, $E \subseteq \{\{u, v\} \mid u, v \in V\}$. For a graph $G = (V, E)$, we set $V(G) := V$ and $E(G) := E$. For some $v \in V$, the set $N(v) := \{u \in V \mid \{u, v\} \in E\}$ is the set of *neighbors* of v. The *closed* neighborhood of vertex v, denoted by $N[v]$, is equal to $N(v) \cup \{v\}$. The *degree* of a vertex v is its number of neighbors $|N(v)|$. We call a graph $G' = (V', E')$ *induced subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' = \{\{u, v\} \mid u, v \in V' \text{ and } \{u, v\} \in E\}$. The graph induced by a vertex set V' in G is denoted $G[V']$. With $G \setminus V'$ for some $V' \subseteq V$, we denote the graph $G[V \setminus V']$. A graph G' is a *minor* of a graph G if G' can be obtained from a subgraph of G by contracting edges, where "contracting an edge" means to delete the edge and to identify its endpoints.

A *path* is a sequence of vertices $v_1, \ldots, v_s$ with $\{v_i, v_{i+1}\} \in E$ for all $1 \leqslant i < s$. A graph is *connected* when there is a path between any two vertices. A *cycle* is a path with $\{v_s, v_1\} \in E$. A *clique* is a complete graph, that is, a graph where $\forall v, w \in V, v \neq w : \{v, w\} \in E$. We use special notation for a number of particular graphs: a P_n is an induced path of n vertices, a C_n is an induced cycle of n vertices, and a K_n is a clique of n vertices.

The problems we examine typically have as input a graph $G = (V, E)$. We then use n to refer to the number of vertices of the input instances, and m to refer to the number of edges. In all problems we examine, degree-0 vertices do not play a role. Therefore, to simplify the running time analysis, we always assume that $n = O(m)$.

A *cut* or *edge cut* between two disjoint vertex sets in a graph is a set of edges whose removal disconnects these two sets in the graph. Similarly, a *separator* in a connected graph is a set of vertices whose deletion makes the graph disconnected.

A *graph class* is a set of graphs that is closed under isomorphism. A large number of graph classes has been considered in the literature (see Brandstädt et al. [1999] for a survey, covering more than 200 classes).

We frequently use the following characterizations of the class of *bipartite graphs* [Kőnig 1936].

Lemma 1.1. *For a graph* $G = (V, E)$*, the following are equivalent:*

1. G *is* bipartite*, that is,* V *can be partitioned into two sets* V_1 *and* V_2 *called* sides *such that there is no* $\{v, w\} \in E$ *with both* $v, w \in V_1$ *or both* $v, w \in V_2$*. In other words,* G *has a cut of size* m*.*

2. V *can be colored with two colors such that for all* $\{v, w\} \in E$ *the vertices* v *and* w *have different colors. The color classes correspond to the sides.*

3. G *does not contain* odd cycles*, that is, cycles of odd length.*

A graph class is *hereditary* if it is closed under deleting vertices. Note that sometimes in literature, "hereditary" is defined as closed under deleting vertices and edges; for this, we use the term *monotone*.

A *hypergraph* is a generalization of an undirected graph, where an edge may contain more than two vertices. We use the same notation for hypergraphs as for graphs.

Directed graphs. For a general introduction to directed graphs, see Bang-Jensen and Gutin [2002]. A directed graph or *digraph* D is a tuple (V, A), where V is a finite set of *vertices* and A is a set of *arcs*, which are ordered tuples of vertices, that is, $A \subseteq V^2$. We consider only digraphs without loops, that is, $(v, v) \notin A$ for all $v \in V$. We call a graph $D' = (V', A')$ *induced subgraph* of a digraph $G = (V, A)$ iff $V' \subseteq V$ and $E' = \{(u, v) \mid u, v \in V' \text{ and } (u, v) \in A\}$. A *tournament* $T = (V, A)$ is a digraph where there is exactly one arc between each pair of vertices. A *cycle* is a sequence of distinct vertices $v_1, \ldots, v_s$ with $(v_i, v_{i+1}) \in A$ for all $1 \leqslant i < s$ and $(v_s, v_1) \in A$. A *triangle* is a cycle of length 3. A *topological sort* of a digraph $D = (V, A)$ is a sequence $v_1, v_2, \ldots, v_n$ of the vertices in V in which each vertex appears exactly once and $i < j$ for each arc $(v_i, v_j) \in A$. Clearly, a digraph has a topological sort iff it is acyclic, that is, it does not contain a cycle.

Chapter 2

Problems

In this chapter, we introduce some of the problems we are dealing with and state previous results. We first focus on one class of problems, namely graph modification problems (Section 2.1), which appear in several instances in our work. We further highlight the three problems Clique Cover (Section 2.2), Bipartization (Section 2.3), and Minimum-Weight Path (Section 2.4), since they are central to our algorithms and further serve as examples to illustrate the rich applications of NP-hard graph problems. Other problems are introduced at the places where we deal with them algorithmically.

2.1 Graph modification problems

Graph modification problems are a popular means of modelling tasks such as error correction, conflict resolution, and reconfiguration in networks. In a graph modification problem, the task is to delete at most k vertices or at most k edges from a graph such that it becomes a member of a particular graph class. In addition to vertex deletion and edge deletion, other modification operations such as edge additions and combinations of modifications have been considered. Graph modification problems have a large number of applications (see e. g. Sharan [2002], Guo [2006] and references therein), and many well-known problems such as Vertex Cover can be reformulated as graph modification problems. All vertex deletion problems for hereditary graph classes are NP-hard [Lewis and Yannakakis 1980] (except for *trivial* classes, that is, classes for which only finitely many graphs are inside or outside the class). For edge deletion problems, no such characterization is known, although there are some works which show edge deletion problems to be NP-hard for a wide set of graph classes

(e. g., Lewis [1978], El-Mallah and Colbourn [1988], Asano and Hirata [1982]). Burzyn et al. [2006] listed a large number of standard graph classes, and in each case the edge deletion problem is NP-hard. However, there are exceptions, for example finding a minimum set of edges to delete to make a connected graph cycle-free can be done in polynomial time by finding a maximum spanning tree.

It is well-known (see e. g. Greenwell et al. [1973]) that hereditary graph classes are exactly those classes that can be characterized by a set of forbidden subgraphs. Sometimes this set is finite (for example, trivially perfect graphs are graphs that do not contain a P_4 or C_4), and sometimes it is infinite (for example for cycle-free graphs). If we have a finite characterization by forbidden subgraphs, then we can solve both the vertex- and the edge-deletion problem by a simple search tree algorithm: Find a forbidden subgraph, and branch into a number of cases corresponding to deleting a vertex or an edge of this subgraph. Cai [1996] showed that this even works when we do not know the set of forbidden subgraphs, but only that it is finite. Therefore, all vertex- and edge-deletion problems that can be characterized by a finite set of forbidden subgraphs are in FPT. For hereditary graph classes that require an infinite number of forbidden subgraphs, no such simple characterization is known. Khot and Raman [2002] provide a complete characterization of problems that are in FPT for the parametric dual of graph modification for hereditary graph classes (that is, finding a subgraph or induced subgraph of size at least k that belongs to a certain hereditary graph class).

For vertex deletion problems, strong hardness results for approximation are also known: Lund and Yannakakis [1993] showed that any vertex deletion problem for a nontrivial hereditary graph class is MaxSNP-hard. They even conjecture that problems that require an infinite forbidden subgraph characterization do not have a constant approximation factor.

Graph modification problems we deal with in this work are Edge Bipartization, Balanced Subgraph, and Vertex Bipartization, which are introduced in-depth in Section 2.3. We further examine Cluster Vertex Deletion (Section 4.3) and Feedback Vertex Set in tournaments (Section 4.4).

2.2 Clique Cover

The Clique Cover problem is a fundamental graph problem that occurs in many contexts and applications. It is also known as Keyword Conflict problem [Kellerman 1973] or Covering by Cliques (GT17) or Intersection Graph Basis (GT59) [Garey and Johnson 1979].

Clique Cover
Instance: An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Is there a set of at most k cliques in G such that each edge in E has both its endpoints in at least one of the selected cliques?

We remark that covering *vertices* by cliques (Vertex Clique Cover or Clique Partition) is of less interest to be studied on its own because it is equivalent to the well-investigated Graph Coloring problem: A graph has a vertex clique cover of size k iff its complement graph can be colored with k colors such that adjacent vertices have different colors. The variant of Clique Cover where it is required that the covering cliques are edge disjoint has also been examined in the parameterized context [Mujuni and Rosamond 2008].

As first observed by Erdős et al. [1966], Clique Cover is equivalent to a problem from intersection graph theory (see McKee and McMorris [1999] for a monograph on intersection graphs). Let $\mathcal{F} = \{S_1, \ldots, S_n\}$ be a family of sets. The *intersection graph* of $\mathcal{F}$, denoted $\Omega(\mathcal{F})$, is the graph having $\mathcal{F}$ as vertex set with S_i adjacent to S_j iff $i \neq j$ and $S_i \cap S_j \neq \emptyset$. It is easy to see that for every *feature* $x \in \mathcal{U}(\mathcal{F}) := \bigcup_{S \in \mathcal{F}} S$, the set $C_x := \{S_i \in \mathcal{F} \mid x \in S_i\}$ forms a clique in $\Omega(\mathcal{F})$, and $\{C_x \mid x \in \mathcal{U}(\mathcal{F})\}$ is a clique cover for $\Omega(\mathcal{F})$. Therefore, finding a minimum cardinality clique cover for a graph G is equivalent to finding a set intersection representation $\mathcal{F}$ for G that minimizes $|\mathcal{U}(\mathcal{F})|$ (called Intersection Graph Basis by Garey and Johnson [1979]). Guillaume and Latapy [2004] argue that this model is very widely applicable to discover underlying structure in complex real-world networks.

Hardness and approximation. Clique Cover is NP-hard [Orlin 1977], even when restricted to planar graphs [Chang and Müller 2001] or graphs with maximum degree 6 [Hoover 1992]. Further, Clique Cover is not approximable within a factor of n^{ε} for some $\varepsilon > 0$ unless P = NP [Lund and Yannakakis 1994], and nothing better than a polynomial approximation factor of $O(n^2(\log\log n)^2/(\log n)^3)$ is known [Ausiello et al. 1999].

Special cases. Clique Cover is polynomial-time solvable for chordal graphs [Ma et al. 1989] (a chordal graph has at most n maximal cliques), graphs with maximum degree 5 [Hoover 1992], line graphs [Orlin 1977], and circular arc graphs [Hsu and Tsai 1991].

Experimental results. There has been substantial work on polynomial-time heuristic algorithms for Clique Cover [Kellerman 1973, Kou et al. 1978, Rajagopalan et al. 2000, Piepho 2004, Gramm et al. 2007b, Behrisch and Taraz 2006]. Most of these provided no quality guarantees; one exception is Behrisch

and Taraz [2006], who gave simple greedy algorithms for Clique Cover that provide asymptotically optimal solutions for certain random intersection graphs. Rajagopalan et al. [2000] compared a heuristic [Kou et al. 1978] and an exact approach using graph coloring software.

Applications. Applications of Clique Cover include compact representation of visibility graphs [Agarwal et al. 1994] and circuit synthesis [Khomenko 2007]. It also has several applications in compiler construction, for example as Keyword Conflict problem [Kou et al. 1978], for optimal placement of synchronization barriers [O'Boyle and Stöhr 2002], and for instruction scheduling [Rajagopalan et al. 2000, 2001].

We now look closer at an application from statistics visualization [Piepho 2004], which will also play a role in our experiments in Section 3.1.3. Consider for example a series of crop yield trials, each under different conditions. We can use multiple pairwise comparisons to classify each pair of trials as *significantly different* (for example, if the mean of the yield differs by more than a threshold value) or not significantly different. The task is now to represent these $O(n^2)$ comparisons for n trials compactly.

The *letter display* was suggested by Piepho [2004] as a generally applicable method for displaying significant differences. For a given set of n trials and a set H of m pairwise comparisons, where $\{T_i, T_j\} \in H$ iff the trials T_i and T_j are significantly different, a letter display is a matrix $\mathbf{M}$ with n rows, one row for each trial, which satisfies the following three conditions:

1. Each column contains a different *letter* and all non-empty entries of one column contain the same letter.

2. Every row has at least one non-empty entry.

3. Two trials differ significantly iff the corresponding two rows of $\mathbf{M}$ contain no common letter.

Since by the first condition, the actual letter used is completely determined by the column, we can simplify the task somewhat by thinking of $\mathbf{M}$ as a binary matrix.

Clearly, it is always possible to find a letter display by creating a new column with two letters for each pair of not significantly different trials. However, displays created this way may get very large; they can have $\Omega(n^2)$ columns. To be useful, we want to minimize the number of columns, which gives rise to the following problem.

1	×			
2				×
3	×			
4	×	×		
5		×		×
6	×		×	
7			×	×
8			×	

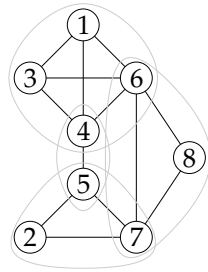


Figure 2.1: Equivalence of Compact Letter Display and Clique Cover for $H = \{\{1,2\},\{1,5\},\{1,7\},\{1,8\},\{2,3\},\{2,4\},\{2,6\},\{2,8\},\{3,5\},\{3,7\},\{3,8\},\{4,7\},\{4,8\},\{5,6\},\{5,8\}\}$.

Compact Letter Display
Instance: A set T of trials, a set H of pairs of trials, that is, $H \subseteq \{\{t_1, t_2\} \mid t_1, t_2 \in T\}$, which gives the pairs of significantly different trials, and an integer $k \geqslant 0$.
Question: Is there a letter display for T and H that uses at most k columns?

In Figure 2.1, we illustrate that Compact Letter Display and Clique Cover are in fact equivalent, in the sense that trivial reductions in both directions exist. For this, we establish the following correspondences between a Compact Letter Display instance and a Clique Cover instance:

- Trials or rows of the letter display correspond to graph vertices;
- Pairs of not significantly different trials correspond to edges, that is, the elements of H correspond to non-edges;
- Letter display columns correspond to cliques in the clique cover.

For example, trials 5 and 7 occur together in the fourth column of the letter display; this means that they are not significantly different, and there is an edge between them in the corresponding Clique Cover instance. In contrast, trials 4 and 7 do not occur together in a column, are thus significantly different, and are not connected by an edge in the Clique Cover instance.

It is then not too hard to see that, with these correspondences, a solution for Clique Cover implies a solution of the same size for the corresponding Compact Letter Display instance, and vice versa. Special handling is only required for trials significantly different from all other trials (corresponding to vertices with no edges attached): they require a letter display column, but no clique is needed to cover them in the given problem formulation. This is easily accounted for by a straightforward preprocessing step.

2.3 Bipartization and Balanced Subgraph

In this section, we introduce the Edge Bipartization problem and its generalization Balanced Subgraph. We further examine the variant Vertex Bipartization. These problems have many applications in areas such as computational biology, social networks, statistical physics, and VLSI design.

2.3.1 Edge Bipartization

We start with the Edge Bipartization problem, also known as (unweighted) Minimum Uncut.

> Edge Bipartization
> **Instance:** An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
> **Question:** Can G be transformed by up to k edge deletions into a bipartite graph?

Edge Bipartization is the parametric dual of the better known MaxCut problem (that is, they can be transferred into each other by replacing the parameter k by $m - k$).

> MaxCut
> **Instance:** An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
> **Question:** Does G have a cut of size at least k?

Let an *edge bipartization set* be a set of edges whose deletion makes a graph bipartite. Then G has an edge bipartization set of size k iff G has a cut of size $m-k$, and the sets can be transformed into each other by taking the complement with respect to E. Thus, as decision questions, these problems are equivalent. However, when considering parameterized complexity or approximation algorithms, we have to differentiate between the two problems.

Hardness and approximation. Edge Bipartization is one of the 21 problems for which NP-hardness was first shown by reduction [Karp 1972]. It remains NP-hard even in triangle-free graphs with maximum degree 3 [Yannakakis 1981]. The problem is known to be MaxSNP-hard [Papadimitriou and Yannakakis 1991] and can be approximated to a factor of $O(\sqrt{\log n})$ in polynomial time [Agarwal et al. 2005]. Another approximation algorithm finds in polynomial time a solution of size $O(k \log k)$, where k is the size of an optimal solution [Avidor and Langberg 2007]. Assuming Khot's Unique Games Conjecture, it is NP-hard to approximate Edge Bipartization within any constant factor [Khot 2002]. For MaxCut, much better guarantees exist: by semidefinite programming,

it can be approximated by a factor of 0.878 [Goemans and Williamson 1995], although it is NP-hard to improve this to a factor of 0.942 [Håstad 2001]; in fact, it has been conjectured that the factor cannot be improved at all [Khot et al. 2007].

Special cases. In planar graphs, Edge Bipartization is solvable in polynomial time [Orlova and Dorfman 1972, Hadlock 1975]; however, it is NP-hard in planar graphs with only a single vertex added [Barahona 1980]. Further, it is polynomially solvable in *weakly bipartite* graphs [Grötschel and Pulleyblank 1981], which comprise bipartite graphs and planar graphs. Weakly bipartite graphs can be characterized as graphs that do not contain a K_5 as minor [Guenin 2001]. Edge Bipartization can further be solved in polynomial time for graphs with any fixed bounded genus [Galluccio et al. 2001], and it can be solved in $O(n^{4c})$ time for graphs where the length of odd cycles is bounded by $2c + 1$ [Grötschel and Nemhauser 1984]. Bodlaender and Jansen [2000] examined a number of further graph classes and showed for example that Edge Bipartization is NP-hard in chordal graphs and split graphs, but polynomial-time solvable in bounded-treewidth graphs and cographs. Bodlaender et al. [2005] showed that Edge Bipartization is polynomial-time solvable in split-indifference graphs and graphs with few P_4's. Finally, Díaz and Kamiński [2007] proved that Edge Bipartization is NP-hard in unit disk graphs.

Parameterized complexity. Edge Bipartization is a graph modification problem for a hereditary graph class, but the set of forbidden subgraphs (which by Lemma 1.1 contains all odd cycles) is infinite. Therefore, it is not immediately clear that Edge Bipartization is fixed-parameter tractable. The parametric dual problem MaxCut is in FPT; the best currently known algorithm [Raman and Saurabh 2007] runs in $1.242^k n^{O(1)}$ time. The fact that MaxCut is in FPT might have suggested that Edge Bipartization is not, since this is often observed for parametrically dual problems [Khot and Raman 2002]. However, based on the *iterative compression* technique, Reed et al. [2004] presented a fixed-parameter algorithm for the closely related Vertex Bipartization problem, introduced below. Together with a simple parameter-preserving reduction from Edge Bipartization to Vertex Bipartization [Wernicke 2003], one can use the algorithm by Reed et al. [2004] to obtain a running time of $O(3^k \cdot k^3 m^2 n)$ for Edge Bipartization, thus proving that Edge Bipartization is fixed-parameter tractable. In Section 4.5, we show how to improve this to $O(2^k \cdot m^2)$ time, also using an iterative compression algorithm.

Exact algorithms. Williams [2005] presented an exact algorithm for Edge Bipartization, which can solve an instance in $O(1.732^n m^3)$ time, the first improvement for the exponential part over the trivial $O(2^n m)$ algorithm. Unfortunately, this algorithm requires exponential space, and is thus probably unusable in practice; no polynomial-space algorithm improving over the trivial bound is known. There has also been some work regarding sparse instances, that is, instances with few edges. Kneis et al. [2005] give an algorithm running in $1.128^m m^{O(1)}$ time with exponential space or $1.143^m m^{O(1)}$ with polynomial space (see also Scott and Sorkin [2007], who point out some errors of Kneis et al. [2005]).

Applications. The problem has applications in genome sequence assembly [Pop et al. 2004], VLSI chip design [Barahona et al. 1988, Kahng et al. 2001], or the calculation of the stability of fullerene molecules [Došlić and Vukičević 2007].

Experimental results. Possibly the most common practical approach to exactly solving Edge Bipartization (or MaxCut) is to use mathematical programming techniques. In this context, usually the weighted version is considered. Barahona et al. [1985] examine the polytope of the maximum-weight bipartite subgraph. Grötschel et al. [1987] present some experimental results using polytope approaches. More recently, Liers et al. [2004] describe a solution using branch & cut, and Rendl et al. [2007] present a MaxCut implementation based on branch & bound and the semidefinite relaxation.

A large number of heuristics without solution quality guarantees, such as genetic algorithms, have also been suggested; see e. g. Dolezal et al. [1999] and Festa et al. [2002] for experimental comparative studies.

2.3.2 Balanced Subgraph

The Balanced Subgraph problem is a generalization of Edge Bipartization. It is defined on *signed graphs*, which are undirected graphs where each edge is annotated with an element of the sign group (that is, the unique two-element group, which can for example be denoted by the two elements 0 and 1 and the binary operation $a \circ b := (a + b) \bmod 2$). The concept of signed graphs has been introduced first by Harary [1953] in the context of social networks, and has been rediscovered frequently since, as it is a natural model for many applications; see Zaslavsky [1998] for a bibliography of signed graphs. The central concept is that of a *balanced* signed graph.

Definition 2.1. *A signed graph* $G = (V, E)$ *with edges labeled by* $h : E \to \{0, 1\}$ *is*

balanced *if there is a vertex coloring* $f : V \to \{0,1\}$ *such that*

$$\forall \{u,v\} \in E : h(\{u,v\}) \equiv (f(u) + f(v)) \pmod 2. \tag{2.1}$$

Put another way, a 0-edge demands that its endpoints have the same color, and a 1-edge demands that they have different colors. Therefore, in the following we use the notations "=-edge" and "≠-edge" instead. Let further $E_=$ be the set of =-edges and $E_\neq$ the set of ≠-edges.

Balanced graphs generalize bipartite graphs, since bipartite graphs are balanced graphs that contain only ≠-edges. Kőnig [1936] proved the following characterization of balanced graphs (in fact, the well-known characterization of bipartite graphs (Lemma 1.1) is only a corollary of this result).

Lemma 2.1. *For a graph* $G = (V,E)$*, the following are equivalent:*

1. G *is* balanced*, that is,* V *can be partitioned into two sets* V_1 *and* V_2 *called* sides *such that there is no* ≠*-edge* $\{v,w\} \in E$ *with both* $v,w \in V_1$ *or both* $v,w \in V_2$ *and no* =*-edge* $\{v,w\}$ *with* $v \in V_1$ *and* $w \in V_2$*.*
2. V *can be colored with two colors such that for all* $\{v,w\} \in E_\neq$ *the vertices* v *and* w *have different colors, and for all* $\{v,w\} \in E_=$ *the vertices* v *and* w *have the same color. The color classes correspond to the sides.*
3. G *does not contain* unbalanced cycles*, that is, cycles with an odd number of* ≠*-edges.*

Using the characterization by a coloring, it is easy to see that balance of a signed graph can be checked in linear time by depth-first search. The Balanced Subgraph problem is now defined as follows:

Balanced Subgraph
Instance: A signed graph $G = (V,E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k edge deletions into a balanced graph?

Clearly, Balanced Subgraph is a generalization of Edge Bipartization and thus NP-hardness and approximation hardness results carry over. Moreover, there is a simple reduction from Balanced Subgraph to Edge Bipartization, which allows us to transfer some tractability results from Edge Bipartization. For this, we subdivide each =-edge by one vertex (see Figure 2.2):

Graph Transformation 2.1. *Given a signed graph* $G = (V,E)$*, construct* $G' = (V',E')$ *with* $V' := V \cup E_=$ *and* $E' := E \setminus E_= \cup \{\{v,e\},\{e,w\} \mid e = \{v,w\} \in E_=\}$.

Proposition 2.1. *A* Balanced Subgraph *instance can be solved with* k *edge deletions iff the transformed* Edge Bipartization *instance can be solved with* k *edge deletions.*

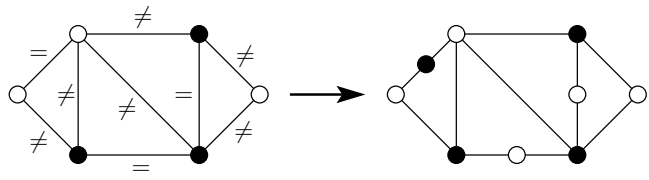


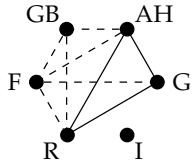
Figure 2.2: Example for reducing Balanced Subgraph to Edge Bipartization. Colors serve to indicate that the graph is balanced (a) resp. bipartite (b).

Proof. Let X be a solution of the Balanced Subgraph instance $G = (V, E)$ with $|X| = k$ and c a balanced coloring of $G \setminus X$. Then let c' be a coloring for the transformed instance where the vertices from V have the same color as in c and newly introduced vertices have the opposite color of both of their two neighbors (this is well-defined because the new vertices subdivide $=$-edges, whose endpoints must be colored equally by c). Then there are at most k edges whose endpoints are colored equally by c', since that can only happen for edges in X.

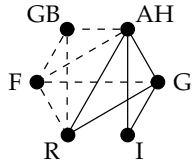
Similarly, let X be a solution of Edge Bipartization on the transformed instance with $|X| = k$ and c a two-coloring of $G \setminus X$. Then $c|_V$ (that is, c restricted to V) has at most k conflicts with the signs of G, since a conflict can only occur when a $\neq$-edge was deleted in X or one of the two edges that replaced a $=$-edge was deleted. □

From Proposition 2.1, we obtain the fixed-parameter tractability of Balanced Subgraph, and obtain the same approximation factors as for Edge Bipartization. With respect to the maximization variant, we can, however, not directly obtain the same approximation factor of 0.878 as for MaxCut [Goemans and Williamson 1995], since the number of edges might double. However, it was shown by Thagard and Verbeurgt [1998] and independently by DasGupta et al. [2007] that the semidefinite programming of Goemans and Williamson [1995] can be adapted to Balanced Subgraph without impairing the approximation factor. Balanced Subgraph is also polynomial-time solvable in planar graphs and weakly bipartite graphs, since these classes are closed under Graph Transformation 2.1.

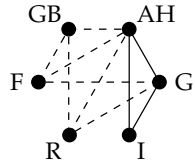
Applications. Balanced Subgraph has a large number of applications. One of the oldest is in modeling social networks [Harary 1959]. Here, an $=$-edge models a positive or friendly connection, a $\neq$-edge models a negative or un-



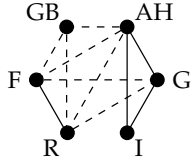
Three emperor's league 1872–81



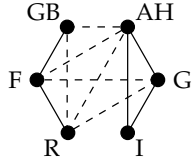
Triple alliance 1882



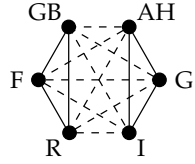
German–Russian lapse 1890



French–Russian alliance 1891–94



Entente cordiale 1904



British–Russian alliance 1907

Figure 2.3: Evolution of the major relationship changes from 1872 to 1907 between the protagonists of World War I. Here, GB = Great Britain, AH = Austria–Hungary, G = Germany, I = Italy, R = Russia, and F = France. *Solid lines* denote friendly links and *dashed lines* unfriendly links. From Antal et al. [2006].

friendly connection, and a non-edge a neutral connection or lack of contact. The conjecture is that changes in social networks can be explained by a striving towards balance in this signed graph. The number of edge deletions required to obtain a balanced graph is then a measure of the distance from stabiblity. An example is given by Antal et al. [2006] for the relations between nations prior to World War I (see Figure 2.3). The claim is that a number of events gradually led to a reorganization of the relations between European nations into a socially balanced state. Clearly, this model is not perfect, though: it cannot explain why Italy actually fought on the side of the Entente Powers that include Russia, France, and the British Empire.

Another application in bioinformatics will be central to our experiments in Section 3.2.4. DasGupta et al. [2007] used balance in signed graphs to model the concept of "monotone subsystems", under the name of "sign-consistent graphs". They examine dynamic systems, where an activating connection is modeled as an $=$-edge and an inhibiting connection is modeled as a $\neq$-edge. The claim is that biological dynamical systems are close to being balanced, and that finding a minimum set of edges to delete to make the graph balanced can be used to decompose the graph into "monotone subsystems", which exhibit stable behavior and thus allow a better understanding of the dynamics of a system.

Further applications of Balanced Subgraph appear in statistical physics [Barahona 1982], portfolio risk analysis [Harary et al. 2002], and VLSI design

[Chiang et al. 2007].

Experimental results. Similar to Edge Bipartization, polyhedral approaches have been used for Balanced Subgraph [Barahona and Ridha Mahjoub 1989, Boros and Hammer 1991], which also cover the weighted case. DasGupta et al. [2007] implemented the semidefinite programming based approximation algorithm.

2.3.3 Vertex Bipartization

Vertex Bipartization, also known as Graph Bipartization, Maximum Bipartite Induced Subgraph or Odd Cycle Transversal, is the vertex deletion version of Edge Bipartization. It has been noted (e. g. by Yannakakis [1981]) that vertex-deletion problems tend to be computationally harder than edge-deletion problems. It turns out this is true on several counts here.

> Vertex Bipartization
> **Instance:** An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
> **Question:** Can G be transformed by up to k vertex deletions into a bipartite graph?

Hardness and approximation. By the general results on vertex deletion problems for hereditary graph properties (Section 2.1), Vertex Bipartization is NP-hard [Lewis and Yannakakis 1980] and MaxSNP-hard [Lund and Yannakakis 1993]. The best known approximation is by a factor of $O(\log n)$ [Garg et al. 1994].

Special cases and variants. Unlike Edge Bipartization, Vertex Bipartization remains NP-hard in planar graphs (even if the maximum degree is 4 [Choi et al. 1989]), although it can be approximated within $9/4$ [Goemans and Williamson 1998]. It is equivalent to the edge-deletion version in graphs with maximum degree 3 [Choi et al. 1989]; thus, for example, it can be solved in polynomial-time in planar graphs with maximum degree 3, but remains NP-hard in triangle-free graphs with maximum degree 3. Cornaz and Ridha Mahjoub [2007] give polyhedral results for the variant where the edges are weighted, that is, one wants to find an induced bipartite subgraph with maximum total edge weight.

Parameterized complexity. In a breakthrough paper, Reed et al. [2004] proved that Vertex Bipartization is solvable by iterative compression in $O(4^k \cdot kmn)$ time, where k is the number of vertices to delete. This is an important theoretical

result, since it implies fixed-parameter tractability for Vertex Bipartization with respect to k, which was posed as an open question more than five years earlier [Mahajan and Raman 1999]. Raman et al. [2007] gave an algorithm running in $O(1.62^n)$ time. From a general result by Khot and Raman [2002], it follows that the parametric dual of Vertex Bipartization, that is, the question whether a graph has a bipartite induced subgraph with at least k vertices, is W[1]-hard.

Applications. An application for Vertex Bipartization is in register allocation for processors that, to save wiring, have their register set divided into two banks and require the two operands of an instruction to reside in different banks [Zhuang and Pande 2003]. Conflicts are modeled by a graph where vertices correspond to operands and edges connect operands that occur together in an instruction. A minimum graph bipartization set then yields the minimum size set of operands that have to be copied into both banks to be able to execute the code.

Another application originates from computational biology. To determine gene sequences, for technical reasons the DNA is first broken into small fragments (*shotgun sequencing*), from which the original sequence is reconstructed by computer. This is complicated by the fact that each gene occurs twice in the human genome. The two copies are mostly identical, but differ at certain sites (so-called SNPs). Given a set of gene fragments, the problem of assigning the fragments to one of these two copies in a consistent manner while dismissing the least number of SNPs as erroneous is called the Minimum Site Removal problem [Panconesi and Sozio 2004, Zhang et al. 2006]. The Minimum Site Removal problem can be solved using Vertex Bipartization algorithms. We evaluate algorithms in this setting with synthetic data in Section 4.6.3.1.

Further applications appear in VLSI design [Choi et al. 1989, Kahng et al. 2001], linear programming [Gülpinar et al. 2004], computational biology [Rizzi et al. 2002], and RFID (radio-frequency identification) reader networks [Deolalikar et al. 2006].

Experimental results. Fouilhoux and Ridha Mahjoub [2006] give polyhedral method based results for Vertex Bipartization and present computational results based on branch & bound.

2.4 Minimum-Weight Path

Minimum-Weight Path is the central problem in Chapter 5, which deals with the color-coding technique. Given an edge-weighted graph and an integer k,

the Minimum-Weight Path asks for a simple (non-crossing) path of k vertices with minimum weight. It is thus a generalization of the well-known NP-hard Longest Path problem. One motivation to study Minimum-Weight Path comes from the investigation of protein interaction networks. Proteins participate in almost every process within the cell, in particular in signaling mechanisms. Understanding protein interactions is central to understanding the workings of the cell and can give guidance in developing drugs. A protein interaction network is a graph that models the way proteins interact: a vertex corresponds to a protein, and an edge is present if the two corresponding proteins are known to interact. Since the experimental methods used to determine this are unreliable, edges are usually annotated with a real number between 0 and 1 that can be interpreted as the probability that these proteins do actually interact [Suthram et al. 2006].

Large, high-quality protein interaction networks have only recently become available [Bader et al. 2003, von Mering et al. 2002]. A *Human Interactome Project* [Ideker and Valencia 2006], analogous to the Human Genome Project, has been initiated with the aim to produce a comprehensive human protein interaction network. Various approaches have been proposed to data-mine these networks for biologically meaningful substructures. A special role—with respect to both biological meaning as well as algorithmic tractability—is played by the most simple structures, namely *linear pathways*; in graph terms, these are simple paths, that is, paths where no vertex can occur more than once. Linear pathways are easy to understand and analyze and can serve as a seed structure for experimental investigation of more complex mechanisms [Ideker et al. 2001].

Initiated by Steffen et al. [2002], there have been efforts to design algorithms for the automated discovery of linear pathways in protein interaction networks. Scott et al. [2006] demonstrated that *high-scoring simple paths* in a protein interaction network are plausible candidates for linear signal transduction pathways, where *high-scoring* means that the product of interaction probabilities p is maximized. For easier handling, we will work with the *weight* $w(e) := -\log p(e)$ of the edges, such that the goal is to minimize the sum of weights for the edges of a path. To make the finding of signaling pathway candidates algorithmically feasible and biologically meaningful, the number of vertices that these paths contain is restricted by some reasonably small integer k. Formally stated, the NP-hard problem that needs to be solved in order to find minimum-weight simple paths thus is the following:

Minimum-Weight Path
Instance: An undirected graph $G = (V, E)$ with edges weighted by $w : E \to \mathbb{Q}_+$ and an integer $k > 0$.
Task: Find a simple path $v_1, \ldots, v_k$ of k vertices with minimum weight, that is, that minimizes $\sum_{i=1}^{k-1} w(\{v_i, v_{i+1}\})$.

Hardness and approximation. Minimum-Weight Path is the weighted generalization of Longest Path, that is, the question whether a graph contains a simple path of length k. Longest Path is a classic NP-hard problem. It is NP-hard because for $k = n$, it is equivalent to Hamiltonian Path. Moreover, even finding a constant factor approximation is NP-hard [Karger et al. 1997]. The best known approximations are by Gabow [2007] and by Feder and Motwani [2005]. They find in a graph with longest path length k in polynomial time a path of length $\exp(\sqrt{\log k / \log\log k})$ and $k^{1/(\log(n/k)+\log\log n)}$, respectively.

Parameterized and exact algorithms. Many algorithms for Longest Path, in particular those using dynamic programming, can be adapted for Minimum-Weight Path. The best known exact (not parameterized) algorithm is a dynamic programming based approach due to Bellman [1962] and also Held and Karp [1962] which runs in $O(n^2 2^n)$. Plehn and Voigt [1990] gave an algorithm running in $O((k^{O(k)} n^{\omega+1})$ time, where ω is the treewidth of the graph.

Monien [1985] gave the first fixed-parameter algorithm with a running time of $O(k!nm)$. Bodlaender [1993] gave an algorithm running in $O(2^k k! n)$ time using dynamic programming. Introducing the color-coding method, Alon et al. [1995] presented an algorithm solving Minimum-Weight Path in $O(5.44^k m)$ time with high probability. The method can be derandomized; the best known derandomization by Chen et al. [2007b] yields an $O(17.4^k m)$ time algorithm. In Section 5.3.1, we show how to speed up the randomized version to achieve an $O(4.32^k m)$ time bound.

Recently, two groups [Kneis et al. 2006, Chen et al. 2007b] independently developed a randomized algorithm inspired by color-coding based on a technique termed "divide-and-color" and running in $O(4^k k^{3.42} m)$ time. This technique can be derandomized, yielding an $O(4^{k+o(k)} m)$ bound. Thus, with 4^k compared to 4.32^k, currently the divide-and-color method has a better exponential bound of the running time. However, a disadvantage of this technique is that 4^k is also a lower bound on the running time, in the sense that it seems hard to exploit the particular structure of instances (e. g., sparseness) to get a lower running time in practice. In contrast, the running times we observed for color-coding with the improvements from Section 5.3 typically remained much below the worst-case bound, often by several orders of magnitude. Therefore, implement-

ing divide-and-color unmodified seems unlikely to yield faster algorithms in practice. In fact, Lu et al. [2007] reported on their implementation: "[...] we did not observe significant improvements in actual running time when compared with the previous approaches" (referring to Steffen et al. [2002] and Scott et al. [2006]).

Since Minimum-Weight Path is MaxSNP-hard, an exact $2^{o(k)}n^{O(1)}$ time algorithm would imply that 3-SAT with n variables, and many other problems, can be solved in $2^{o(n)}$ time [Cai and Juedes 2003], a result that is considered very hard to achieve, if possible at all.

A "hybrid algorithm" was given by Vassilevska et al. [2006], which for any (constructible) function $l(n) \in o(n)$ always either produces a longest path in $O(m + n2^{l(n)}l(n)!)$ time, or an $l(n)$ node path in linear time.

Further application. Besides the described application for finding signaling pathways, Minimum-Weight Path has found different applications. Deshpande et al. [2007] are concerned with the task of automatically generating headlines for texts. The assumption is that no semantic information is available; therefore, the headline is generated by selecting and ordering words from the input text. This is modeled by a vertex- and edge-weighted graph, where vertices are words and edges represent pairwise ordering preferences. The desired solution is a maximum-weight acyclic path of a prespecified length. Acyclicity is important, because no words should be repeated in the headline. By converting vertex weights to edge weights, this can easily be reduced to Minimum-Weight Path.

Chapter 3

Data reduction

Data reduction, also known as polynomial-time preprocessing, is a classic approach for dealing with NP-hard combinatorial optimization problems (see Guo and Niedermeier [2007] for a recent survey). The idea is to remove redundant parts of the input, thereby obtaining a hard "core" of the instance. Costly algorithms need then only be applied to this core. Consider for example the Vertex Cover problem, which asks for a minimum number of vertices to delete to get rid of all edges. In an instance, we can clearly omit all degree-0 vertices. Further, if we encounter a degree-1 vertex v, we can safely delete its neighbor w, since either v or w need to be deleted to cover the edge $\{v, w\}$, and taking w will cover potentially more edges than taking v.

More formally, a (*data*) *reduction rule* replaces in polynomial time a given problem instance I by an instance I′ with $|I'| < |I|$ such that I has a solution iff I′ has a solution. An instance to which none of a given set of reduction rules applies is called *reduced* with respect to these rules.

Data reduction has provided a number of success stories. For example, Bixby [2002] mentioned a large linear program that can be solved in half an hour when data reduction is employed, but is "far from even being feasible" for the unreduced instance even after hours of computation. Another account of the power of data reduction was given by Weihe [1998], where in the context of the European railroad network two simple data reduction rules allow an NP-hard problem to be solved in mere minutes for a graph consisting of more than 160 000 vertices and 1 600 000 edges.

Some data reductions rules are very simple and readily discovered by everybody tackling a problem. Others are hidden gems that require deeper digging and insight into a problem's combinatorial structure. Once an effective (and efficient) reduction rule has been found, however, it is useful in virtually any

problem solving context, whether it be heuristic, approximative, or exact.

Data reduction for hard problems was usually seen as a heuristic task, because in the classic one-dimensional complexity analysis, nothing can be proved about the quality of data reduction; this is because even the smallest provable data reduction would, by repetition, imply polynomial-time solvability of the instance and thus $P = NP$. Parameterized complexity, however, provides a useful notion of the power of data reduction with the concept of a *problem kernel*.

Definition 3.1. *Let* L *be a parameterized problem. A* reduction to a problem kernel *or* kernelization *is a transformation of an instance* (x, k) *to an instance* (x', k')*, such that*

- $(x, k) \in L \iff (x', k') \in L$,
- $|x'| \leqslant g(k)$ *for some arbitrary computable function* g *depending only on* k,
- $k' \leqslant k$, *and*
- *the transformation runs in polynomial time.*

In other words, a kernelization is a data reduction rule that creates an instance whose size depends only on the parameter k, and not on the original input size n anymore. As an example, consider again the Vertex Cover problem, this time in a parameterized context, where we are also given the maximum number of deletions as parameter k. If now there is a vertex of degree at least $k + 1$, then we can immediately delete it, because otherwise, we would have to delete all of its $k + 1$ neighbors, which is already too many. If we use this data reduction rule exhaustively, then no vertex in the remaining graph has a degree higher than k, meaning that choosing a vertex into the cover can cause at most k edges to become covered. Since the solution set may be no larger than k, the remaining graph can have at most k^2 edges if it is to have a solution. By the rules dealing with degree-0 and degree-1 vertices, every vertex has degree at least two, which implies that the remaining graph can contain at most k^2 vertices. Thus, we have found an $O(k^2)$-size kernel for Vertex Cover.

The notion of kernels opens the door to a fruitful dialog between practitioners and theoreticians: kernelizations can explain, and prove, why rules work so well in practice; and the quest for kernelizations can lead to new and powerful data reduction rules based on deep structural insights. Note that even if the parameter is not small, reduction rules can still be useful, since they run in polynomial time.

The connection between fixed-parameter tractability and existence of a problem kernel is in fact very immediate: they are the same.

Theorem 3.1 (Cai et al. [1997])**.** *Every problem that is fixed-parameter tractable is kernelizable and vice versa.* □

Unfortunately, the practical use of this theorem is limited: the running time of a fixed-parameter algorithm directly obtained from a kernelization is usually not practical; and, in the other direction, the theorem does not constructively provide us with a data reduction for a fixed-parameter tractable problem. Hence, the main use of Theorem 3.1 is to establish the fixed-parameter tractability or amenability to kernelization of a problem—or to show that we need not search any further (e. g., if a problem is known to be fixed-parameter intractable, we do not need to look for a kernelization).

One success story for kernelization is Dominating Set, the task of finding a minimum subset D of vertices in a graph such that each vertex is in D or has at least one neighbor in D. For the case that the input graph is planar, Alber et al. [2004] gave a linear-size problem kernel, where the parameter is the size of a dominating set D. Alber et al. [2006] showed that the corresponding data reduction rules work well on real-world instances (including non-planar ones), in many cases reducing 99 % or even all of the vertices of the input graph. As another example, Abu-Khzam et al. [2004a] and Abu-Khzam et al. [2007] studied various kernelization schemes for Vertex Cover and demonstrated large savings for data from computational biology.

The use of data reduction techniques is not restricted to a preprocessing phase; there is empirical as well as theoretical evidence that interleaving data reduction techniques with the main problem solving algorithm can yield significant speedups [Niedermeier and Rossmanith 2000]. We employ interleaving for Clique Cover in Section 3.1.2.

3.1 Data reduction for Clique Cover

The Clique Cover problem was introduced and motivated in Section 2.2. We recall its definition:

Clique Cover
Instance: An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Is there a set of at most k cliques in G such that each edge in E has both its endpoints in at least one of the selected cliques?

First, as our main algorithmic contribution, we introduce and analyze data reduction techniques for Clique Cover. As a side effect, we provide a problem kernel for Clique Cover, showing—somewhat surprisingly—that the problem is fixed-parameter tractable with respect to the parameter k. We continue with describing an exact algorithm based on a search tree. For our experimental

investigations, we combined our data reduction rules with the search tree, clearly outperforming heuristic algorithms [Kellerman 1973, Kou et al. 1978] in several ways. For instance, we can solve real-world instances from a statistical application—so far solved heuristically [Piepho 2004]—optimally without time loss. This indicates that for a significant fraction of real-world instances our exact approach is clearly to be preferred to a heuristic approach that is without guaranteed solution quality. We also experimented with random graphs of different densities, showing that our exact approach works extremely well for sparse graphs. In addition, our empirical results reveal that for dense graphs a data reduction rule that was designed for showing the problem kernel is often useful. In particular, this gives strong empirical support for further theoretical studies in the direction of improved fixed-parameter tractability results for Clique Cover, nicely demonstrating a fruitful interchange between applied and theoretical algorithmic research.

We formulate reduction rules for a generalized version of Clique Cover in which already some edges may be marked as "covered". Then, the question is to find a clique cover of size k that covers all uncovered edges. Clearly, Clique Cover is the special case of this annotated version where no edge is marked as covered.

We start by describing an initialization routine that sets up auxiliary data structures *once* at the beginning of the algorithm such that the *many* applications of Rule 3.2 (defined below) become cheaper in terms of running time. Moreover, the data structures initialized here are also used by the exact algorithm proposed in Section 3.1.2. From the reduction rules below, only Rule 3.1 updates these auxiliary data structures.

Initialization. We inspect every edge $\{u, v\}$ of the original graph. We use two auxiliary variables: We compute a set $N_{\{u,v\}}$ of u and v's common neighbors, and we determine whether the vertices in $N_{\{u,v\}}$ induce a clique. More precisely, we compute a positive integer $c_{\{u,v\}}$, which denotes the number of edges interconnecting the vertices of $N_{\{u,v\}}$.

Lemma 3.1. *The proposed initialization can be done in* $O(m^2)$ *time.*

Proof. For an edge $\{u, v\}$, we compute $N_{\{u,v\}}$ in $O(n)$ time. Computing $c_{\{u,v\}}$ can be done in $O(m)$ time. Doing this for all edges requires $O(m^2)$ time in total. □

3.1.1 Data reduction rules

We start the presentation of data reduction rules with a trivial rule removing isolated elements.

Reduction Rule 3.1. *Remove isolated vertices and vertices that are only incident to covered edges.*

Lemma 3.2. *Every application of Rule 3.1 including the update of auxiliary variables can be executed in* $O(nm)$ *time.*

Proof. The applicability of Rule 3.1 can be checked in $O(m+n)$ time. Notably, after removing a vertex, Rule 3.1 requires an update of the data structures provided by the initialization. For one removed vertex v, we have to adjust the sets $N_{\{u,w\}}$ and counters $c_{\{u,w\}}$ for all adjacent neighbors u and w of v. For an edge $\{u,w\}$ with $v \in N_{\{u,w\}}$, the affected sets and counters can be updated in $O(n)$ time by removing vertex v from $N_{\{u,w\}}$ and decreasing $c_{\{u,w\}}$ by $|N(v) \cap N_{\{u,w\}}|$. For m edges, the asymptotic running time of Rule 3.1 amounts to $O(nm)$. □

The next reduction rule is concerned with maximal cliques. Note that we can safely assume that an optimal solution consists of maximal cliques only, since a non-maximal clique in a solution can always be replaced by a maximal clique it is contained in. The following rule identifies maximal cliques which have to be part of *every* optimal solution.

Reduction Rule 3.2. *If an uncovered edge* $\{u,v\}$ *is contained in exactly one maximal clique* C*, that is, if the common neighbors of* u *and* v *induce a clique, then add* C *to the solution, mark its edges as covered, and decrease* k *by one.*

Lemma 3.3. *Rule 3.2 is correct. Every application of Rule 3.2 can be executed in* $O(m)$ *time.*

Proof. The rule is correct: Edge $\{u,v\}$ has to be covered and, as mentioned above, we can assume without loss of generality that it is covered by a maximal clique. If there is exactly one maximal clique C covering $\{u,v\}$, then C has to be part of every optimal solution.

Using the proposed initialization, we can apply Rule 3.2 in $O(m)$ time: For one edge, by looking up $N_{\{u,v\}}$ and $c_{\{u,v\}}$, we can determine in constant time whether the rule is applicable. Scanning through all edges and invoking the rule as soon as we find an edge for which the rule is applicable can be done in $O(m)$ time. □

Rules 3.1 and 3.2 remove all degree-1 and degree-2 vertices from the instance. Further, they imply that an isolated clique is deleted: Its edges belong to exactly one maximal clique; the clique, if it contains more than one vertex, is added to the solution by Rule 3.2 and its vertices are cleaned up by Rule 3.1.

In the following, we present two related Rules 3.3 and 3.4. Rule 3.3 is subsumed by Rule 3.4. Nevertheless, we choose to present the two rules separately,

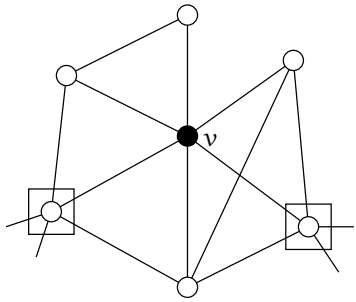


Figure 3.1: An illustration of the partition of the neighborhood of a vertex v. The two vertices with rectangles are exits, the other white ones are prisoners. Since both exits have a prisoner as neighbor, the prisoners dominate the exits.

since Rule 3.3 is easier to understand. Moreover, as will be shown in Theorem 3.2, already Rule 3.3 is sufficient to show a problem kernel for Clique Cover.

Reduction Rule 3.3. *If there is an edge $\{u, v\}$ whose endpoints have exactly the same closed neighborhood, that is, $N[u] = N[v]$, then mark all edges incident to u as covered. To reconstruct a solution for the unreduced instance, add u to every clique containing v.*

Comparing $N[u]$ and $N[v]$ for each edge $\{u, v\}$, we can in $O(nm)$ time search an edge for which Rule 3.3 is applicable and invoke the rule.

For formulating a generalization of Rule 3.3, we introduce additional terminology. For a vertex v, we partition the set of vertices that are connected by an uncovered edge to v into *prisoners* p with $N(p) \subseteq N[v]$ and *exits* x with $N(x) \setminus N[v] \neq \emptyset$. We say that the prisoners *dominate* the exits if every exit x has an adjacent prisoner. An illustration of the concept of prisoners and exits is given in Figure 3.1. The concept of prisoners and exits (and, in addition, "gates") was introduced for data reduction rules designed for the Dominating Set problem [Alber et al. 2004].

Reduction Rule 3.4. *Consider a vertex v which has at least one prisoner. If each prisoner is connected to at least one vertex other than v via an uncovered edge (note that this is automatically given when the instance is reduced with respect to Rules 3.1 and 3.2), and the prisoners dominate the exits, then delete v. To reconstruct a solution for the unreduced instance, add v to every clique containing a prisoner of v.*

Observe that a vertex v is always a prisoner of a vertex u if $u \neq v$ and $N[u] = N[v]$ (and vice versa). In particular, this prisoner v dominates all exits in $N[u]$. Thus, Rule 3.3 is subsumed by Rule 3.4.

Lemma 3.4. *Rule 3.4 is correct. Every application of Rule 3.4 can be executed in* $O(n^3)$ *time.*

Proof. Let G' be the input graph G after one application of Rule 3.4. By definition, every neighbor of v's prisoners in G is also a neighbor of v itself. If a prisoner of v participates in a clique C in G', then $C \cup \{v\}$ is also a clique in G. Therefore, it is correct to add v to every clique containing a prisoner in G'. Next, we show that this also covers all edges incident to v. We consider separately edges connecting v to prisoners and edges connecting v to exits. Regarding an edge $\{v, p\}$ to a prisoner p, vertex p has to be part of a clique C of the solution for G'. Therefore, the edge $\{v, p\}$ is covered by $C \cup \{v\}$ in the solution for the unreduced instance. Regarding an edge $\{v, x\}$ to an exit x, the exit x is dominated by a prisoner p and therefore x has to be part of a clique C with p, since the edge $\{x, p\}$ needs to be covered. Hence, the edge $\{v, x\}$ is covered by $C \cup \{v\}$ in the solution for the unreduced instance.

For executing the rule, we inspect every vertex v to test whether the rule is applicable. To this end, we inspect every neighbor u of v. In $O(n)$ time, we determine whether u is an exit or a prisoner. Having identified all prisoners, we can for each exit u determine in $O(n)$ time whether u is dominated by a prisoner. □

One easily observes that there are cases where Rule 3.4 applies but Rule 3.3 does not; however, we could not make use of this fact to improve the theoretical analysis which follows in Theorem 3.2.

Lemma 3.5. *Using Rules 3.1, 3.2, and 3.4, in* $O(n^4)$ *time one can generate a reduced instance where none of these rules applies any further.*

Proof. First, we apply Rule 3.1 to remove the isolated vertices. Then, we repeat the following operation until neither Rule 3.2 nor Rule 3.4 is applicable: Apply one of Rules 3.2 and 3.4, if possible, and, after each application, use Rule 3.1 to remove the vertices only adjacent to covered edges. Since each application of Rule 3.2 or Rule 3.4 results in at least one vertex only incident to covered edges and each application of Rule 3.1 removes at least one vertex from the graph, the above operation is repeated at most n times. From Lemmas 3.2, 3.3, and 3.4, we can conclude that the total running time for the application of the three rules amounts to $O(n \cdot (nm + m + n^3))$. Together with the $O(m^2)$ running time of the initialization shown in Lemma 3.1, the claimed running time follows. □

From a theoretical viewpoint, the main result of this section is a problem kernel with respect to the parameter k for Clique Cover (as we found later, the central underlying observation was already made by Gyárfás [1990]):

Theorem 3.2. *A* Clique Cover *instance reduced with respect to Rules 3.1 and 3.3 contains at most* 2^k *vertices or, otherwise, has no solution.*

Proof. Consider any graph $G = (V, E)$ with more than 2^k vertices that has a clique cover $C_1, \ldots, C_k$ of size k. We assign to each vertex $v \in V$ a binary vector b_v of length k where bit i, $1 \leqslant i \leqslant k$, is set to 1 iff v is contained in clique C_i. Since there are only 2^k possible vectors, there must be $u, v \in V$ with $u \neq v$ but $b_u = b_v$. If b_u and b_v are the all-zero vector, Rule 3.1 applies; otherwise, u and v are contained in the same cliques. This means that u and v are connected and share the same neighborhood, and thus Rule 3.3 applies. □

Corollary 3.1. Clique Cover *is fixed-parameter tractable with respect to the parameter* k.

Proof. By Theorem 3.2, a Clique Cover instance that is reduced with respect to Rules 3.1 and 3.3 has at most 2^k vertices. It takes $O(n^4)$ time to generate a reduced instance (Lemma 3.5). Thus, we have found a kernel (Definition 3.1) for Clique Cover, which by Theorem 3.1 implies that Clique Cover is fixed-parameter tractable. □

The result of Corollary 3.1 might be surprising when noting that many graph problems that involve cliques turn out to be hard in the parameterized sense. For example, the NP-hard Clique problem, which asks for a clique of size k in a graph, is known to be W[1]-hard with respect to k [Downey and Fellows 1999], that is, we have a clear indication that this problem is not fixed-parameter tractable with respect to the parameter "clique size". Another example even more closely related to Clique Cover is given by the NP-hard Clique Partition problem, which is also hard in the parameterized sense. Herein, we ask, given an undirected graph and $k \geqslant 0$, for a set of k cliques covering all *vertices* of the input graph (in contrast to covering all *edges* as in Clique Cover). Clique Partition is NP-hard already for $k = 3$ [Garey and Johnson 1979]. This can be seen by observing that Clique Partition is equivalent to Coloring on the complement graph; the number of colors required for Coloring corresponds to the number of cliques required for Clique Partition. It is well-known that Coloring is already NP-hard for three colors [Garey and Johnson 1979]. It follows that there is no hope for obtaining fixed-parameter tractability for Clique Partition with respect to parameter k, unless $P = NP$. In contrast, Clique Cover is shown fixed-parameter tractable in Corollary 3.1; in both cases the number of required cliques is the considered parameter.

We conclude this section by mentioning an additional rule which does not yield an improvement on the problem kernel, but gives an empirical speedup for certain instances (Section 3.1.3). It differs from the above rules in that it does

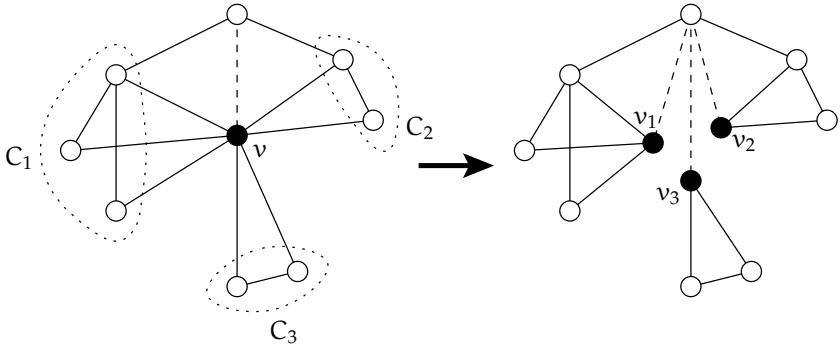


Figure 3.2: Example for Rule 3.5. *Dashed* edges are covered.

not make the instance smaller (in fact it makes it larger), but intuitively seems to facilitate application of other data reduction rules since it decomposes the input into separate components. Figure 3.2 gives an example.

Reduction Rule 3.5. *Consider the set* N *of vertices that are connected to a vertex* v *via uncovered edges. Let* $C_1, \ldots, C_l$ *be the connected components induced by* N*. If there is more than one connected component (that is,* $l > 1$*), then replace* v *by* l *new vertices* $v_1, \ldots, v_l$*, and connect for each* $1 \leqslant i \leqslant l$ *the vertex* v_i *to all vertices in* C_i*. In addition, all of* $v_1, \ldots, v_l$ *are connected by covered edges to all vertices that are connected to* v *with a covered edge.*

Lemma 3.6. *Rule 3.5 is correct and can be executed in* $O(nm)$ *time.*

Proof. Let G be a graph and G' the result of applying Rule 3.5 to some vertex v. Consider a clique that contains v in a clique cover for G. Besides v, it can contain only vertices from a single component C_i and vertices that are connected to v by a covered edge. Therefore, we can transform it to a clique in G' by replacing v by v_i. This transformation will cover all edges in G', since for any edge $\{v_i, w\}$ in G' the edge $\{v, w\}$ will be covered by some clique in a clique cover of G. The converse direction works analogously. The transformation can be done in $O(m)$ time for each vertex v. □

3.1.2 Search tree algorithm

Search trees are a popular means of exactly solving hard problems; we gave an introduction and some examples in Section 1.3.1. The basic method is to identify for a given instance a small set of simplified instances such that the given instance has a solution iff at least one of the simplified instances has one.

The algorithm then branches recursively into each of these cases until a stop criterion is met.

The search tree algorithm presented here for Clique Cover makes use of the fact that without loss of generality, we can assume that each clique in the cover is maximal. We choose an uncovered edge, enumerate all maximal cliques this edge is part of, and then branch according to the clique we add to the clique cover. The recursion stops as soon as a solution is found or k cliques have been chosen without finding a solution. The algorithm is presented in pseudo-code in Figure 3.3.

At first glance, this branching seems to be impractical, since the number of maximal cliques in a graph can be exponentially large, resulting in a double-exponentially large search tree; in particular, we do not get a fixed-parameter time bound with respect to the parameter k, unless we employ the kernelization from Theorem 3.2 first. However, in practice it turns out that there are usually only a few branching cases. We try to give some intuition for this: Sensible inputs for clustering problems are expected to exhibit transitivity in the sense that if $\{a, b\}$ and $\{b, c\}$ are edges, then probably also $\{a, c\}$ is an edge (that is, its *clustering coefficient* is high). Graphs with many maximal cliques, however, are closely related to the presence of certain complete multipartite graphs [Prisner 1995]; these multipartite graphs are very nontransitive.

Regarding the choice of the edge to branch on, we would, ideally, like to branch on the edge that is contained in the least number of maximal cliques. However, this calculation would be costly. Therefore, we make use of the data structures set up for an efficient incremental application of Rule 3.2. The initialization described in Section 3.1.1 provides a set $N_{\{i,j\}}$ containing the common neighborhood of edge $\{i, j\}$ and a counter $c_{\{i,j\}}$ containing the number of edges in the common neighborhood of its endpoints. Therefore, $\binom{|N_{\{i,j\}}|}{2} - c_{\{i,j\}}$ is the number of edges missing in the common neighborhood of edge $\{i, j\}$ as compared to a clique (called *score*). For branching, we choose the edge with the lowest score. If the score is 0, then the edge is contained in only one maximal clique (and thus will be marked as covered by Rule 3.2). If the score is 1, the edge is contained in exactly two maximal cliques. Generalizing this, it is plausible to assume that an edge is contained in few maximal cliques if its score is low.

Having chosen the edge to branch on, we determine the set of maximal cliques the edge is contained in using a variant of the classic Bron–Kerbosch algorithm [Bron and Kerbosch 1973] described by Koch [2001]. While the original Bron–Kerbosch algorithm does not exhibit the desired output sensitivity (it runs in exponential time even for a single maximal clique), the variant by Koch [2001] turns out to be fast enough for our purposes.

As suggested e. g. by Niedermeier and Rossmanith [2000], we interleave the

Input: Graph $G = (V, E)$.
Output: A minimum cardinality clique cover for G.

```
1   k ← 0; X ← nil
2   while X = nil:
3       X ← branch(G, k, ∅)
4       k ← k + 1
5   return X
6   function branch(G, k, X):
7       if X covers G: return X
8       reduce(G, k)
9       if k < 0: return nil
10      choose {i, j} such that (|N_{i,j}| choose 2) − c_{i,j} is minimal
11      for each maximal clique C in N[i] ∩ N[j]:
12          X' ← branch(G, k − 1, X ∪ {C})
13          if X' ≠ nil: return X'
14      return nil
```

Figure 3.3: Exact algorithm for Clique Cover, where *reduce* applies Rules 3.1 to 3.5

search tree with the data reduction, that is, we apply data reduction after each modification to the graph due to branching. We use the branching routine within an iterative deepening framework, that is, we impose a maximum search depth k and increase this limit by one when no solution is found.

3.1.3 Implementation and experiments

We implemented the search tree algorithm from Section 3.1.2 and the data reduction rules from Section 3.1.1. The program is written in the Objective Caml programming language [Leroy et al. 1996] and consists of about 1400 lines of code. The source code is free software and available from http://theinf1.informatik.uni-jena.de/ecc/. Graphs are implemented using a purely functional representation based on Patricia trees [Okasaki and Gill 1998]. This allows to (conceptually) modify the graph in the course of the algorithm without having to worry about how to restore it when returning from the recursion. Moreover, it allows for quick intersection operations on neighbor sets, as required for some reduction rules. The cache data structure N described in Section 3.1.1 is implemented using a priority search queue also based on Patricia trees.

Table 3.1: Clique cover sizes for five real-world Clique Cover instances, where "Heuristic" is the heuristic by Kellerman [1973] with the postprocessing by Kou et al. [1978].

			Clique cover size	
	n	m	Heuristic	Optimal
A	13	55	4	4
B	17	86	6	5
C	124	4847	50	49
D	121	4706	48	48
E	97	3559	34	31

We tested our implementation on various inputs on an AMD Athlon 64 3700+ with 2.2 GHz, 1 MB cache, and 1 GB main memory, running under the Debian GNU/Linux 3.1 operating system.

Real-world data. We first tested the implementation on five "real-world" instances from an application in graphical statistics [Piepho 2004] (see Table 3.1). Currently, heuristics like that of Kou et al. [1978] are used to solve the problem in practice [Piepho 2004, Gramm et al. 2007b]. With our implementation of the heuristic by Kellerman [1973]—which gives no guarantee for the quality of the solution—, the running time is negligible for these instances (< 0.1 s). Our implementation based on the search tree with data reduction could solve all instances to optimality within less than one second. In all cases, no branching was required: Rules 3.1 and 3.2 already completely reduced the instances. We observe that the heuristic produces reasonably good results for these cases; previously nothing was known about its solution quality. In summary, the application of our algorithm in this area seems quite attractive, since we can provide provably optimal results within acceptable running time bounds.

Random graphs. Next, we tested the implementation of the exact algorithm on random graphs, that is, graphs where every possible edge is present with a fixed probability ($G_{n,p}$ model). It is known that with high probability a random graph has a large clique cover of size $\Theta(n^2/\log^2 n)$ [Frieze and Reed 1995]. Therefore, relying on branching and a not too large search tree is unlikely to succeed, and data reduction rules are crucial. The results are presented in Figure 3.4. In the following, the "size" of an instance means the number of vertices. We examine three trials: Sparse graphs with $m \approx n \ln n$, graphs with edge probability 0.1,

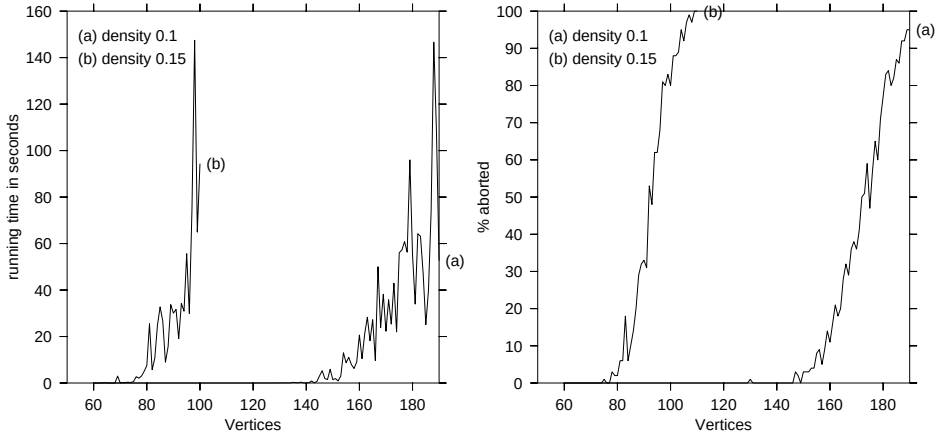


Figure 3.4: Running time for random graphs. Runs were aborted after 10 minutes. Left: average running time of successful runs; right: percentage of aborted runs.

and graphs with edge probability 0.15. For the denser graphs outliers occur: for example for graphs of size 79 and edge probability 0.15, all of 20 instances could be solved within 10 seconds but one, which took 16 minutes. In contrast, sparse graphs could be solved uniformly very quickly: Instances of size 5 000 could still be solved within 80 seconds and instances of size 10 000 within 7 minutes, with a standard deviation for the running time of less than 2 %. Very little branching is required for sparse instances, with most being solved by data reduction alone, and the largest search tree observed in the experiments having 528 nodes. Thus, our approach is very promising for sparse instances up to moderate size, while for denser instances probably a fallback to heuristic algorithms is required to compensate for the outliers.

The presence of extreme outliers for some parameters makes it difficult to get a clear picture based only on combining statistics such as averages. Therefore, we show measurements for several concrete instances in Table 3.2. For edge probability 0.1 and 0.15, respectively, we select an instance that takes very long, and additionally present two arbitrary instances with similar parameters. For sparse graphs, no such outliers occurred, so we show three arbitrary instances of similar size.

The reason for the outliers are the usually but not always effective data reduction rules. In all instances, we observed an initial reduction phase with many applications of reduction rules. Most of the random graphs with 75 vertices and edge density 0.15 are almost entirely processed by the reduction

Table 3.2: Statistics for selected random Clique Cover instances. Here, p is the edge probability, running time is measured in seconds, $|C|$ is the size of the clique cover, "search tree" is the number of nodes in the search tree, and "Rule r" is the number of applications of Rule r. Rule 3.4 was not successfully applied.

	n	m	$\|C\|$	runtime	search tree	Rule 3.1	Rule 3.2	Rule 3.5
sparse	1000	6 954	6 180	1.00	1	1 000	6180	0
	1000	6 816	6 022	0.96	1	1 000	6022	0
	1000	6 861	6 107	0.96	1	1 000	6107	0
$p = 0.1$	156	1 230	653	20.58	254584	845 180	429 193	39 042
	156	1 194	644	0.02	27	194	664	1
	156	1 226	646	3285.43	21 889 796	112 527 915	63 709 259	5 313 473
$p = 0.15$	85	524	273	0.01	1	85	273	0
	85	545	272	15.88	132056	705743	382767	25032
	85	560	265	1505.94	8725027	47947699	27087827	3295196

rules: Out of 50 examined instances we observe search trees with depth more than 3 for 8 instances and 24 instances are completely reduced without branching. However, in rare cases we do encounter instances with an "unreducible core" to which no reduction rule is applicable. Moreover, with an unreducible core it is only rarely the case that reduction rules become applicable after the next branching. Consequently, an unreducible core does cause a significant number of branchings in the search tree.

Synthetic data. Real instances are not completely random; in particular, in most sensible applications the clique cover is expected to be much smaller than that of a random graph. The fixed-parameter result also promises a better running time for instances where the clique cover is small. To examine this, we generated random intersection graphs using the $G_{n,m,p}$ model (see Behrisch and Taraz [2006] and references therein), where each of n vertices draws each of m features with probability p (note here m does not denote the number of edges as elsewhere). We can control the size of a clique cover by choosing m: the size of the clique cover must be m or slightly lower in case $C_x \subseteq C_y$ for two features x and y (see Section 3.1 for the notation). By choosing p, we can generate instances with a desired edge density.

We generated instances with 100 vertices and about 1500 edges (making for a density of 0.3), and varying number of planted cliques m. Figure 3.5 shows the resulting running times. In fact, these quite dense instances can be solved very quickly when the size of the clique cover is small. This makes our exact algorithm also attractive for the numerous applications where we can expect a

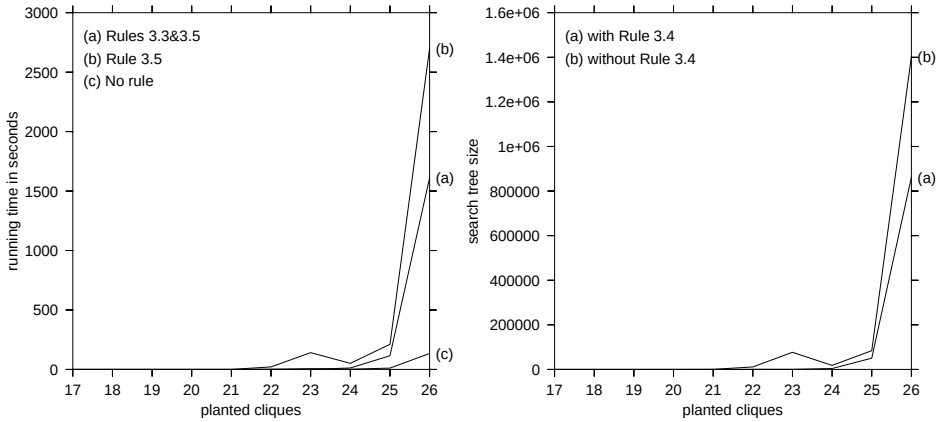


Figure 3.5: Running Time and search tree size for random intersection graphs of density 0.3 (average over 20 instances each)

small clique cover as solution—an observation which is in agreement with our fixed-parameter tractability result.

Effectiveness of Rules 3.4 and 3.5. The prisoner-exit-rule (Rule 3.4) and the neighborhood-splitting-rule (Rule 3.5) are comparably complicated; Rule 3.4 has been developed in context with searching for a problem kernel. Do they really gain any benefit in practice? To examine this question, we repeated the previous experiment with either Rule 3.4 or Rules 3.4 and 3.5 disabled (see Figure 3.5). Rule 3.5 does in fact increase the running time, simply because it is rarely ever applicable in these dense instances. Although Rule 3.4 increases also the running time as shown in the left-hand diagram of Figure 3.5, we recommend to apply it. This is justified by the right-hand side of Figure 3.5: The reduction of the search tree size indicates that—together with a more efficient implementation of Rule 3.4, the overall complexity can be (asymptotically) improved. Currently, the implementation does not take advantage of the fact that after branching, the instance is only minimally changed, but rather calculates everything from scratch; therefore, we believe that using incremental calculations, the running time of Rule 3.4 could be substantially reduced.

3.1.4 Outlook

In the experiments (Section 3.1.3), we found that there are often outliers with exceedingly high running times when compared to "similar" instances (Table 3.2).

It is an intriguing open question whether there are further data reduction rules that can cope with the remaining outliers. In parallel, this might also lead to a better upper bound on the problem kernel size and improved fixed-parameter tractability for Clique Cover.

An interesting problem variant is to change the optimization objective from "number of cliques" (#-optimal) to "sum of clique sizes" (Σ-optimal). This objective has been suggested for the equivalent Compact Letter Display problem [Gramm et al. 2007b] and plays a role in circuit synthesis [Khomenko 2007]; however, I am not aware of literature dealing with its complexity or exact algorithms. It is easy to find examples that are #-optimal, but not Σ-optimal; however, the converse does not seem to be true. Therefore, I conjecture that any clique cover that is Σ-optimal is also #-optimal. This would imply the NP-hardness of Σ-optimality. The fixed-parameter tractability of Σ-optimality with respect to the parameter "sum of clique sizes" Σ can be seen by the fact that if there are more than $O(\Sigma^2)$ edges, the instance is not solvable, thus a trivial kernel exists.

Another variant is Biclique Cover [Orlin 1977]: given a bipartite graph, find a minimum number of (not necessarily disjoint) complete bipartite subgraphs that cover all edges. Analogous to Theorem 3.2, Fleischner et al. [2007] derived a kernel of $O(2^k)$ vertices for Biclique Cover. Due to a number of applications (see e. g. Amilhastre et al. [1998] and references therein), it would be worthwhile to find data reductions and fixed-parameter algorithms for Biclique Cover better than those that result from this kernel.

3.2 Data reduction for Balanced Subgraph

In this section, we present data reduction rules for the Balanced Subgraph problem introduced in Section 2.3.2. We quickly recall the definition. Balanced Subgraph is defined on *signed graphs*, that is, graphs where every edge is annotated with $=$ or $\neq$. A signed graph is *balanced* if its vertices can be colored with two colors such that the relation at each edge holds with respect to the colors of its endpoints. The Balanced Subgraph problem is then defined as follows:

Balanced Subgraph
Instance: A signed graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k edge deletions into a balanced graph?

In this section, we assume that G is a *multigraph*, which can have multiple labeled edges between two vertices. This is useful for several applications and actually makes the reducions easier to formulate.

The data reduction is based on finding small separators and a novel gadget construction scheme. It unifies and generalizes a number of previously known data reductions [Wernicke 2003] and seems applicable to a wider range of graph problems where a coloring or a subset of the vertices is sought. Our implementation, which solves the irreducible instances by iterative compression (see Section 3.2.4) can solve biological real-world instances exactly for which previously only approximations [DasGupta et al. 2007] were known.

3.2.1 Data reduction scheme

Many data reduction rules have been developed in a problem-specific and ad-hoc way based on small fixed-size substructures. A typical example is the following one from Wernicke [2003] for Edge Bipartization, a special case of Balanced Subgraph (see Section 2.3.2).

Reduction Rule 3.6. *Let* $G = (V, E)$ *be an* Edge Bipartization *instance and let* $abcd$ *be an induced* C_4 *in* G*, where the two nonadjacent vertices* b *and* d *have degree* 2 *in* G*. Then delete* b*.*

This rule is correct because without loss of generality we never need to delete b or d, since deleting a or c destroys at least as many odd cycles, and further there is an odd cycle containing b iff there is an odd cycle containing d. In a similar way, considering structures of a few vertices, Wernicke [2003] presented several more data reduction rules. Another example is the "vertex folding" rule, which gets rid of degree-2 vertices in Vertex Cover instances [Chen et al. 2001].

When looking at the correctness proofs of these reduction rules, one notices that they are often, implicitly or explicitly, based on a separator (that is, a set of vertices whose deletion separates the graph into at least two connected components): we have a small number of vertices (e. g. b and d in Rule 3.6) that are separated by a small separator (e. g. a and c in Rule 3.6) from the rest of the graph. Our aim is to generalize this kind of data reduction rule.

The idea is to find a small separator S that cuts off a small component C from the rest of the graph. Then, we replace S and C by a smaller gadget that exhibits the same behavior. A similar method has been suggested by Polzin and Vahdati Daneshmand [2006] for the Steiner Tree problem. However, they do not employ gadgets and have no formal characterization of reducible cases.

Another similar method are *crown reduction* rules [Abu-Khzam et al. 2004a, Chor et al. 2004, Abu-Khzam et al. 2007, Prieto Rodríguez 2005]. Crown reductions also work by finding a separator S that cuts off a component C, and impose additional demands on S and C (for instance for Vertex Cover, C must be an independent set and there must be a matching between S and C that matches all vertices of S [Abu-Khzam et al. 2004a]). Crown reductions are then used

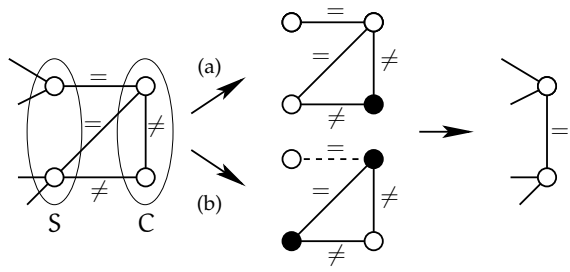


Figure 3.6: Example for Reduction Scheme 3.1

to prove kernels by showing that either a "crown" (reduction opportunity) can be found in polynomial time, or the instance is already kernelized. The main difference to our approach is that we do not assume any particular properties of S and C, except that they are small.

As is standard with separator-based methods (such as tree-decomposition based algorithms (see Section 1.3.3)), the behavior is examined by exhaustively enumerating possible states of the separator and finding exact solutions to the small component C. For Balanced Subgraph, the states are possible colorings of the vertices in the separator in an optimal solution. We now present the scheme more formally.

Reduction Scheme 3.1. *Let S be a separator and let C be a small component obtained by deleting S from the given graph G. Then, determine for each of the (up to symmetry)* $2^{|S|-1}$ *colorings of S the size of an optimal solution for the induced subgraph* $G[S \cup C]$ *and replace in G the subgraph* $G[S \cup C]$ *by a gadget that contains the vertices of S and possibly some new vertices.*

The above scheme leaves open some details. Before filling them in, let us show a simple example.

Example 3.1. *In Figure 3.6, the separator S cuts off the vertices in C from the rest of the graph. Up to symmetry, there are only two possibilities how the vertices in S can be colored: equal or unequal. If they are colored equal (a), the subgraph* $G[S \cup C]$ *is balanced without edge deletions. Otherwise (b), one edge deletion is required (dashed line). We can simulate this behavior with a single =-edge between the two vertices in S: it also incurs a cost of* 0 *when the two vertices of S are colored equal, and a cost of* 1 *otherwise. Therefore, we can replace the subgraph* $G[S \cup C]$ *by the gadget shown on the right.*

To fully describe the reduction scheme, four questions have to be answered:

(a) The instances $G[S \cup C]$ have some vertices (those of the separator) pre-colored. How to solve these already partly colored instances?

(b) There is a combinatorial explosion with the sizes of S and C affecting the running time. Therefore, how do we restrict the choices of S and C?

(c) How can we efficiently find useful (S, C)-combinations?

(d) If existing, how can we construct a gadget that is smaller than $G[S \cup C]$ and correctly "simulates" $G[S \cup C]$?

Regarding (a), we reduce the instance to an instance without pre-colored vertices, and then solve the instance recursively. For this, we merge all vertices pre-colored black into a single uncolored vertex and all vertices pre-colored white into a single uncolored vertex. Here, to *merge* two vertices v and w means to delete v and w and all incident edges, and add a new vertex x with edges from x to each vertex that was connected to v or w. We then add m edges labeled $\neq$ between the two new vertices. Any solution for this instance will then color the two vertices differently, and we can (possibly by flipping all colors) reconstruct a solution for the pre-colored instance.

Regarding (b), this can be simply done by imposing a fixed limit. In our implementation, we restrict the size of S to at most 4, mainly because of difficulties with the gadget construction. The size of C is (somewhat arbitrarily) restricted by 32; however, due to the structure of our instances, this limit did not play a role, because all components found were much smaller.

As to (c) and (d), we will answer these questions in the next two subsections.

3.2.2 Efficiently finding separators

To improve running time, we special-case the search for separators of size 0 (that is, the graph consists of more than one connected component) and separators of size 1 (that is, articulation points). They can be found in linear time using depth-first search [Gabow 2000]. For these cases, the gadget construction can be omitted: the 2-connected components[1] can be treated independently, and optimal colorings of two components can always be merged (possibly by flipping all colors in one component), since they overlap only in one vertex. Note that this phase in particular removes all degree-1 vertices.

Separators of size 2 can also be found in linear time [Hopcroft and Tarjan 1973]. However, we did not implement this algorithm, since it is quite complicated and error-prone to implement (several errors in the original publication have been pointed out [Gutwenger and Mutzel 2000]).

[1] A set of vertices is 2-connected if there are at least two vertex-disjoint paths between any pair of vertices from this set.

Separators of size k for small k can be found efficiently by flow techniques [Henzinger et al. 2000]. However, after some experiments we settled for the subsequently described heuristic instead, which is faster and produces no worse results in our tests. Let $N(X) := \{u \mid \{u, v\} \in E \wedge v \in X\} \setminus X$. For each vertex v, set $C := \{v\}$ and iteratively enlarge C by a vertex v' that minimizes the size of $S := N(C \cup \{v'\})$ until $|C| > 32$. The size of S can grow and shrink during this process; we record all combinations of S and C with $S \leqslant 4$.

To get a heuristic speedup, it is useful to first treat separators that are easy to deal with, but promise large reductions. Therefore, we sort the (S, C)-combinations primarily by increasing size of S and secondarily by decreasing size of C. In our experiments, the finding of separators in the above way altogether never took more than few seconds for graphs with up to about 700 vertices.

3.2.3 Gadget construction

The goal is to show how the subgraph $G[S \cup C]$ induced by the separator S and the small component C can be replaced by a smaller, "equivalent" subgraph (gadget). A simple case has already been described in Example 3.1. Now, we describe a general methodology, leading also to theoretically interesting problems that deserve further investigation.

Let us call a separator of size i simply i-*cut*. As mentioned before, it is easy to deal with 0- and 1-cuts. Hence, we focus on larger separators, thereby describing constructions delivering optimal gadgets in case of 2- and 3-cuts and a heuristic approach for 4-cuts. We also briefly discuss the mathematical and algorithmic challenges behind constructing gadgets for i-cuts for general i.

By an *optimal* gadget we refer to one with a minimum number of vertices (the alternative setting of minimizing the number of edges might be worth consideration as well). When speaking of an *equivalent* gadget which replaces the subgraph $G[S \cup C]$, we refer to a subgraph H with the following properties:

1. Gadget H contains all vertices from S and possibly more; in particular, S forms the "interface" where H is plugged in instead of $G[S \cup C]$.

2. The original graph G has a solution for Balanced Subgraph of size k iff the modified graph where H replaces $G[S \cup C]$ has a solution of size $k' \leqslant k$, where the difference between k' and k is determined by the gadget. Moreover, an optimal solution for G can be reconstructed from an optimal solution for the modified graph.

3.2.3.1 Gadget construction for 2-cuts

2-cuts generalize Example 3.1. Up to symmetry, there are only two colorings of the two separator vertices u and v. In each of these two cases, we compute recursively an optimal solution for $G[S \cup C]$, which can be done quickly, since only small C are considered.

Let n_e be the size of an optimal solution for $G[S \cup C]$ where u and v have the same color and let n_d be the size of an optimal solution where they have distinct colors. We perform the following gadget construction, where the gadget consists solely of vertices from S. If $n_e \geqslant n_d$, then remove C and all edges within S and add $n_e - n_d$ edges labeled $\neq$ between u and v. Otherwise, remove C and all edges within S and add $n_d - n_e$ edges labeled $=$ between u and v. Note that reducing 2-cuts in particular gets rid of all vertices of degree 2.

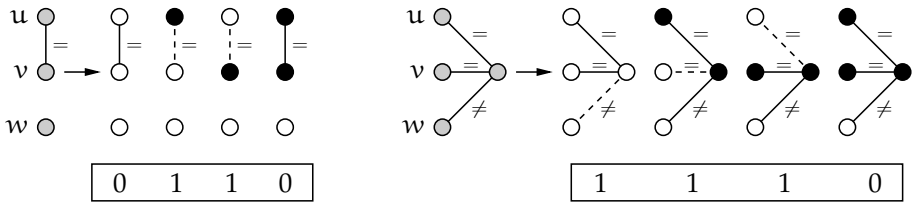
Lemma 3.7. *Let* G *be the original graph and let* G' *be the graph originating from* G *by performing the described gadget replacement. Then* G *has a solution of size* k *iff* G' *has a solution of size* $k - \min\{n_e, n_d\}$.

Proof. Consider first the case $n_e \geqslant n_d$. From a solution of size k for G, we can construct a solution of size $k - n_d$ for G' by using the same coloring restricted to the remaining vertices and deleting all inconsistent edges. If this solution colors u and v differently, we save n_d edges within $G[S \cup C]$; the $\neq$-edges do not incur any additional cost. If this solution colors u and v equally, we save n_e edges within $G[S \cup C]$, but need to delete all $n_e - n_d$ $\neq$-edges between u and v, also resulting in a solution of size $k - n_d$. In the same way, we can construct from a solution of size $k - n_d$ for G' a solution of size k for G. The case $n_e < n_d$ works in complete analogy. □

3.2.3.2 Gadget construction for 3-cuts

The basic approach is the same as for 2-cuts. The gadget construction, however, becomes more intricate. The idea is to construct the final gadget from *atomic gadgets*, which can be added independently until in total they have the desired effect. To characterize the effect of an atomic gadget, we introduce the concept of a *cost vector*. In the case of 3-cuts, up to symmetry, we have four possibilities to color the vertices from the separator S. For each case, we compute the cost of an optimal Balanced Subgraph solution of $G[S \cup C]$. For a fixed order of the colorings, these values build the cost vector of the form (c_1, c_2, c_3, c_4). The goal is then to find atomic gadgets such that their corresponding atomic cost vectors add up to the cost vector associated with $G[S \cup C]$.

We show that it is sufficient to consider atomic gadgets that, besides S, have at most one additional vertex. The first type of atomic gadgets are gadgets



(a) no extra vertex (b) one extra vertex

Figure 3.7: Examples for atomic gadgets for a size-3 separator $\{u, v, w\}$

exclusively made of vertices from S. More specifically, there are six possibilities to put exactly one edge, either labeled $=$ or $\neq$, between the three possible vertex pairings in S. Each of these possibilities yields an atomic gadget. Moreover, each of these atomic gadgets naturally one-to-one corresponds to a cost vector with 0/1-entries. For instance, let $\{u, v, w\}$ form the separator. Then, the atomic gadget with an $=$-edge between u and v corresponds to the cost vector $(0, 1, 1, 0)$ (see Figure 3.7a): If u and v have the same color (once white, once black), then the insertion of the $=$-edge does not cause an inconsistency. Thus, we have an additional solution cost of 0, justifying the two zero-entries in the cost vector. If u and v have different colors, then the insertion of the $=$-edge causes an inconsistency, generating an additional solution cost of 1, justifying the two one-entries in the cost vector. Generalizing this to the five other possibilities of putting exactly one labeled edge, we arrive at the following:

Lemma 3.8. *By inserting exactly one edge labeled $=$ or $\neq$ between the vertices from S, we obtain the six atomic cost vectors* $(0, 0, 1, 1)$, $(0, 1, 0, 1)$, $(0, 1, 1, 0)$, $(1, 0, 0, 1)$, $(1, 0, 1, 0)$, *and* $(1, 1, 0, 0)$.

All cost vectors in Lemma 3.8 have even parity. Hence, we need a second type of gadgets to be able to construct cost vectors with odd parity: gadgets that contain all vertices from S plus a new vertex connected to all vertices from S. We derive four atomic gadgets of this kind with different cost vectors, namely the cases that the edges connecting S to the new vertex are labeled $(\neq, \neq, \neq)$, $(=, =, \neq)$, $(\neq, =, \neq)$, or $(=, \neq, \neq)$ (an example is shown in Figure 3.7b).

Lemma 3.9. *By inserting one new vertex and connecting it to all vertices from S and assigning various edge labels, we obtain four atomic gadgets corresponding to the atomic cost vectors* $(0, 1, 1, 1)$, $(1, 0, 1, 1)$, $(1, 1, 0, 1)$, *and* $(1, 1, 1, 0)$.

The four atomic cost vectors from Lemma 3.9 all have odd parity. In this sense, we now may speak of *even* or *odd* cost vectors.

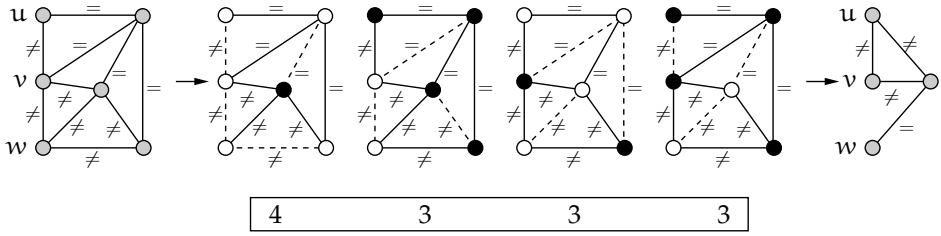


Figure 3.8: Example for the construction of a gadget with $|S| = 3$

Now, we can describe the general gadget construction. To do so, first note that vectors where all entries have the same value x are easy because this means that the solution for $G[S \cup C]$ is independent of the coloring of S and hence one can simply remove C and all edges between vertices of S and decrease the parameter k by x. This means that if we are given a cost vector (c_1, c_2, c_3, c_4), then without loss of generality we can *normalize* it by simply subtracting or adding the vector $(1, 1, 1, 1)$, each time decreasing or increasing the parameter by one. Now, given a cost vector (c_1, c_2, c_3, c_4), the gadget construction task one-to-one corresponds to finding a way to subtract atomic cost vectors from (c_1, c_2, c_3, c_4) such that one receives the vector $(0, 0, 0, 0)$. If we arrive at a cost vector with at least two 0-entries that cannot be transformed into $(0, 0, 0, 0)$, then due to the above reasoning we may also add the vector $(1, 1, 1, 1)$. Altogether, this results in the following algorithm:

1. Compute the cost vector for given S and C.
2. Normalize the cost vector by subtracting the vector $(1, 1, 1, 1)$ until at least one entry becomes 0.
3. If the cost vector has odd parity and has more than one 0-entry, then add $(1, 1, 1, 1)$.
4. If the cost vector has odd parity, then subtract a suitable odd atomic cost vector (that is, one that does not produce negative entries).
5. While the vector is not $(0, 0, 0, 0)$, repeat:
   (a) If the cost vector has three 0-entries, then add $(1, 1, 1, 1)$.
   (b) Subtract a suitable even atomic cost vector that decreases the maximum entry.

We show an example in Figure 3.8. We start with the induced subgraph $G[S \cup C]$, where $S = \{u, v, w\}$ is the separator. In the middle we show optimal solutions for the (up to symmetry) four possible colorings of S and mark by a dashed line the edges that have to be deleted. Cost figures are displayed below these figures, forming the cost vector $(4, 3, 3, 3)$. Normalization yields the vector $(1, 0, 0, 0)$. Since this is an odd vector with more than one zero, it gets padded to $(2, 1, 1, 1)$. This is an odd vector, so we need a gadget using an extra vertex and three edges (Lemma 3.9). From the three vectors whose subtraction decreases the maximum element 2, we arbitrarily choose $(1, 1, 1, 0)$, corresponding to adding from a new vertex a $\neq$-edge to u, a $\neq$-edge to v, and an $=$-edge to w. The remaining cost vector $(1, 0, 0, 1)$ can be covered by adding a $\neq$-edge between u and v, leaving the all-zero vector. The resulting gadget is shown on the right of Figure 3.8. We have subtracted the all-1 vector twice and added it once, and therefore the parameter decreases by one.

Theorem 3.3. *The above algorithm produces a gadget with the minimum number of vertices for every pair (S, C) where S is a 3-cut.*

Proof. First of all, it is clear from the one-to-one correspondence between atomic gadgets and atomic cost vectors that by "superimposing" the atomic gadgets corresponding to each (possibly multiple times) subtracted atomic cost vector, one directly arrives at an overall gadget (with possibly multiple edges). Concerning the usage of the normalization vector $(1, 1, 1, 1)$, we have already argued before that this does not affect the correctness of the gadget construction. Hence, in the remainder we focus on showing that the algorithm always terminates with having generated the vector $(0, 0, 0, 0)$ by subtracting atomic cost vectors and possibly subtracting or adding $(1, 1, 1, 1)$.

Once subtracting a suitable odd atomic vector, we arrive at a cost vector with even parity. In the further process, we will always have an even parity and it suffices to concentrate on the termination of the while-loop of the algorithm. Since the six atomic cost vectors represent all possible vectors with exactly two 1-entries, as long as we have at least two nonzero-entries in the cost vector, there is at least one even atomic cost vector which we can subtract. Now, assume that we have a cost vector with three zero-entries and one nonzero-entry e (this is the only remaining possibility besides having the all nonzero-entry). Then, the algorithm adds $(1, 1, 1, 1)$. Now, after this addition we can up to three times subtract an atomic vector, decreasing the entry e before again all entries except for e would be zero. Repeatedly proceeding this way, we thus always can arrive at the vector $(0, 0, 0, 0)$ in a finite number of steps. The construction is optimal because the gadget has at most one additional vertex (besides S), and this happens only for odd cost vectors, where it is unavoidable. □

Note that the construction is not necessarily optimal with respect to the number of edges introduced, nor with respect to the decrease in k. However, in our experiments these objectives rarely had different optimal solutions.

As a consequence of the considerations so far, we obtain the following result illustrating the power of our approach.

Corollary 3.2. *With the described data reduction scheme, all separators with* $|S| = 2$ *and* $|C| \geqslant 1$ *and all separators with* $|S| = 3$ *and* $|C| \geqslant 2$ *are subject to data reduction.*

3.2.3.3 Gadget construction for larger cuts

The gadget construction for 3-cuts already has required quite some machinery. The case of 4-cuts becomes still much more involved due to the increased combinatorial complexity. A provably optimal gadget construction as for 3-cuts currently does not seem practically feasible. Thus, we have chosen a heuristic approach for finding and constructing gadgets for 4-cuts.

We conjecture that atomic gadgets with at most two vertices in addition to the four separator vertices suffice. Thus, we generated 2^6 atomic gadgets with no extra vertex (corresponding to the choices of labels for the 6 edges within a 4-vertex separator), 2^4 atomic gadgets with one extra vertex (4 edges connecting a vertex in the separator to the new vertex), and 2^9 atomic gadgets with two extra vertices (8 edges to the new vertices, and one edge connecting the two new vertices). We then filtered out those that can be obtained by combining cheaper ones, and arrived after about five minutes of computation time at a set of 2948 atomic gadgets. They are stored in a fixed lookup table.

Once given this toolbox of atomic gadgets, we again try to derive the all-zero vector in a way analogous to the case of 3-cuts. This procedure is now realized by an exhaustive branch & bound algorithm. We start with the normalized vector. Should this fail, the vector $(1, 1, 1, 1)$ is added once and the procedure is repeated. Each gadget vector is associated with a cost corresponding to its number of extra vertices; this number is minimized. In fact, it is not too hard to see that this algorithm produces for 3-cuts, when given the 10 atomic cost vectors, the same result as the algorithm given for 3-cuts.

The branch & bound algorithm works quite well for cost vectors with small entries, but can become a bottleneck for vectors with high entries. We examine a simple heuristic to mitigate this in Section 3.2.4. We close with a description of challenges for further research that arise in our work with cost vectors. For this, we describe the scenario in a more abstract way.

Given a set S of n vectors of length l with nonnegative integer components, let a *linear combination* be a sum of some vectors of S, where vectors can occur multiple times (equivalently, have a positive integer scalar factor). Let a *basis*

be a set that allows to obtain any vector of length l with nonnegative integer components as a linear combination. (The terms are chosen in analogy to vector spaces, but because of the nonnegative integer restriction, we do not have a vector space here.) We face the following problems:

- How to recognize whether a vector set is a basis?
- Given a basis and a target vector t, how to find a linear combination that produces t?
- Given a large set of vectors, how can we find a smallest or minimal basis?

In our work, we actually have a small modification of this problem because as single vector with negative components also the vector $(-1,-1,\ldots,-1)$ is allowed. Also, the vectors come at different costs (number of new vertices), and we would like to find linear combinations of minimum cost.

This touches a deep and old subject in mathematics (see e. g. Barvinok and Woods [2003], Sturmfels [1996]). Seemingly, our questions seem to be more special than what is generally studied there, but this clearly deserves future theoretical studies.

3.2.4 Implementation and experiments

We applied our data reduction for Balanced Subgraph (Section 3.2) combined with the improved iterative compression routine (Section 4.5.4) to gene regulatory networks and randomly generated graphs. Our implementation consists of about 1600 lines of Objective Caml [Leroy et al. 1996] code and about 300 lines of C code that implements the time critical compression routine of the iterative compression method. All experiments were run on an AMD Athlon 64 3400+ machine with 2.4 GHz, 512 KB cache, and 1 GB main memory running under the Debian GNU/Linux 3.1 operating system. The program was compiled with the Objective Caml 3.08.3 compiler and the GNU gcc 3.3.5 compiler with options "-O3 -march=athlon". For the approximation algorithm by DasGupta et al. [2007], we used MATLAB version 7.0.1.24704 (R14). Our source code is available as free software from http://theinf1.informatik.uni-jena.de/bsg/.

Besides the data reduction rules described in Section 3.2.1, we additionally delete self loops and pairs of edges sharing the same end vertices if the edges have different types. These reductions can be seen as special cases of our data reduction scheme from Section 3.2.1 with $|C| = 0$ and $|S| = 1$ and $|S| = 2$, respectively. Furthermore, we only replace a small component by a gadget if this leads to an improvement; that is, either the number of vertices is reduced, or, in the case of an equal number of vertices, the number of edges is reduced.

Table 3.3: Comparison of approximation [DasGupta et al. 2007] and our exact algorithm. Here, t denotes the running time in minutes. For the approximation algorithm, "$k \leqslant$" is the solution size, and "$k \geqslant$" is the lower bound gained from the approximation guarantee. The approximation algorithm was run with 500 randomizations.

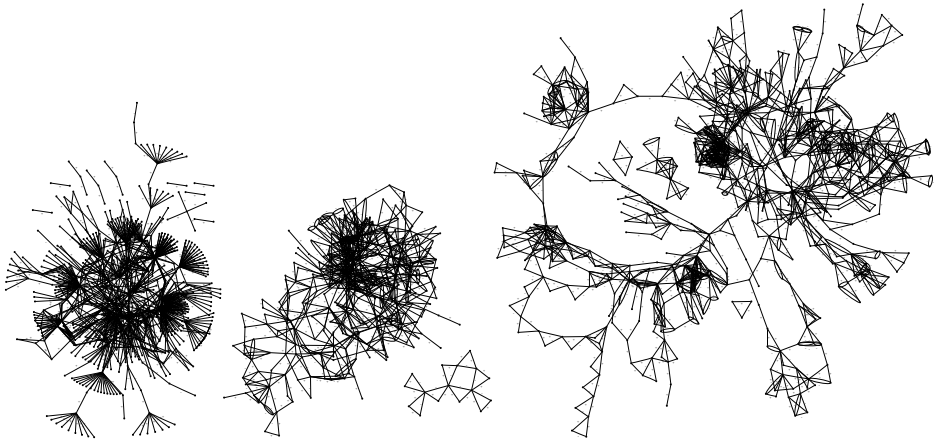
			Approximation			Exact alg.	
Data set	n	m	$k \geqslant$	$k \leqslant$	t	k	t
EGFR	330	855	196	219	7	210	108
Yeast	690	1082	0	43	77	41	1
Macrophage	678	1582	218	383	44	374	1

Additionally, we tested a heuristic running time improvement to circumvent a problem with the data reduction based on 4-cuts: For some instances the running time drastically increased because we encountered a cost vector with entries having high values. This increased the number of possible linear combinations and therefore the running time. An example appeared when the algorithm processed the regulatory yeast network: it ran into the cost vector $(2, 8, 8, 0, 31, 39, 39, 31)$, and therefore the instance could not be solved within several hours (whereas it could be solved without 4-cut reduction within minutes). To take advantage of 4-cut reductions without wasting hours of running time through such (rarely occurring) cases, based on experimental findings we introduced a new cut-off parameter. More precisely, we stop the gadget construction if the sum of the entries of a cost vector is more than 25. We experimentally show in below that this cut-off value is sufficient for the considered networks.

As a further comparison point, we implemented an integer linear programming (ILP)-based approach, which is a straightforward extension of that for Edge Bipartization (Section 4.6.3). However, when solved by GNU GLPK [Makhorin 2004], it was consistently outperformed by the iterative compression approach as soon as the heuristic speedup mentioned in Section 4.5.3 was employed; therefore, we do not give details on its performance.

Biological Networks. We started our experimental investigations with gene regulatory networks up to the size of about 700 vertices and more than 7000 edges.

We begin with comparing our algorithm to the randomized approximation algorithm of DasGupta et al. [2007]. The authors considered the regulatory networks of yeast and human epidermal growth factor (EGFR) that are graphically displayed in Figure 3.9. We additionally examined a macrophage network [Oda



(a) Yeast (b) EGFR (c) Macrophage

Figure 3.9: Example gene regulatory networks

et al. 2004]. The yeast network is larger than the EGFR network, but one can immediately see applicability of data reduction rules in form of many degree-1 and -2 vertices. The results of both algorithms are given in Table 3.3. Apart from giving an optimal solution instead of an approximative one, we could also decrease the running time for the yeast and macrophage networks from about one hour to less than a minute. Note, however, that the running time of the approximation algorithm could probably be much improved by implementing it in a more efficiently executed language, or simply by doing fewer randomized trials at the cost of a possibly worse result. For the macrophage network, we could compute an optimal solution of size $k = 374$. This emphasizes the importance of our data reduction rules, since for such high solution sizes the iterative compression algorithm (Section 4.5.4) cannot be applied directly. Furthermore, here it is remarkable that the data reduction breaks up the network into several smaller components of up to 70 vertices that have to be solved by iterative compression independently, whereas for the other two networks only one large component remains after data reduction. As a further comparison point, the ILP-based approach was not able to solve the three instances even after applying the data reduction.

To investigate the power of our data reduction rules for different sizes s of the separator S, we investigated stepwise in terms of s the results for the yeast, EGFR, and macrophage networks. The results are given in Table 3.4, where

Table 3.4: Size of the largest component remaining and overall running time t (including solution by iterative compression) when reducing only separators up to size c.

	Yeast			EGFR			Macrophage		
s	n	m	t	n	m	t	n	m	t
0	690	1080	91 s	329	783	$>$ 15 h	678	1582	$>$ 1 day
1	321	709	77 s	290	727	$>$ 15 h	535	1218	$>$ 1 day
2	173	469	11 s	167	468	$>$ 15 h	140	397	$>$ 1 day
3	155	424	4 s	99	283	$>$ 15 h	113	335	$\approx$ 1 day
4	?	?	$>$ 5 h	89	259	108 min	70	228	4.5 h
4r	144	405	5.6 s	89	260	97 min	70	228	18 s

setting s to 4r means that we use a cut-off of 25 for the sum of the entries of a cost vector in the case of cut sets of size 4.

We denote applying our data reduction to a separator of size s by s-*reduction*. The yeast network can already be solved with improved iterative compression and 2-reduction. In contrast, the EGFR network cannot be solved within reasonable time without also using 3- and 4-reduction. For the macrophage network, the use of 4-cuts reduces running time severely.

We now investigate 4-reduction with and without cut-off value. For all networks, we could achieve the best data reduction results by using 4r-reduction: As mentioned above, for the yeast network the "normal" 4-reduction does not return any results within 5 hours. In contrast to the other entries for which we aborted the experiments in Table 3.4, here the running time is caused by the data reduction itself and not due to the iterative compression routine. Therefore, we cannot give the size of the reduced graph. Setting the cut-off parameter to 25, we obtained an instance that is more reduced than by applying 3-reduction alone. The reason that we still cannot achieve a better overall running time is the running time for the 4r-reduction itself. For the EGFR network, the size of the largest component barely changes going from 4- to 4r-reduction, indicating that we do not lose much by the cut-off; in fact, we achieve a better overall running time for 4r. Applying 4r-reduction instead of 4-reduction to the macrophage network does not change the size of the remaining largest component, but decreases the running time from hours to seconds.

Note that we really need the combination of data reduction and the improvements of iterative compression to solve the instances.

Table 3.5: Reduction effect for random networks. Average over 5 instances for each column. Here, n is the number of vertices in the original graph, n' is the number of vertices after data reduction, m' is the number of edges after data reduction, and t is the running time in seconds.

n	100	200	300	400	500	600	700	800	900	1000
m	172.6	336.8	492.4	640.2	791.2	970.6	1108.8	1286.6	1435.6	1585.6
n'	29	48.8	75	95	119.8	153.2	169.2	193.4	211.6	239.6
m'	102.3	165.8	252	324	398.4	518	565.8	672.4	734.6	815.8
t	1	7	6	5.5	6	8.5	8	15.5	18.5	15.5

Further regulatory networks. We also considered four small regulatory networks obtained from the Panther pathways database [Mi et al. 2005], consisting of about 100 vertices and up to 200 edges. With 3-reduction we could compute optimal solutions ranging from 20 to 28 in split seconds.

Finally, we describe our results for two larger networks that cannot yet be solved optimally with our method. For the regulatory network for a toll-like receptor [Oda and Kitano 2006], we could reduce the number of vertices from 688 to 244 and the number of edges from 2208 to 1159 within three minutes. For the regulatory network of the archaeon *Methanosarcina barkeri* [Feist et al. 2006], we were less successful. The number of vertices was decreased from 628 to 500 and the number of edges from 7302 to 6845 in 25 minutes. This could be a hint that the dense structure of this network is hard to attack by our data reduction.

Random Networks. To further substantiate our experimental results, we generated a test bed of random graphs with the algorithm described by Volz [2004]. We tried to model the yeast network by choosing the following settings: the cluster coefficient is 0.016, the distribution of vertex degrees follows power-law with $\alpha = -2.2$, and the probability to assign $\neq$ to an edge is 20.5 %.

We generated 5 instances each for graph sizes ranging from 100 to 1000 vertices. The number of edges of the generated instances is slightly more than 1.5 times of the number of vertices. We investigated the power of our data reduction by computing the number of vertices and edges of the reduced instances. Table 3.5 shows the average results for instances of each size. Independent of the instance size, about 75% of the vertices are reduced. Note that this is also true for the yeast network that we try to model.

The results given in Table 3.5 are received with setting the cut-off parameter again to 25. Redoing the test with a higher threshold of 50 did in no case change the number of reduced edges or vertices by more than one, but increased the

running time for some instances from seconds to several hours.

Considering the size of instances that can be solved optimally by improved iterative compression after data reduction, here the threshold seems to be at graphs with about 500 vertices. Three out of the five instances could be optimally solved in up to 20 hours, where the sizes of the optimal solutions are between $k = 76$ and $k = 91$. Note that the solution sizes are higher than for the yeast network, which has more than 600 vertices and an optimal solution of size 41. Because of this, the random instances seem to be somewhat more difficult than the yeast network itself, which is consistent with observations by DasGupta et al. [2007].

3.2.5 Outlook

There are numerous avenues for future research. DasGupta et al. [2007] also introduced a directed version of the Balanced Subgraph problem. The approximation results are worse than for the undirected case, which is probably why there is no implementation yet [DasGupta et al. 2007]. Fortunately, the directed case can be reduced to the Vertex Bipartization problem, which can be solved in $O(3^k \cdot mn)$ time (Section 4.6). Again, this opens the route for experimental studies. We conclude with some further research possibilities.

- Edge Bipartization and Balanced Subgraph still lack a problem kernel with a nontrivial size bound on the problem kernel size. Perhaps our data reduction scheme can be a first step in this direction.

- Chiang et al. [2007] use the fact that Balanced Subgraph is polynomial-time solvable on planar graphs to obtain good results for their "nearly-planar" instances. It would be interesting here to have a fixed-parameter algorithm where the parameter is the "distance from planarity". It is NP-hard to solve Balanced Subgraph for graphs that are planar with already a single vertex added [Barahona 1980]; however, the number of edges added, or the minimum number of crossings of a plane drawing might be a useful parameter.

- The theoretical problems that arise in the construction of optimal gadgets (Section 3.2.3.3) deserve further investigation.

- In principle, our data reduction scheme is applicable to all graph problems where a coloring of the vertices is sought. This includes problems where a subset of the vertices is sought, such as Vertex Cover or Dominating Set. However, it remains to find appropriate gadget constructions for problems other than Balanced Subgraph. It seems promising to extend

our data reduction scheme to practical solutions of other graph problems. A loosely related approach—also based on graph separation but without the gadgeteering—has been used for solving Steiner tree problems [Polzin and Vahdati Daneshmand 2006].

- Estivill-Castro et al. [2006, Section 3.3] also sketch a general approach to data reduction rules. In particular, they suggest to use the algebraic Myhill-Nerode machinery adapted to graph theory [Fellows and Langston 1989, Downey and Fellows 1999]. It is possible that this approach can be adapted to the task of computing gadgets for arbitrary-sized separators. This might also lead to a formal characterization of graphs for which our separation-based data reduction scheme is useful. This is clearly an interesting area of further research.

Chapter 4

Iterative compression

Of the three main techniques we examine, iterative compression is by far the youngest, appearing first in a work by Reed et al. in 2004. Although not quite as generally employable as data reduction or search trees, it appears to be applicable to a wide range of problems, and it has already led to several breakthroughs in showing fixed-parameter tractability results.

For instance, the Vertex Bipartization problem, that is, the task of finding a minimum set of vertices whose deletion destroys all odd-length cycles, has been shown to be fixed-parameter tractable with respect to the number of deleted vertices by means of iterative compression [Reed et al. 2004]. For years this had been an important open problem in parameterized complexity [Mahajan and Raman 1999]. A number of further fixed-parameter results for various feedback set problems in graphs have been found since [Dehne et al. 2007, Guo et al. 2006, Dom et al. 2006b, Marx 2006]. Finally, iterative compression was used to settle the fixed-parameter tractability of Directed Feedback Vertex Set [Chen et al. 2008], which for quite a while was probably the most notorious concrete open questions in the field of parameterized complexity, already mentioned by Downey and Fellows [1995].

Structure of the chapter. We survey known results on iterative compression in Section 4.1. We then illustrate the basic method of iterative compression by means of the 3-Hitting Set problem in Section 4.2. Next, we show that also the Cluster Vertex Deletion problem (Section 4.3) and the Feedback Vertex Set problem in tournaments (Section 4.4) are amenable to iterative compression. Then, we provide an iterative compression algorithm for Edge Bipartization (Section 4.5) and present several speedups. These results generalize to the

Balanced Subgraph problem (Section 4.5.4). Further, they can be adapted to the variant Vertex Bipartization (Section 4.6).

Since the running time bounds of iterative compression algorithms are typically moderate compared to other techniques, iterative compression is attractive for practical implementations; however, no experimental results were known. We present the first experimental evidence that iterative compression is a worthwhile alternative for solving Balanced Subgraph (Section 3.2.4) and Vertex Bipartization in practice (Section 4.6.3).

4.1 Known results

All currently known iterative compression algorithms solve graph modification problems for hereditary graph classes (see Section 1.5 for definitions and general results on such problems). We discuss later in this section why these problems are particularly suited for iterative compression. Most of the algorithms solve *feedback set problems* in graphs, that is, problems where one wishes to destroy certain cycles in the graph by deleting at most k vertices or edges (see Festa et al. [1999] for a survey on feedback set problems). Clearly, these are all graph modification problems for hereditary graph properties, but since they have an infinite set of forbidden subgraphs, it is not immediately clear that they are in FPT.

- Vertex Bipartization: Destroy all odd cycles by deleting a minimum number of vertices (Reed et al. [2004], Section 4.6).
- Edge Bipartization: Destroy all odd cycles by deleting a minimum number of edges (Section 4.5).
- Feedback Vertex Set: Destroy all cycles by deleting vertices [Dehne et al. 2007, Guo et al. 2006, Chen et al. 2007a].
- Feedback Vertex Set in tournaments: Destroy all cycles in a tournament (that is, a directed graph with exactly one directed edge between any two vertices) by deleting a minimum number of vertices (Section 4.4).
- Chordal Deletion: Destroy all chordless cycles (that is, induced cycles of length at least 4) by deleting a minimum number of edges [Marx 2006].
- Directed Feedback Vertex Set: Destroy all cycles in a directed graph by deleting a minimum number of vertices [Chen et al. 2008].

For some further graph problems that are not feedback set problems, iterative compression has been used. These all have a characterization by a finite set of

forbidden subgraphs, and so a simple fixed-parameter search tree algorithm can be given; however, the iterative compression algorithms have a much smaller exponential part of the running time.

- Vertex Cover: Destroy all edges by deleting vertices [Guo 2006, Wernicke 2006, Peiselt 2007]. These algorithms mostly served to explore the technique of iterative compression and are not competitive with search tree techniques such as those of Chen et al. [2006].
- 1-regular Deletion: Make the graph 1-regular (that is, every vertex has degree 1) by deleting vertices [Moser 2007]. This can be seen as a variant of Vertex Cover, since Vertex Cover is equivalent to 0-regular Deletion.
- Cluster Vertex Deletion: Transform a graph into a disjoint union of cliques by deleting vertices (Section 4.3).

Finally, recently the first application of iterative compression to a problem that is not a graph problem was given:

- Almost 2-SAT: Remove clauses to make a 2-CNF formula satisfiable [Razgon and O'Sullivan 2008].

The flexibility of iterative compression is illustrated by the fact that the Feedback Vertex Set algorithm can be extended to weighted inputs [Chen et al. 2007a] and to enumerate all solutions [Guo et al. 2006].

4.2 Basic method

The central idea of iterative compression is to use structural induction, for example over the vertices of a graph. Given an instance graph G, we assume that we already have a means to solve a smaller instance $G - v$, that is, G with one vertex deleted. It is usually easy to adapt the solution for $G - v$ to a solution for G; however, this solution might then not be optimal anymore. This naturally leads to the concept of a *compression routine*.

Definition 4.1. *A* compression routine *is an algorithm that, given a problem instance and a solution, either calculates a smaller solution or proves that the given solution is of minimum size.*

Using this routine, one finds an optimal solution to a problem by inductively building up the problem structure and compressing intermediate solutions. The induction can easily be replaced by iteration, which also explains the name of iterative compression; we will use the iterative presentation in the following.

Iterative compression can often lead to fixed-parameter algorithms, where the parameter is the solution size. The point is that if we manage to bound the size of any intermediate solution to be compressed by the parameter, and the compression routine is a fixed-parameter algorithm, then so is the whole algorithm.

The strength of iterative compression is that it allows to see the problem from a different angle: The compression routine does not only have the problem instance as input, but also a nearly-optimal solution, which carries valuable structural information on the input. Also, it does not need to find an optimal solution at once, but only any better solution. Therefore, the design of a compression routine can often be simpler than designing a complete fixed-parameter algorithm.

However, while the mode of use of the compression routine is often straightforward, finding the compression routine itself is typically not. It is not even clear that a compression routine with interesting running time exists even when we already know a problem to be fixed-parameter tractable. Therefore, the art lies in designing the compression routine.

Iterative compression for 3-Hitting Set. As introductory example, we use 3-Hitting Set. To emphasize the similarity to the other problems of this section, we formulate it as a hypergraph modification problem.

> 3-Hitting Set
> **Instance:** A hypergraph $G = (V, E)$ with $|e| = 3$ for all $e \in E$ and an integer $k \geqslant 0$.
> **Question:** Is there a *hitting set* $X \subseteq V$ with $|X| \leqslant k$, that is, a set of vertices whose deletion destroys all hyperedges, that is, yields $E = \emptyset$? Here, deleting a vertex implies also completely deleting all hyperedges that contain this vertex.

3-Hitting Set is NP-hard [Garey and Johnson 1979]. There is a simple 3-approximation (repeatedly take all three vertices of a hyperedge); it has been conjectured that this approximation factor cannot be improved [Khot and Regev 2008]. Note that the variant 2-Hitting Set is equivalent to the NP-hard Vertex Cover problem. 3-Hitting Set has applications in the study of phylogenetic trees [Downey et al. 1999]; further, it generalizes several problems such as Cluster Vertex Deletion and Feedback Vertex Set in tournaments, for which we will give specialized, more efficient iterative compression algorithms in Section 4.3 and Section 4.4, respectively. 3-Hitting Set can be solved in $O(3^k \cdot m)$ time by a simple search tree algorithm: choose any hyperedge $\{v_1, v_2, v_3\} \in E$ and branch into the three cases $v_1 \in X$, $v_2 \in X$, and v_3 in X. By case distinction and careful analysis, this has been improved in a series of results to $O(2.270^k + m)$

```
IterativeCompression(G = (V, E))
1   V' ← ∅
2   X ← ∅
3   for each v ∈ V:
4       V' ← V' ∪ {v}
5       X ← X ∪ {v}
6       X ← Compress(G[V'], X)
7   return X
```

Figure 4.1: Pseudo-code for iterative compression, using the compression routine Compress

[Niedermeier and Rossmanith 2003], then $O(2.179^k + m)$ [Fernau 2004, 2005], and finally $O(2.076^k + m)$ [Wahlström 2007]. A kernel of size $O(k^3)$ is known [Niedermeier and Rossmanith 2003], which has recently been improved to $O(k^2)$ vertices and $O(k^3)$ edges [Abu-Khzam 2007].

The most obvious way to employ a compression routine is to start with an approximate solution and then use the compression routine until no further compression is possible. However, since the running time of the compression routine depends exponentially on the size of the solution to compress, it is faster to build up the graph vertex-by-vertex while always keeping a minimal solution. This is illustrated in the pseudo-code in Figure 4.1.

We start with $V' = \emptyset$ and $X = \emptyset$; clearly, X is a minimum hitting set for $G[V']$. In lines 4 and 5, we add one vertex $v \notin V'$ from V to both V' and X. Then X is still a hitting set for $G[V']$, although possibly not a minimum one. We can, however, obtain a minimum one by applying our compression routine. Here, the compression routine Compress takes a hypergraph G and a hitting set X for G, and returns a smaller hitting set for G if there is one; otherwise, it returns X unchanged. Therefore, it is a loop invariant that X is a minimum-size hitting set for $G[V']$. Since eventually $V' = V$, we obtain an optimal solution for G once the algorithm returns X.

Note that we defined a compression routine as a function that returns a *smaller* solution, but not necessarily a minimum one. This suffices here, because the hitting set $X \cup \{v\}$ to be compressed can be larger by at most one than an optimal hitting set X' for $G[V' \cup \{v\}]$; this is because X' is also a hitting set for $G[V']$, and cannot be smaller than the minimum hitting set X. We come back to this property at the end of Section 4.1.

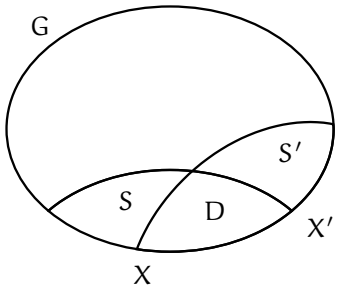


Figure 4.2: Partition of X

```
COMPRESS(G, X)
1   for each S ⊆ X:
2       D ← X \ S
3       if G[S] is a hyperedge-free graph:
4           G′ ← G[V \ D]
5           S′ ← COMPRESSDISJOINT(G′, S)
6           if |S′| < |S|:
7               return (X \ S) ∪ S′
8   return X
```

Figure 4.3: Pseudo-code for COMPRESS

It remains to describe the compression routine. The basic idea, which is shared with all other known iterative compression algorithms, is to reduce the compression problem to a *disjoint* compression problem:

Definition 4.2. *A* disjoint compression routine *is an algorithm that, given a problem instance and a solution* X*, either calculates a smaller solution that is disjoint from* X *or proves that this is not possible.*

The reason for working with a disjoint compression routine is that it gives us extra structure to work with: Not only do we know that $G \setminus X$ is hyperedge-free, but also that $G[X]$ is hyperedge-free, because otherwise we can immediately claim that no compression is possible, since we are not allowed to delete vertices from X.

For the transformation from compression to disjoint compression, consider a smaller solution X' as a modification of the known solution X. It will retain some vertices D from X and replace the other vertices S with fewer vertices S' (Figure 4.2). The idea (see Figure 4.3) is to try by brute force all $2^{|X|}$ possibilities to partition X into S and D (line 1). If $G[S]$ still has hyperedges, then there is no solution disjoint from S, and we can skip this partition (line 3). Since we decided to keep all vertices of X in the solution except for those in S, we can immediately get rid of the other vertices (line 4). We have thus gained the disjointness assumption at a cost of a factor of $2^{|X|} = O(2^k)$ in the running time. For 3-HITTING SET, it seems difficult to get this assumptions in a cheaper way; we later show how to achieve an equivalent result for EDGE BIPARTIZATION by a simple input transformation (Graph Transformation 4.1), but that does not seem to be applicable here. It now remains to find in $G[V \setminus D]$ an optimal hitting set that is disjoint from S, which is done by the function COMPRESSDISJOINT.

To implement COMPRESSDISJOINT, we examine possible configurations of hyperedges (Figure 4.4). Configuration (a) is not possible because of the check

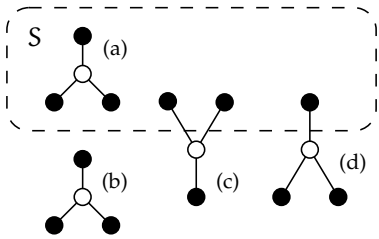


Figure 4.4: Hyperedges in disjoint compression for 3-Hitting Set. Black circles are vertices, white circles connected to three vertices are hyperedges.

in line 3. Configuration (b) is not possible either, because S is a hitting set for G'. If we encounter configuration (c), we can immediately delete the single vertex that is not in S, since there is no other way to get rid of these hyperedges. So the only remaining possibility is (d): each remaining hyperedge has exactly one vertex in S and two in $V \setminus S$. Since we are not allowed to remove any vertex in S, we might as well omit them. This leaves us with a number of 2-element edges, the task still being to remove vertices to get rid of all edges. This is exactly the Vertex Cover problem. For Vertex Cover, many fast parameterized algorithms exist, which we can use to solve the remaining instance. We arrive at the following theorem.

Theorem 4.1. 3-Hitting Set *can be solved in* $O(2.274^k kn^2)$ *time by using iterative compression.*

Proof. The data reduction in CompressDisjoint (removal of edges with two vertices in S) can be executed in $O(kn)$ time, if we do it incrementally and enumerate subsets of X in a way such that at each step only membership of one vertex changes; this can be done using a Gray code [Knuth 2004, Section 7.2.1.1]. Then, the remaining task is to solve a Vertex Cover instance with at most n vertices and m edges. Vertex Cover with a cover size of at most k' can be solved in $O(1.274^{k'} + k'n)$ time [Chen et al. 2006]. We thus can execute Compress in $O(\sum_{S \subseteq X}(1.274^{|S|} + |S|n))$ time. Using $\sum_{i=0}^{k} \binom{k}{i} c^i = (c+1)^k$ for any c and the fact that $|S|$ is bounded by $k+1$, this gives an $O(2.274^k kn)$ time bound. The compression routine is called at most n times, giving an overall running time of $O(2.274^k kn^2)$ as claimed. □

Using a kernelization [Niedermeier and Rossmanith 2003] and the fact that the rounding of the exponential base allows us to omit polynomial factors of k, we can even claim a running time of $O(2.274^k + m)$ (although this borders on abuse of the Big O notation).

The running time of this iterative compression algorithm is already competitive with that of the algorithm of Niedermeier and Rossmanith [2003], which runs in $O(2.270^k + m)$ time; however, it is not as fast as the best known 3-Hitting Set algorithm by Wahlström [2007] running in $O(2.076^k + m)$ time. Still, it might be a useful approach to solving 3-Hitting Set in practice, in particular since except for the Vertex Cover subroutine, it is very simple, and high-performance Vertex Cover implementations have been presented (e. g. Abu-Khzam et al. [2004a], Felner et al. [2004]).

Furthermore, we can in the same way use iterative compression to solve 4-Hitting Set using a 3-Hitting Set algorithm, or more generally d-Hitting Set using iterative compression and a $(d-1)$-Hitting Set algorithm. If we use the 3-Hitting Set algorithm by Wahlström [2007], we obtain the following theorem.

Theorem 4.2. 4-Hitting Set *can be solved in* $O(3.076^k + m)$ *time, and* 5-Hitting Set *can be solved in* $O(4.076^k + m)$ *time.*

These algorithms are faster than the previously fastest known by Fernau [2005] running in $O(3.116^k + m)$ and $O(4.079^k + m)$ time, respectively. For d-Hitting Set with $d > 5$, this approach does not yield new records anymore; further, we have an increasing polynomial overhead with growing d.

We now examine how compression routines work for the known algorithms. They all start with the same opening move that we used for 3-Hitting Set, namely forcing the new solution to be disjoint from the known one. After that, however, they widely diverge:

- For Vertex Bipartization and Edge Bipartization, it is possible to reduce the remaining task to finding a vertex (resp. edge) cut set, which can be done in polynomial time by maximum flow techniques. We describe this in detail for Edge Bipartization in Section 4.5.4. The running time is $O(2^k \cdot m^2)$ for Edge Bipartization and $O(3^k \cdot mn)$ for Vertex Bipartization.

- For Feedback Vertex Set, data reduction rules allow to shrink the remaining instance so that it can be solved by brute force [Dehne et al. 2007, Guo et al. 2006] or a search tree [Chen et al. 2007a], the latter yielding an $O(5^k kn^2)$ time bound.

- For Feedback Vertex Set in tournaments, data reduction further constrains the possible solution such that it can be found with a polynomial-time Longest Increasing Subsequence routine. We describe this algorithm with a running time bound of $O(2^k \cdot n^2(\log n + k))$ in detail in Section 4.4.

- For Chordal Deletion, the graph is reduced until it obtains bounded treewidth, a property that allows fixed-parameter algorithms [Marx 2006].

The running time of the overall algorithm is not stated explicitly and a simple analysis cannot bound the running time by $c^k \cdot n^{O(1)}$ for any c.

- For Directed Feedback Vertex Set, the remaining task is reduced to a Multicut variant in a directed acyclic graph [Chen et al. 2008]. The running time is $k!\,8^k \cdot n^{O(1)}$.
- For Vertex Cover, the remaining task is trivially polynomial-time solvable. By omitting to consider certain subsets of the known solution, the running time can be reduced to $O(1.443^k mn\sqrt{n})$ [Peiselt 2007].
- For 1-regular Deletion, the instance is reduced until the remaining instance can be solved by finding a maximum flow [Moser 2007]. The running time is $2^k \cdot n^{O(1)}$.
- For Cluster Vertex Deletion, after some data reduction the remaining instance can be solved in polynomial time using matching techniques, yielding a running time of $O(2^k \cdot km\sqrt{n}\log n)$.
- For Almost 2-SAT, the problem is reduced to the satisfiability variant 2-SLASAT by iterative compression [Razgon and O'Sullivan 2008]. The running time is $O(15^k \cdot km^3)$.

There have been some attempts at classifying problems amenable to iterative compression. As Guo [2006, Section 2.1] notes, we need an *augmentation element*, over which the main loop (Figure 4.1, line 3) iterates. In all known examples, this is either a vertex or an edge of a graph. For iterative compression to work, the problem must behave *monotonous* with respect to adding augmentation elements, that is, an intermediate solution must not become a smaller by adding an augmentation element. This is because otherwise we cannot any longer bound the size of intermediary solutions X by k. As an example, consider the following NP-hard problem:

Cluster Deletion
Instance: An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k edge deletions into a *cluster graph*, that is, a graph where every connected component is a clique?

A natural choice for augmentation elements is edges. However, the problem is not monotonous then. To see this, consider as input an n-vertex clique. Before the last iteration, we have a clique with one edge missing, which requires $n-2$ edge deletions. However, when we add the final edge, we need no deletion at all anymore, so the clearly the size of the intermediary solution cannot be bounded by $k = 0$.

For vertex deletion problems with vertices as augmentation elements, it suffices that the graph class is hereditary to get the monotonicity property: if we add one vertex v to a graph, the optimal solution size cannot become smaller, since if X is a solution for $G + v$, then $X - v$ is a solution for G by the hereditariness, and this solution is not larger. For edge deletion problems with edges as augmentation elements, we additionally need the graph class to be *monotone*, that is, closed under vertex *and edge* deletions, which is not the case for cluster graphs. However, iterative compression might still be feasible for non-hereditary or non-monotone classes by using a different augmentation element.

4.3 Iterative compression for Cluster Vertex Deletion

In this section, we show how to solve the weighted Cluster Vertex Deletion problem by iterative compression. This yields the fastest currently known FPT algorithm for this problem. It also demonstrates that iterative compression can be successfully employed for problems that are not feedback set problems. Further, it is one of the first examples where iterative compression is used to solve a weighted problem (another recent example was given by Chen et al. [2007a]). We first state the unweighted version.

Cluster Vertex Deletion
Instance: An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k vertex deletions into a *cluster graph*, that is, a graph where every connected component is a clique?

Cluster Vertex Deletion is motivated by graph-modeled clustering (see Hüffner et al. [2007c] for a survey on FPT techniques in graph-modeled clustering). The underlying model is that we have a number of samples, some of which are equivalent (e. g., DNA samples, some of which are from the same species) and a method to test two samples for equivalence. A graph is formed where each vertex corresponds to a sample and an edge between two vertices is added when their samples are tested as equivalent. In the absence of errors, the resulting graph is a cluster graph, where each connected component corresponds to an equivalence class of the equivalence relation (e. g., a species). However, an unknown subset of samples is contaminated and produces unpredictable comparisons to other samples. A minimum solution for Cluster Vertex Deletion, that is, a minimum set of vertices whose deletion produces a cluster graph, provides the most parsimonious explanation for the data under this model. Note that cluster graphs are exactly the graphs with clustering coefficient 1 (the

clustering coefficient is defined as the probability that if $\{u, v\} \in E$ and $\{v, w\} \in E$, then also $\{u, w\}$ in E).

Cluster Vertex Deletion can also be seen as the problem of making a symmetric relation transitive by omitting a minimum number of elements. The related problem of making an antisymmetric relation transitive by omitting a minimum number of elements is also known as Feedback Vertex Set in tournaments; we will also show an iterative compression algorithm for Feedback Vertex Set in tournaments in Section 4.4, albeit using very different algorithmic techniques for the compression routine.

In practical clustering applications, we often have additional information available on the confidence in certain measurements. For example, Rahmann et al. [2007] advocated to consider *weighted* Cluster Editing, where a real number describes the confidence put into the correctness of each edge or non-edge. In our setting, we can use vertex weights to model how much confidence we have in a sample to be non-faulty. The task is then to delete a set of vertices with minimum total weight to obtain a cluster graph:

Weighted Cluster Vertex Deletion
Instance: An undirected graph $G = (V, E)$, a vertex weight function $\omega : V \to [1, \infty)$, and a nonnegative number t.
Question: Is there a subset $X \subseteq V$ with $\sum_{v \in X} \omega(x) \leqslant t$ such that deleting all vertices in X from G results in a cluster graph?

4.3.1 Known results on Cluster Vertex Deletion

We first consider the unweighted case. The variant Cluster Editing, where the assumption is that the data has been distorted by adding and deleting edges instead of vertices, has been studied in several works (e. g., Shamir et al. [2004], Bansal et al. [2004]). For Cluster Editing, the task is to delete or insert at most k edges such that the input graph becomes a cluster graph. The search tree algorithm of Gramm et al. [2005] with a running time bound of $O(2.270^k + n^3)$ has been experimentally evaluated [Dehne et al. 2006]. A faster algorithm running in $O(1.920^k + n^3)$ time was obtained using computer-generated search trees [Gramm et al. 2004]. A recent manuscript by Böcker et al. [2007] claims a running time of $O(1.83^k + n^3)$ by using a different branching strategy and reports further experimental results. The best known polynomial-time approximation is by a factor of 2.5 [Ailon et al. 2005b, van Zuylen and Williamson 2007].

In comparison to Cluster Editing, so far Cluster Vertex Deletion has been neglected. The class of cluster graphs is hereditary, and thus Cluster Vertex Deletion is NP-hard and MaxSNP-hard (see Section 2.1). Because of the hereditariness, it must have a characterization by forbidden subgraphs. In fact, this well-known characterization is very simple.

Lemma 4.1. *A graph is a cluster graph iff it does not contain an induced* P_3*, that is, an induced path of* 3 *vertices.*

Proof. We show both directions by contraposition. If a graph contains a P_3 with vertices u, v, w, then the connected component containing u, v, and w is clearly not a clique. If a graph is not a cluster graph, then there is a connected component where two vertices u and w are not connected by an edge. Let a shortest path between u and w be $v_1 = u, v_2, \ldots, v_c = w$. Then v_1, v_2, v_3 forms a P_3. □

This characterization leads to a simple polynomial-time factor-3 approximation: Repeatedly find a P_3 and delete all three of its vertices. The characterization also easily yields an $O(3^k m)$ time search tree algorithm: Find a P_3, and branch into 3 cases, corresponding to deleting one of the three vertices of the P_3. Note that a P_3 can be found in linear time. Gramm et al. [2004] used an elaborate case distinction found with computer help to improve this search tree algorithm to $O(2.257^k m)$ time. The fastest previously known algorithm, however, is obtained by a reduction to the 3-Hitting Set problem introduced in Section 4.2: we use V as ground set and the set of P_3's as subset collection. In combination with the currently best known (rather involved) 3-Hitting Set algorithm [Wahlström 2007], this yields a running time of $O(2.076^k + n^3)$.

Abu-Khzam [2007] showed that 3-Hitting Set has a kernel of $O(k^2)$ vertices. The running time of the 3-Hitting Set kernelization algorithm is the maximum of $O(m)$ and $O(n^{2.5})$, where m is the number of hyperedges and n is the number of vertices, and the kernel is an induced subhypergraph of the original instance. Therefore, it can be used to find an $O(k^2)$-vertex kernel for Cluster Vertex Deletion by collecting the set of $O(n^3)$ P_3's, applying the 3-Hitting Set kernelization, and then taking the subgraph induced by the remaining vertices. The total running time of this kernelization is $O(n^3)$. In contrast, for Cluster Editing, after a series of improvements [Gramm et al. 2005, Protti et al. 2006, Fellows et al. 2007b], a kernel of only $4k$ vertices is known [Guo 2007].

We now turn our attention to the weighted case. For both Cluster Editing and Cluster Vertex Deletion, the simple 3^k-size search tree still works, because we can bound the height of the search tree by k, since every vertex has weight at least 1. However, more sophisticated branching strategies for the unweighted case do not necessarily apply. For weighted Cluster Editing, Rahmann et al. [2007] give experimental results for the 3^k-size search tree. Böcker et al. [2007] improve the running time to $(1.83^k + n^3)$ and give a problem kernel of $O(k^2)$ vertices, which can be found in $O(n^3)$ time. The best known polynomial-time approximation is by a factor of 4 [Charikar et al. 2005]. For weighted Cluster Vertex Deletion, the previously best algorithm is obtained by a reduction to weighted 3-Hitting Set [Fernau 2006a], which yields a running time of

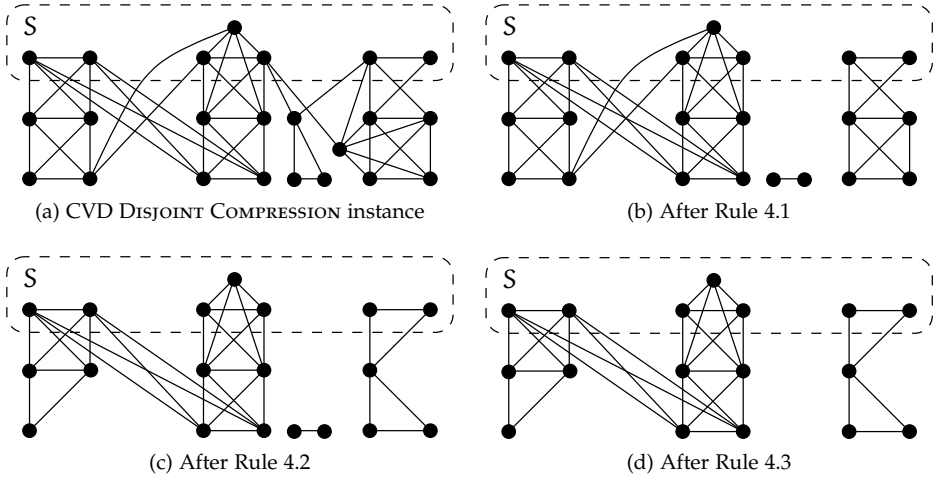
```
CompressCVD(G, X)
 1  X' ← X
 2  for each S ⊆ X:
 3      if G[S] is a cluster graph:
 4          G' ← G \ (X \ S); R ← V(G' \ S)
 5          G' ←ReduceRule4.1(G')
 6          G' ←ReduceRule4.2(G')
 7          G' ←ReduceRule4.3(G')
 8          Classify each vertex u in R according to N(u) ∩ S
 9          H ← auxiliary graph
10          M ← maximum weight matching in H
11          Delete all vertices not in a class corresponding to an edge in M
12          D ← vertices deleted in lines 4–7 and 11
13          if ω(D) < ω(X'):
14              X' ← D
15  return X'
```

Figure 4.5: Pseudo-code for CompressCVD

$2.248^k \cdot n^{O(1)}$. Abu-Khzam and Fernau [2006] give a kernelization for unweighted 3-Hitting Set that yields $O(k^3)$ vertices and runs in linear time. Since all this kernelization does is to delete vertices that have to be deleted in any case, it can also be used to kernelize weighted Cluster Vertex Deletion to an $O(k^3)$-vertex kernel.

4.3.2 Iterative compression algorithm

We now describe an algorithm for weighted Cluster Vertex Deletion which improves the running time to $O(2^k \cdot k^9 + n^3)$. The iterative compression framework is the same as for 3-Hitting Set (Figure 4.1): we add vertices one by one to both the graph and the solution set, and keep the solution minimum by using the compression routine. In the rest of this section, we describe the compression routine CompressCVD following the pseudo-code in Figure 4.5. The basic approach is also the same as for 3-Hitting Set: We try by brute force all possibilities to divide the known solution into two parts, one part to keep and one part S to exchange by a smaller part (line 2). We discard partitions where S does not induce a cluster graph (line 3); these cannot lead to a solution, since we determined that none of the vertices in S would be deleted. Note that since we are considering a weighted problem, we cannot bail out as soon as we have



(a) CVD Disjoint Compression instance (b) After Rule 4.1

(c) After Rule 4.2 (d) After Rule 4.3

Figure 4.6: Data reduction in the disjoint compression routine for Cluster Vertex Deletion

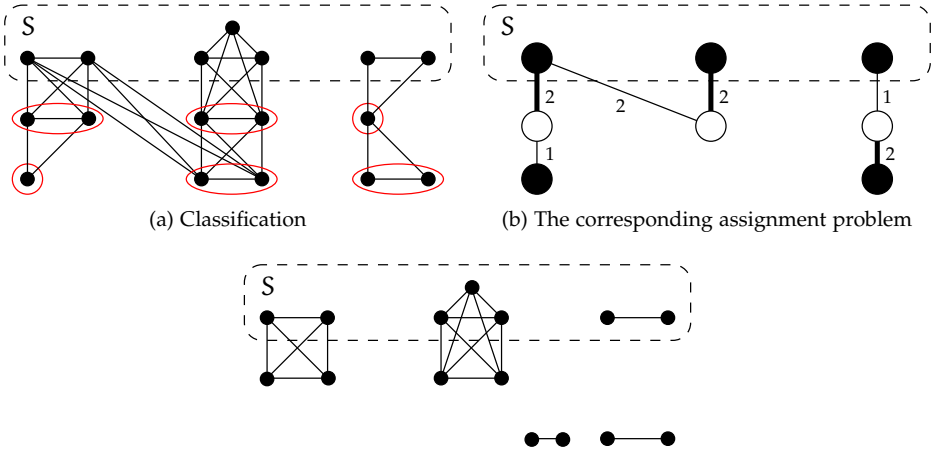
achieved some compression, since there might be several solutions of smaller weight; therefore, we store the currently best found solution in X'. As described in Section 4.2, it remains to find a *disjoint* compression routine, formalized as follows.

CVD Disjoint Compression
Instance: An undirected graph $G = (V, E)$, a vertex weight function $\omega : V \to [1, \infty)$, and a subset $S \subseteq V$ such that $G[S]$ and $G[V \setminus S]$ are cluster graphs.
Task: Find a set $X' \subseteq V \setminus S$ such that $G \setminus X'$ is a cluster graph and $\sum_{v \in X} \omega(x)$ is minimum.

While for 3-Hitting Set, this is done with an exponential-time Vertex Cover algorithm, here we will be able to solve the task in polynomial time (Figure 4.5 lines 4–14). An example instance is shown in Figure 4.6a. We call a connected component in a cluster graph a *cluster*. The instance can now be reduced by a series of data reduction rules. The results are shown in Figures 4.6b–d.

Reduction Rule 4.1. *Delete all vertices in* $R := V \setminus S$ *that are adjacent to more than one cluster in* $G[S]$.

Proof of correctness. If a vertex $v \in R$ is adjacent to vertices u and w in different



(a) Classification

(b) The corresponding assignment problem

(c) Final resulting cluster graph

Figure 4.7: Assignment problem in the iterative compression algorithm for Cluster Vertex Deletion

clusters in S, then uvw induces a P_3, which can only be removed by deleting v. □

Reduction Rule 4.2. *Delete all vertices in R that are adjacent to some, but not all vertices of a cluster in $G[S]$.*

Proof of correctness. If a vertex $v \in R$ is adjacent to a vertex u, but not to a vertex w in a cluster in S, then vuw induces a P_3, which can only be removed by deleting v. □

Reduction Rule 4.3. *Remove connected components that are cliques.*

Proof of correctness. No optimal solution can delete vertices in such components. □

After Rules 4.1–4.3 have been applied, the instance is much simplified (Figure 4.6d). In each cluster in $G[R]$, we can divide the vertices into equivalence classes according to their neighborhood in S; each class then contains either vertices adjacent to all vertices of a particular cluster in $G[S]$, or the vertices adjacent to no vertex in S (see Figure 4.7a). This classification is useful because of the following lemma.

Lemma 4.2. *In an optimal* CVD COMPRESSION *solution, for each cluster in* $G[R]$*, either the vertices of exactly one class are still present, or the whole cluster is deleted.*

Proof. Clearly, it is never useful to delete only some, but not all vertices of a class, since if that led to an optimal solution, we could always re-add the deleted vertices without introducing new P_3's. Further, if $v \in R$ is connected to some $w \in S$, and u is a vertex from the same cluster as v, but from a different class, then uvw is a P_3; therefore, we cannot keep vertices from two different classes within a cluster. □

Because of Lemma 4.2, the remaining task is an assignment of each cluster in $G[R]$ to one of its classes (corresponding to the preservation of this class, and the deletion of all other classes within the cluster) or to nothing (corresponding to the complete deletion of the cluster). However, we cannot do this independently for each cluster; we must not choose two classes from different clusters in $G[R]$ that are connected to the same cluster in $G[S]$, since that would create a P_3. This can be modelled as a bipartite matching problem (also known as assignment problem) in an auxiliary graph H, where each edge corresponds to a possible choice (see Cormen et al. [2001] for an introduction or Lovász and Plummer [1986] for details on matching problems). The graph H is constructed as follows (see Figure 4.7b, where we assume unit weights for the vertices):

- Add a vertex for every cluster in $G[R]$ (white vertices).
- Add a vertex for every cluster in $G[S]$ (black vertices in S).
- For a cluster C_S in $G[S]$ and a cluster C_R in $G[R]$, add an edge between the vertex for C_S and the vertex for C_R if there is a class in C_R connected to C_S. This edge corresponds to choosing this class for C_R and is weighted with the total weight of the vertices in this class.
- Add a vertex for each class in a cluster C_R that is not connected to a cluster in $G[S]$ (black vertices outside S), and connect it to the vertex representing C_R. Again, this edge corresponds to choosing this class for C_R and is weighted with the total weight of the vertices in this class.

Since we only added edges between a black and a white vertex, H is bipartite. The task is now to find a *maximum-weight bipartite matching*, that is, a set of edges of maximum weight where no two edges have an endpoint in common. This allows any choice for a cluster, as long as no two clusters share edges to the same cluster in $G[S]$. The following lemma shows that this is a valid approach:

Lemma 4.3. *A maximum-weight bipartite matching in* H *provides an optimal* CVD COMPRESSION *solution.*

Proof. Each edge in a matching corresponds to a class in a cluster of $G[R]$. The CVD compression solution is to delete all vertices in R but those of the selected classes. The matching cannot select two classes within the same cluster, since the corresponding edges have an endpoint in common; similarly, it cannot select two classes that share a connection to the same cluster in $G[S]$. Therefore, a matching yields a feasible solution. By Lemma 4.2, an optimal CVD compression solution corresponds to an assignment of each cluster to one of its classes or to nothing, and therefore, it corresponds to a matching. Finally, the weight of a matching corresponds to the weight of the vertices not deleted from R, and therefore a maximum-weight matching corresponds to an optimal CVD compression solution. □

Figure 4.7c shows the resulting cluster graph for our example after deleting the vertex sets corresponding to edges that are not selected by the minimum-weight matching shown in Figure 4.7b by bold edges. We can now summarize the findings in the following theorem.

Proposition 4.1. *Weighted* Cluster Vertex Deletion *can be solved in* $O(2^t \cdot n^2(m + n \log n))$ *time.*

Proof. The correctness of the algorithm has been shown above. It remains to bound the running time. We can find clusters in S and R in $O(m)$ time by depth-first search within $G[S]$ and $G[R]$. Rule 4.1 can then be executed in $O(m)$ time. If Rule 4.1 has been applied, Rule 4.2 can be executed in $O(m)$ time by examining the degree of each vertex in R. Finally, we need to find a maximum weight matching in a bipartite graph with at most n vertices and at most m edges, which can be done in $O(n(m + n \log n))$ time [Fredman and Tarjan 1987]. Therefore, we can solve CVD Compression in the same time. The number of vertices in an intermediary solution X to be compressed is bounded by $t + 1$, because any such X consists of an optimal solution for a subgraph of G plus a single vertex. In one compression step, CVD Compression is thus solved $O(2^t)$ times, and there are n compression steps, yielding a total of $O(2^t \cdot n^2(m+n \log n))$ time. □

For the unweighted case, we can get better running times. For each matching instance, we can then use an algorithm for integer weighted matching with a maximum weight of $C = n$ [Gabow and Tarjan 1989], yielding a running time of $O(m\sqrt{n} \log(nC)) = O(m\sqrt{n} \log n)$. Further, as Guo [2006, Section 4.3] points out, in iterative compression, we can save some iteration rounds by starting with a large subgraph for which we can still find in polynomial time a solution of size at most k. The same idea is attributed by Chen et al. [2007a] to Hans L. Bodlaender (Universiteit Utrecht). We call a set of vertices whose deletion produces a cluster graph a *CVD set*. For Cluster Vertex Deletion, we can find

a CVD set of size at most $3k$ by simply repeatedly finding a P_3 and then taking all three of its vertices. We can then start the iteration with G lacking these at most $3k$ vertices and an empty CVD set, after which we will need only $3k$ iteration rounds. This gives the following theorem.

Theorem 4.3. *Unweighted* Cluster Vertex Deletion *can be solved in* $O(2^k \cdot km\sqrt{n}\log n)$ *time.*

Note also that in the unweighted case, we can bail out in line 14 of Figure 4.5 as soon as we have found a CVD set of smaller size, which presumably gives some speedup in practice. We can further avoid the factor of a polynomial of n in the exponential term by kernelization.

Theorem 4.4. *Unweighted* Cluster Vertex Deletion *can be solved in* $O(2^k \cdot k^6 \log k + n^3)$ *time.*

Proof. In $O(n^3)$ time, we apply the kernelization by Abu-Khzam [2007], which gives us an instance with $O(k^2)$ vertices. We then apply Theorem 4.3. □

Curiously, we can use this unweighted algorithm as a subroutine to speed up the weighted case: if we have a solution for an unweighted instance, we can get an optimal weighted solution by executing the compression routine once. This works because the compression does only require that the set X to compress is a CVD set, and does not make any assumptions about its weight.

Theorem 4.5. *Weighted* Cluster Vertex Deletion *can be solved in* $O(2^k \cdot k^9 + n^3)$ *time.*

Proof. We first solve the unweighted problem in $O(2^k \cdot k^6 \log k + n^3)$ time using Theorem 4.4. A single compression takes $O(2^k \cdot n(m + n\log n))$ time. Using the kernelization by Abu-Khzam and Fernau [2006], we can in $O(n^3)$ time first shrink the instance to $O(k^3)$ vertices, giving a time of $O(2^k(k^9 + k^6 \log k)) = O(2^k k^9)$ for the compression. In total, we arrive at the claimed bound. □

Note that the algorithm behind Theorem 4.5 always gives a better bound than Proposition 4.1: first, because of the minimum weight of 1, the parameter k is always less or equal to the parameter t, and second, we can more precisely bound the running time in Theorem 4.5 by $O(2^k \cdot \min\{k^9, n(m + n\log n)\}) + n^3$, because the size of the kernel is bounded by n.

4.3.3 Outlook

The results open up a number of possible lines of research.

Clearly, it is desirable to further improve the running time. It seems difficult to improve the factor of 2^k in Proposition 4.1; if for example the solution X to be compressed already induces a cluster graph, then also any subset of X induces a cluster graph, and we can omit none in the enumeration (Figure 4.3, line 1). However, in practice, it is conceivable that X induces a graph that is not a cluster graph. In that case, we can save some time by directly enumerating only those subsets that induce cluster graphs. For example, if X induces a set of k C_4's (induced cycles of length 4), then only 12^k of the $2^{4k} = 16^k$ subsets induce cluster graphs, a tremendous speedup. To take the maximum advantage of this, it would be desirable to have an enumeration algorithm whose running time is polynomial per subgraph that is generated (Damaschke [2005] provides such an algorithm for Cluster Editing). It would require some experiments to see whether this improvement is worthwhile in practice.

A further bottleneck is the matching routine. Unfortunately, it seems difficult to replace it with a simpler approach, since it is not hard to see that the general bipartite matching problem can be reduced to the subproblem encountered. However, a speedup here might be attainable by exploiting the similarity of the subproblems solved; this worked for Edge Bipartization (Section 4.5.2).

It is further open to improve the trivial factor-3 approximation. Cai et al. [2001] improved the approximation factor for Feedback Vertex Set in tournaments, which is also characterized by a 3-vertex forbidden subgraph, from 3 to 2.5; perhaps similar techniques are applicable here.

Finally, for practical applications it is desirable to improve the size of the kernel. Possibly, some ideas of the kernels for Cluster Editing [Protti et al. 2006, Fellows et al. 2007b, Guo 2007] could be adapted.

4.4 Iterative compression for Feedback Vertex Set in tournaments

In this section, we show how to solve weighted Feedback Vertex Set in tournaments using iterative compression, which yields the fastest currently known algorithm for this problem. This is the first application of iterative compression to a problem on *directed* graphs (see also Gutin and Yeo [2007] for a survey on parameterized problems on directed graphs). The question whether Feedback Vertex Set on general directed graphs is fixed-parameter tractable had been famously open for a long time and has only recently been resolved positively, also using iterative compression [Chen et al. 2008]; however, the given algorithm incurs a much worse combinatorial explosion with respect to the parameter k than those specialized to tournaments.

The Feedback Vertex Set problem asks for a minimum number of vertices to delete from a graph to make it acyclic. It was one of the 21 problems for which NP-hardness was first shown by reduction [Karp 1972]. Due to applications such as voting systems [Charon and Hudry 2007] and rank aggregation [Ailon et al. 2005a], the class of *tournaments* has received particular interest. A tournament is an orientation of a complete undirected graph, or equivalently, it is a directed graph where between any two distinct vertices there is exactly one arc. Thus, the central problem of this section is defined as follows:

Feedback Vertex Set in tournaments (FVST)
Instance: A tournament $T = (V, A)$ and an integer $k \geqslant 0$.
Question: Can T be transformed by up to k vertex deletions into a directed acyclic graph?

4.4.1 Known results on Feedback Vertex Set in tournaments

FVST is NP-hard [Speckenmeyer 1989]. The following well-known lemma allows a different characterization of FVST.

Lemma 4.4. *A tournament is acyclic iff it contains no* triangles, *that is, directed cycles of length* 3.

Proof. Clearly, an acyclic graph contains no triangle; it remains to show that a tournament that contains a cycle also contains a triangle. Consider for a contradiction a shortest cycle $v_1, v_2, \ldots, v_c, v_1$ with $c > 3$ in a tournament $T = (V, A)$. If $(v_c, v_2) \in A$, then $v_2, \ldots, v_c, v_2$ is a shorter cycle, which is a contradiction. Otherwise, $(v_2, v_c) \in A$, and we have a triangle $v_1 v_2 v_c$, which is also a contradiction. □

By Lemma 4.4, like with Cluster Vertex Deletion, we can think of FVST as a special case of 3-Hitting Set: find a set of vertices which hits each triangle. This easily yields a 3-approximation: while there is a triangle, delete all three of its vertices. The approximation factor has been improved to 2.5 [Cai et al. 2001]. An approximation-preserving reduction from Vertex Cover to FVST [Speckenmeyer 1989] together with the inapproximability result for Vertex Cover [Dinur and Safra 2005] shows that it is NP-hard to approximate FVST better than by a factor of 1.36.

The approximation hardness results make fixed-parameter algorithms with the natural parameter k attractive. The hitting set characterization immediately gives an $O(3^k)$-size search tree. Raman and Saurabh [2006] have given the first nontrivial result by giving fixed-parameter algorithms for weighted FVST with real weights $\geqslant 1$ running in $O(2.415^k \cdot n^{2.376})$ time, where k is the total weight of

the removed vertices. For the unweighted case of FVST, the previously fastest parameterized algorithm is obtained by an elaborate 3-Hitting Set algorithm and runs in $O(2.076^k + n^3)$ time [Wahlström 2007]. Using Lemma 4.4, in the same way as for Cluster Vertex Deletion, we can obtain in $O(n^3)$ time a kernel of $O(k^2)$ vertices for Feedback Vertex Set in tournaments by using the kernelization for 3-Hitting Set [Abu-Khzam 2007].

The related problem Feedback Arc Set in tournaments (FAST), where we ask for a minimum number of arcs to delete to make a tournament acyclic, has also been considered frequently. Its NP-hardness was open for a long time and has only recently been proved. Alon [2006] gave a randomized reduction, which was independently derandomized by Ailon et al. [2005a] and Charbit et al. [2007]. Also independently, Conitzer [2006] gave a deterministic reduction from MaxSAT. FAST is easier to approximate than FVST: there is a polynomial-time approximation scheme (PTAS) [Kenyon-Mathieu and Schudy 2007], meaning that for any any fixed approximation factor, a polynomial-time approximation algorithm can be given. Like for FVST, Raman and Saurabh [2006] gave a fixed-parameter algorithm running in $O(2.415^k \cdot n^{2.376})$ time. It is also easy to show that FAST has a kernel of $O(k^2)$ vertices [Dom et al. 2006b]. Raman et al. [2007] provided an exact (not parameterized with respect to k) algorithm solving FAST in $O(1.555^m)$ time.

We improve the time bound of exactly solving weighted FVST to $O(2^k \cdot n^2(\log n + k))$. This also demonstrates the applicability of the elegant iterative compression method in contrast to the more standard case-distinction based search tree approaches employed by Raman and Saurabh [2006] and Wahlström [2007]. Further, this allows us to give an exact (not parameterized) algorithm for FVST running in $O(1.709^n)$ time, answering a question of Woeginger [2008].

4.4.2 Iterative compression algorithm

We use the same overall scheme as for 3-Hitting Set (Figure 4.1); this works since we are dealing with a vertex deletion problem for a hereditary graph class, and thus have the desired monotonicity of solution size with respect to adding vertices.

Compression Routine. To make the task of looking for a smaller feedback vertex set for a tournament T easier, we would like to restrict our search to feedback vertex sets that are disjoint from a given one. This is the same approach as used in Section 4.3 and all other iterative compression algorithms. We can achieve this in the same way as for 3-Hitting Set (see Figure 4.3): by a brute-force enumeration of all $O(2^k)$ possibilities to partition the given feedback vertex set X into two vertex sets S and $X \setminus S$. For each partition, we then look only for

solutions that contain all of $X \setminus S$ (they can immediately be deleted from the tournament), but none of S. Further, we can omit all partitions where $T[S]$ is not cycle-free, since we determined none of the vertices in S would be deleted. Therefore, all that remains is to deal with the following problem.

FVST Disjoint Compression
Instance: A tournament $T = (V, A)$ and a subset $S \subseteq V$ such that $T[S]$ and $T[V \setminus S]$ are acyclic.
Task: Find a set $S' \subseteq V \setminus S$ with $|S'| < |S|$ such that $T \setminus S'$ is acyclic.

Up to this point, the algorithm is analogous to the iterative compression algorithm for undirected Feedback Vertex Set [Dehne et al. 2007, Guo et al. 2006]. The core part of the compression routine, however, is completely different; in particular, we will be able to solve the remaining task of finding a smaller feedback vertex set that is disjoint from the given one S in polynomial time, whereas Dehne et al. [2007] and Guo et al. [2006] still require exponential time.

Consider a FVST Disjoint Compression instance (T, S). As mentioned, both $T[S]$ and $T[V \setminus S]$ are acyclic and thus have a topological sort (note that the topological sort of a tournament is unique). Then, the topological sort of a maximum acyclic subtournament of T containing all of S can be thought of as resulting from inserting a subset of $V \setminus S$ into the topological sort of S. On the one hand, the order of the inserted subset must not violate the topological sort of $T[V \setminus S]$. On the other hand, we can achieve by a data reduction rule that for every $v \in V \setminus S$, the subtournament $T[S \cup \{v\}]$ is acyclic and therefore v has a "natural" position within the topological sort of S. We then obtain the maximum acyclic subtournament as the longest common subsequence of the topological sort of $T[V \setminus S]$ and $V \setminus S$ sorted by natural position within S.

We describe this in more detail using the subroutine displayed in Figure 4.8. First we apply data reduction to the instance: whenever there is a triangle with two vertices in S, we can only get rid of this triangle by deleting the third vertex (lines 3–5). After applying this reduction rule exhaustively, for any $v \in V \setminus S$ the subtournament $T[S \cup \{v\}]$ clearly does not contain triangles anymore and therefore is acyclic by Lemma 4.4. This means that we can insert v at some point in the topological sort $s_1, \ldots, s_{|S|}$ of S without introducing back arcs (that is, arcs pointing from a higher indexed vertex to a lower indexed vertex in the sort). Since T is a tournament, there is thus some integer $p[v]$ such that for $i < p[v]$, there is an arc from s_i to v, and for $i \geqslant p[v]$, there is an arc from v to s_i (Figure 4.9):

$$(v, s_i) \in A \iff i \geqslant p[v]. \tag{4.1}$$

Input: Tournament $T = (V, A)$ and a feedback vertex set S for T.
Output: A minimum feedback vertex set F for T with $F \cap S = \emptyset$.

```
1  s_1, ..., s_|S| ← topological sort of T[S]
2  R ← ∅
3  while there is a triangle u, v, w with u, v ∈ S and w ∈ V \ S:
4      R ← R ∪ {w}
5      T ← T − w
6  for each v ∈ V \ S:
7      p[v] ← min({i | (v, s_i) ∈ A} ∪ {|S| + 1})
8  L ← topological sort of T[V \ S]
9  P ← V \ S sorted by p, with position in L as tie-breaker
10 Y ← vertices in a longest common subsequence of L and P
11 return R ∪ ((V \ S) \ Y)
```

Figure 4.8: Algorithm for FVST Disjoint Compression

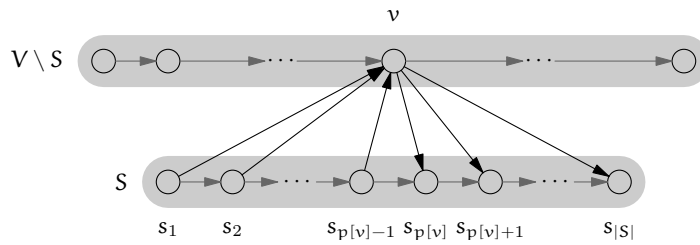


Figure 4.9: Illustration of equivalence (4.1). For clarity, only some of the arcs within the acyclic subtournaments $T[S]$ and $T[V \setminus S]$ are shown.

We calculate p in lines 6–7: when we encounter the first s_i in the topological sort of S where $(v, s_i) \in A$, we can insert v before s_i; if there is no such s_i, we set $p[v]$ to $|S| + 1$, and (4.1) still holds.

We now construct a sequence P from p (line 9), where vertices from $V \setminus S$ that are positioned by p between the same two vertices of S are ordered according to their relative position in the topological sort of $T[V \setminus S]$. Clearly, any acyclic subtournament of T containing all of S must have a topological sort where the vertices from $V \setminus S$ occur in the same order as in P. The same holds for the topological sort L of $T[V \setminus S]$, which is calculated in line 8. This leads to the following lemma.

Lemma 4.5. *After line 9 of the algorithm in Figure 4.8,* T *is acyclic iff the sequences* L *and* P *are equal.*

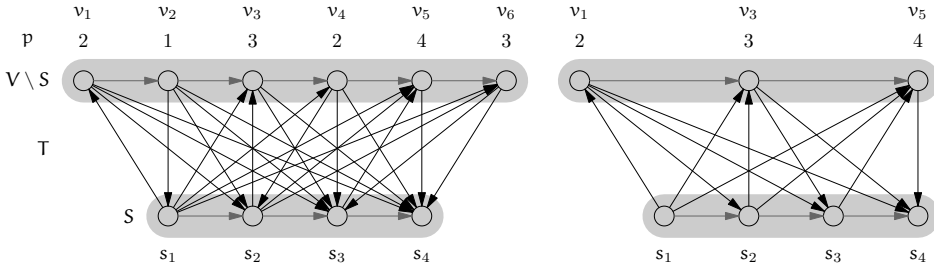


Figure 4.10: Example for the subroutine in Figure 4.8. For clarity, only some of the arcs within the acyclic subtournaments $T[S]$ and $T[V \setminus S]$ are shown. Left: Tournament T after data reduction with $L = v_1, v_2, v_3, v_4, v_5, v_6$ and $P = v_2, v_1, v_4, v_3, v_6, v_5$. A longest common subsequence is v_1, v_3, v_5, yielding the acyclic graph shown on the right.

Proof. "⇒": Consider the case that for all $v, w \in V \setminus S$ with $(v, w) \in A$ we have $p[v] \leqslant p[w]$. Then v occurs in L before w, because L is a topological sort, and by definition we also have that v occurs before w in P. Therefore, the relative order of any two vertices is the same in L and P, and L and P are equal.

Thus, if L and P are not equal, then there are $v, w \in V \setminus S$ with $(v, w) \in A$ but $p[v] > p[w]$. Then by (4.1) we have $(w, s_{p[w]}) \in A$ and $(v, s_{p[w]}) \notin A \Rightarrow (s_{p[w]}, v) \in A$, and T contains the cycle $v, w, s_{p[w]}$.
"⇐": By Lemma 4.4, it suffices to look for triangles to decide whether T is acyclic. Since $T[S]$ and $T[V \setminus S]$ are acyclic and we destroyed all triangles with two vertices in S, there can only be triangles with exactly two vertices in $V \setminus S$. If L and P are equal, then for all $v, w \in V \setminus S$ with $(v, w) \in A$ we have $p[v] \leqslant p[w]$. Then by (4.1), there cannot be any s_i with $(w, s_i) \in A$ and $(s_i, v) \in A$, and there can be no triangle in T. □

With the same justification, Lemma 4.5 holds for induced subgraphs of T and the corresponding sequences L and P. Clearly, deleting a vertex $v \in V \setminus S$ from T affects L and P only insofar as v disappears from L and P. Therefore, the cheapest way to make T acyclic by vertex deletions can be obtained by finding the cheapest way to make L and P equal by vertex deletions; this is exactly the complement of the longest common subsequence of L and P. We then obtain the desired feedback vertex set for T by adding the vertices of this complement to those of R, since R contains the vertices that were determined to be in any feedback vertex set in the reduction step (lines 10–11). Figure 4.10 shows an example for the execution of the subroutine from Figure 4.8.

In summary, the subroutine from Figure 4.8 is correct and can be used to solve Feedback Vertex Set in tournaments by iterative compression as described at the beginning of this section.

Theorem 4.6. *Using iterative compression,* Feedback Vertex Set *in tournaments can be solved in* $O(2^k \cdot n^2(\log n + k))$ *time.*

Proof. We have shown how to solve Feedback Vertex Set in tournaments using iterative compression. It remains to analyze the running time. First we examine the subroutine from Figure 4.8. Topological sort (line 1) can be easily done in $O(|S|) = O(k)$ time. Finding triangles in line 3 can be done in $O(nk)$ time: for every $v \in V \setminus S$, we iterate over the topological sort of S; if we encounter a vertex s_i with $(v, s_i) \in A$ and later a vertex s_j with $(s_j, v) \in A$, we have a triangle as desired. Line 8 can be done in $O(n)$ time and line 9 in $O(n \log n)$ time. Since L and P are permutations of each other, finding a longest common subsequence reduces to finding a longest increasing subsequence, which can be done in $O(n \log n)$ time [Fredman 1975]. In summary, the subroutine can be executed in $O(n(\log n + k))$ time. In the compression routine, the subroutine is called $O(2^k)$ times, once for each partition of X into two subsets. The compression routine itself is called n times when inductively building up the graph structure. In total, we have a running time of $O(2^k \cdot n^2(\log n + k))$. □

We note in passing that by using a more careful analysis and elaborate data structures for the longest increasing subsequence problem [Hunt and Szymanski 1977], we could replace the $\log n$ term by $\log \log n$. Further, using the $O(k^2)$-vertex 3-Hitting Set kernelization [Abu-Khzam 2007], we arrive at the following running time.

Theorem 4.7. *Using iterative compression,* Feedback Vertex Set *in tournaments can be solved in* $O(2^k \cdot k^5 + n^3)$ *time.*

We now show how to extend our results to the weighted case with real weights:

> Weighted Feedback Vertex Set in tournaments
> **Instance:** A tournament $T = (V, A)$, a vertex weight function $\omega : V \to [1, \infty)$, and a number $t \geqslant 0$.
> **Question:** Is there a subset $X \subseteq V$ with $\sum_{v \in X} \omega(x) \leqslant t$ such that deleting all vertices in X from G results in a directed acyclic graph?

Note that for arbitary weights, the problem is not fixed-parameter tractable unless P = NP, since otherwise we could solve FVST in polynomial time by scaling down the weights sufficiently.

We modify our algorithm only in the last iteration of the iterative compression, where we have a feedback vertex set X of size at most $k+1$ for T. Clearly, we can still enumerate all $O(2^k)$ possibilities of which part S to keep and which part to omit from X to get a minimum-weight solution X'. The data reduction (Figure 4.8 lines 3–5) is also still correct, and Lemma 4.5 holds. Therefore, again, the cheapest way to make T acyclic by vertex deletions can be obtained by finding the cheapest way to make L and P equal by vertex deletions; therefore, we need a *minimum-weight* common subsequence of L and P. Since L and P are permutations of each other, this reduces to finding a minimum-weight increasing subsequence, which in turn reduces to finding a minimum-weight independent set in a permutation graph. This can be done in $O(n \log\log n)$ time [Chang and Wang 1992]. Further, since a weighted optimal solution needs at least as many vertices as an unweighted optimal solution, and each vertex weighs at least 1, we have $t \geqslant k$. We arrive at the following result.

Theorem 4.8. *Weighted* Feedback Vertex Set *in tournaments can be solved in* $O(2^t \cdot n^2(\log n + t))$ *time.*

Woeginger [2008] noted that by combining an algorithm by Schwikowski and Speckenmeyer [2002] that enumerates all (inclusion-)minimal feedback vertex sets in a directed graph with polynomial delay with the fact that a tournament has at most 1.717^n minimal feedback vertex sets [Moon 1971], one obtains an algorithm that solves FVST in $O(1.717^n)$ time. He asks whether this bound can be improved. Raman et al. [2007] pointed out that one can sometimes gain fast exact algorithms by using an FPT algorithm for small parameter values and brute force only for large parameter values. This approach can be applied here. We try all possible parameter values $k = 0, \ldots, n$; if $k \leqslant \lambda n$, we use the $2^k \cdot n^{O(1)}$ algorithm of Theorem 4.6, and otherwise, we try by brute force all $\binom{n}{k}$ possible solutions. The running time of the brute force approach is maximum for $\lambda = 1/2$, since $\binom{n}{n/2} \approx 2^n$; therefore, we can improve the trivial 2^n bound if $\lambda > 1/2$. The optimal λ is attained when $2^{\lambda n} = \binom{n}{\lambda n}$, which gives (asymptotically) $\lambda \approx 0.773$. Thus, we can answer Woeginger's question affirmatively.

Theorem 4.9. Feedback Vertex Set *in tournaments can be solved in* $O(1.709^n)$.

4.4.3 Outlook

Like for Cluster Vertex Deletion, it seems hard to improve the factor of 2^k in Theorem 4.6. It might happen that X already induces an acyclic graph, in which case we can never bail out early. However, in this or similar cases where we cannot save much, we have an *almost* optimal feedback arc set that is acyclic.

A speedup in the disjoint compression step might also be attainable by exploiting the similarity of the subproblems solved; this worked for Edge Bipartization (Section 4.5.2).

Of particular interest would be an iterative compression algorithm for Feedback Arc Set in tournaments. Since here also we get an $O(2.076^k + m)$ time algorithm by reduction to 3-Hitting Set [Wahlström 2007], we would probably need a polynomial-time compression routine to improve the running time bound.

A further area of application might be feedback set problems on *bipartite* tournaments. The Feedback Vertex Set problem on bipartite tournaments is NP-hard and can be approximated within a factor of 3.5 [Cai et al. 2002]. With a reduction to 4-Hitting Set and Theorem 4.2, we get a running time of $O(3.076^k + n^4)$; Sasatte [2007] gave an algorithm with running time $O(3^k n^2 + n^3)$ based on branching. Feedback Arc Set in bipartite tournaments was also recently shown to be NP-hard [Guo et al. 2007b], and can be approximated within a factor of 4 [Gupta 2008]; further, an FPT algorithm running in $(3.373^k \cdot n^6)$ time is known [Dom et al. 2006b]. For both problems, it would be interesting to see whether we can improve the running time using the techniques of this section.

4.5 Iterative compression for Edge Bipartization

In this section, we show how to solve the Edge Bipartization problem, which was introduced in Section 2.3.1, by iterative compression. Unlike for Cluster Vertex Deletion and Feedback Vertex Set in tournaments, where fixed-parameter tractability was already known from the finite forbidden subgraph characterization and thus the existence of a simple search tree algorithm, this is the only known FPT algorithm for Edge Bipartization. We can also obtain fixed-parameter tractability by a reduction from Vertex Bipartization [Wernicke 2003]; however, the reduction results in a running time of $O(3^k \cdot k^3 m^2 n)$, while our algorithm runs in $O(2^k \cdot m^2)$ time.

Our Edge Bipartization algorithm can be generalized to solve the Balanced Subgraph problem (Section 4.5.4). As we demonstrate in Section 3.2.4 by experiments, the resulting algorithm is quite fast and even competitive with polynomial-time approximations, while still providing optimal results.

The different characterizations of bipartite graphs (Lemma 1.1) lead to the following equivalent reformulations of Edge Bipartization:

1. Can we find a partition of V into V_1 and V_2 such that the number of edges within V_1 and within V_2 together is at most k (that is, $|E(G[V_1])| + |E(G[V_2])| \leqslant k$)?

```
IterativeCompressionEB(G = (V, E))
1   E' ← ∅
2   X ← ∅
3   for each e ∈ E:
4       E' ← E' ∪ {e}
5       if X is not a bipartization set for (V, E'):
6           X ← X ∪ {e}
7           X ← CompressEB((V, E'), X)
8   return X
```

Figure 4.11: Pseudo-code for iterative compression for Edge Bipartization

2. Can we find a coloring of V with two colors such that at most k edges are between vertices of the same color?

3. Can we find a set E' of at most k edges such that each odd cycle contains at least one edge in E'?

We will use all of these interchangeably.

A large number of results is known on Edge Bipartization; these are surveyed in Section 2.3.1.

4.5.1 Iterative compression algorithm

For Cluster Vertex Deletion and Feedback Vertex Set in tournaments, the task is to delete a set of vertices to achieve a certain property. Here, in contrast, we want to delete *edges*. The iterative compression approach can be easily adapted to this, though: we simply add edges one-by-one, while keeping the solution set minimum using the compression routine, as illustrated in the pseudo-code in Figure 4.11.

Here, the loop invariant is that X is a minimum edge bipartization set for (V, E'). We start with $E' = \emptyset$; since (V, E') then consists only of isolated vertices, $X = \emptyset$ is a minimum edge bipartization set. In line 4, we add one edge $e \notin E'$ from E to E'. If we are lucky, then X is still a bipartization set for the grown instance. Otherwise, we add e to X (line 6). Then X is again an edge bipartization set for (V, E'), although possibly not a minimum one. The compression routine CompressEB takes a graph G and an edge bipartization set X for G, and returns a smaller edge bipartization set for G if there is one; otherwise, it returns X unchanged. Therefore, after compression, our loop

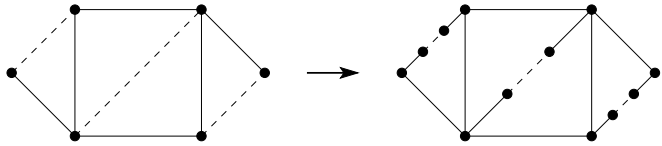


Figure 4.12: Illustration of Graph Transformation 4.1. *Dashed edges* are edges in the edge bipartization set.

invariant still holds. Since eventually $(V, E') = G$, we obtain an optimal solution. It remains to derive the compression routine CompressEB.

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

Lemma 4.6. *Given a graph* $G = (V, E)$ *and an inclusion-minimal edge bipartization set* X *for* G*, the following two properties hold:*

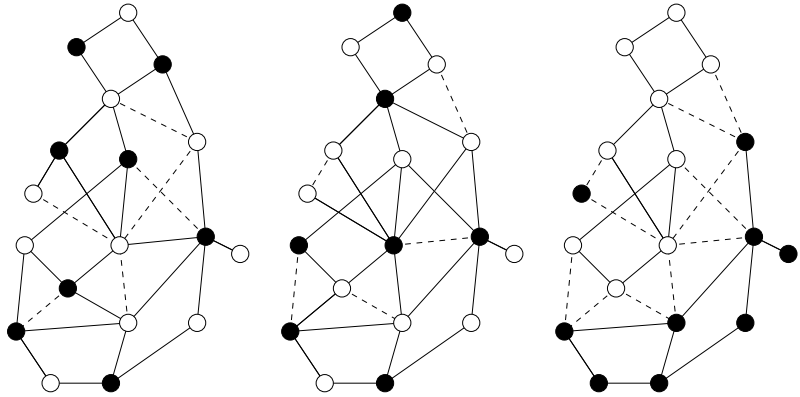
1. *For every odd-length cycle* C *in* G, $|E(C) \cap X|$ *is odd.*
2. *For every even-length cycle* C *in* G, $|E(C) \cap X|$ *is even.*

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

As for Cluster Vertex Deletion and Feedback Vertex Set in tournaments, it is helpful to assume that an edge bipartization set that is smaller than a given edge bipartization set X is disjoint from X. For the mentioned two problems, we achieved this by a brute force case enumeration. Here, in contrast, we can assume this property without loss of generality by applying a simple input transformation (see Figure 4.12): we subdivide each edge that was part of the edge bipartization set by two vertices, and the middle segment of each subdivided edge into the new edge bipartization set. More formally:

Graph Transformation 4.1. *Given a graph* $G = (V, E)$ *and an edge bipartization set* X*, construct* $G' = (V', E')$ *with* $V' := V \cup \{e_1, e_2 \mid e \in X\}$ *and* $E' := E \setminus X \cup \{\{v, e_1\}, \{e_1, e_2\}, \{e_2, w\} \mid e = \{v, w\} \in X\}$*. Let further* $X' := \{\{e_1, e_2\} \mid e \in X\}$*.*

Lemma 4.7. *After Graph Transformation 4.1, the transformed graph has an edge bipartization set with* i *edges iff the original graph has an edge bipartization set with*



(a) Two-coloring of $G \setminus X$ (b) Two-coloring of $G \setminus Y$ (c) Comparison coloring

Figure 4.13: Comparing disjoint edge bipartization sets

i *edges. Moreover, for each minimal edge bipartization set* X *for the transformed graph there is an edge bipartization set of the same size that is disjoint from* X*.*

Proof. The first point is easy to see, since the transformation preserves the parities of the lengths of all cycles. Regarding the second point, if we have a bipartization set Y for the transformed graph that also contains edges from X, we can obtain an edge bipartization set of the same size by replacing every edge in $Y \cap X$ by any of its two adjacent edges. □

Thus, we have achieved the desired disjointness property in polynomial time, whereas for Cluster Vertex Deletion and Feedback Vertex Set in tournaments, this was the source of the exponential running time. Since we cannot expect overall polynomial time for this NP-hard problem, the compression routine will be exponential-time, in contrast to the other two problems, where it was polynomial.

The idea for the compression routine is to compare the two-colorings induced by the known bipartization set X (Figure 4.13a) and the (yet unknown) compressed solution Y (Figure 4.13b) and mark a vertex black when the two colorings coincide, or white when they differ (Figure 4.13c). The key observation is then that the two bipartization sets together form an edge cut between the black and the white vertices, that is, removing them destroys all paths from a black to a white vertex.

The following simple definition is the only remaining prerequisite for the central lemma for the Edge Bipartization compression routine, which formalizes

this observation.

Definition 4.3. *Let* $G = (V, E)$ *be a graph and* $X \subseteq E$. *Then,* $V(X)$ *denotes the set* $\bigcup_{\{u,v\}\in X}\{u, v\}$ *of their endpoints. A mapping* $\Phi : V(X) \to \{\bigcirc, \bullet\}$ *is called* valid partition of $V(X)$ *if for each* $\{u, v\} \in X$, *we have* $\Phi(u) \neq \Phi(v)$.

Lemma 4.8. *Consider a graph* $G = (V, E)$ *and a minimal edge bipartization set* X *for* G. *For a set of edges* $Y \subseteq E$ *with* $X \cap Y = \emptyset$, *the following are equivalent:*

(1) Y *is an edge bipartization set for* G.

(2) There is a valid partition Φ *of* $V(X)$ *such that* Y *is an edge cut in* $G \setminus X$ *between* $\bigcirc_\Phi := \Phi^{-1}(\bigcirc)$ *and* $\bullet_\Phi := \Phi^{-1}(\bullet)$.

Proof. (2) ⇒ (1): Consider any odd-length cycle C in G. It suffices to show that $E(C) \cap Y \neq \emptyset$. Let $s := |E(C) \cap X|$. By Property (1) in Lemma 4.6, s is odd. Let the edges in $E(C) \cap X$ be $\{\{u_0, v_0\}, \{u_1, v_1\}, \ldots, \{u_{s-1}, v_{s-1}\}\}$ such that vertices v_i and $u_{(i+1) \bmod s}$ are connected by a path in $C \setminus X$. Since Φ is a valid partition of $V(X)$, we have $\Phi(u_i) \neq \Phi(v_i)$ for all $0 \leqslant i < s$. 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 Y destroys all paths in $G \setminus X$ between $\bigcirc_\Phi$ and $\bullet_\Phi$, we obtain that $E(C) \cap Y \neq \emptyset$.

(1) ⇒ (2): Let $C_X : V \to \{\bigcirc, \bullet\}$ be a two-coloring of the bipartite graph $G \setminus X$ and $C_Y : V \to \{\bigcirc, \bullet\}$ a two-coloring of the bipartite graph $G \setminus Y$. Define

$$\Phi : V \to \{\bigcirc, \bullet\},\ v \mapsto \begin{cases} \bigcirc & \text{if } C_X(v) = C_Y(v), \\ \bullet & \text{otherwise.} \end{cases} \tag{4.2}$$

We show that $\Phi|_{V(X)}$ (that is, Φ with domain restricted to $V(X)$) is a valid partition with the desired property (see Figure 4.13c for an example).

First we show that $\Phi|_{V(X)}$ is a valid partition. Consider any edge $\{u, v\} \in X$. There must be at least one even-length path in $G \setminus X$ from u to v; otherwise, $\{u, v\}$ would be redundant as X would not be minimal. Therefore, $C_X(u) = C_X(v)$. In $G \setminus Y$, the vertices u and v are connected by an edge, and therefore $C_Y(u) \neq C_Y(v)$. It follows that $\Phi(u) \neq \Phi(v)$.

Since both C_X and C_Y change in value when going from a vertex to its neighbor in $G \setminus (X \cup Y)$, the value of Φ is constant along any path in $G \setminus (X \cup Y)$. Therefore, there can be no path from any $u \in \bigcirc_\Phi$ to any $v \in \bullet_\Phi$ in $G \setminus (X \cup Y)$, that is, Y is an edge cut between $\bigcirc_\Phi$ and $\bullet_\Phi$ in $G \setminus X$. □

Figure 4.14 shows how to make use of Lemma 4.8 to obtain the compression routine that, given a graph and a (with respect to set inclusion) minimal edge

```
CompressEB(G, X)
1   G ← EdgeExpand(G, X)
2   for each valid partition Φ of V(X):
3       Y ←MinCut(G, ○_Φ, ●_Φ)
4       if |Y| < |X|:
5           return Y
6   return X
```

Figure 4.14: Pseudo-code for CompressEB

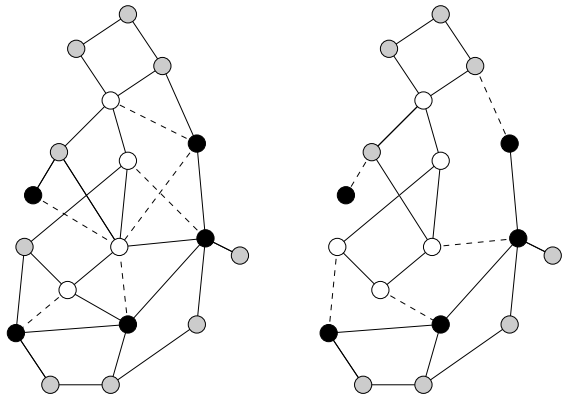


Figure 4.15: A valid partition leading to a compressed solution (black and white: value of valid partition; grey: not in domain of valid partition).

bipartization set X of size k, either computes a smaller edge bipartization set Y in $O(2^k \cdot km)$ time or proves that no such Y exists. First, we apply the input transformation from Figure 4.12, which allows us to assume the prerequisite of Lemma 4.8 that $Y \cap X = \emptyset$ (line 1). We then enumerate all 2^k valid partitions Φ of $V(X)$ (line 2) and determine a minimum-size edge cut between $\bigcirc_\Phi$ and $\bullet_\Phi$ (line 3) until we find an edge cut Y of size $k-1$ (line 4). This is illustrated in Figure 4.15: In Figure 4.15a, we have a graph with an edge bipartization set X (dashed lines) and a particular valid partition for $V(X)$. A smaller edge bipartization set is obtained as minimum-size edge cut between the black and white vertices (Figure 4.15b), which is a polynomial-time task. By Lemma 4.8, Y is an edge bipartization set; furthermore, if no such Y is found, we know that k

is of minimum size.

Theorem 4.10. Edge Bipartization *can be solved in* $O(2^k \cdot km^2)$ *time.*

Proof. The correctness of the algorithm comprising IterativeCompressionEB and CompressEB has been argued for. It remains to analyze the running time. The argument X of CompressEB has size at most $k+1$, so there are at most $2^{k+1} = O(2^k)$ valid partitions to check in each call. Each of the MinCut instances can individually be solved in $O(km)$ time with the Edmonds–Karp algorithm [Dinic 1970, Edmonds and Karp 1972] that goes through at most $k+1$ rounds, each time finding a shortest flow augmenting path by breadth-first search in $O(m)$ time. In IterativeCompressionEB, there are m calls to CompressEB, so in total we obtain the claimed running time. □

Like for Cluster Vertex Deletion (Theorem 4.3), we can use an approximation to reduce the number of iterations. Guo [2006] suggests to start with a spanning tree, which improves the number of rounds from m to $m - n$. Other possibilities are to use the approximation by Agarwal et al. [2005], which yields $O(\sqrt{\log n})$ rounds, or the approximation by Avidor and Langberg [2007], which yields $O(k \log k)$ rounds. However, all of these introduce an additional polynomial cost for the approximation. It depends on the concrete parameters which of these approaches is the fastest.

4.5.2 Exploiting subproblem similarity

In this section, we show how to save a factor of k in the running time claimed in Theorem 4.10 by exploiting the similarity of the minimum cut subproblems to be solved. To be able to describe the improvement, we first need to go into the details of the MinCut algorithm used in CompressEB (Figure 4.14).

Network flow. Each minimum cut subproblem is solved using network flow. We briefly recall the basics of network flow (for a more detailed introduction, see e. g. Cormen et al. [2001] or the monograph on network flows by Ahuja et al. [1993]). A *flow network* is a directed graph $F = (V, A)$ in which each arc $(u, v) \in A$ has a *capacity* $c(u, v) \geqslant 0$ (for convenience, $c(u, v) := 0$ for $(u, v) \notin A$), and that has two distinguished vertices, the *source* s and the *sink* t. A *flow* is a function $f : V \times V \to \mathbb{Q}$ with the following properties:

$$\forall u, v \in V : f(u,v) \leqslant c(u,v) \qquad \text{(capacity constraint)} \tag{4.3}$$

$$\forall u, v \in V : f(u,v) = -f(v,u) \qquad \text{(skew symmetry)} \tag{4.4}$$

$$\forall u \in V \setminus \{s,t\} : \sum_{v \in V} f(u,v) = 0 \qquad \text{(flow conservation).} \tag{4.5}$$

The *value* of a flow f is defined as

$$|f| := \sum_{v \in V} f(s,v). \tag{4.6}$$

A maximum flow is simply a flow of maximum value. The well-known max-flow min-cut theorem [Elias et al. 1956, Fulkerson and Ford 1956] tells us that in a directed graph, the size of a minimum cut is equal to the value of a maximum flow and that moreover we can retrieve a minimum cut from a maximum flow. To apply this to the task at hand (finding MinCut$(G, \circ_\Phi, \bullet_\Phi)$ in line 3 of Figure 4.14), we first convert the undirected graph to a directed graph by replacing each undirected edge $\{u,v\}$ with two arcs (u,v) and (v,u). By the max-flow min-cut theorem, for every minimum cut there is a maximum flow where the arcs of the cut have positive flow. Therefore, not both of (u,v) and (v,u) can be part of a minimum cut, since that would imply a violation of the skew symmetry in the corresponding maximum flow network. Thus, there is a simple bijection between minimum cuts of the directed and the undirected graph. Further, in our setting we have more than one source and more than one target. Therefore, we use the standard technique of adding an extra vertex s with arcs to each *start vertex* (vertex in $\circ_\Phi$) and an extra vertex t with arcs from each *target vertex* (vertex in $\bullet_\Phi$) to t. All arcs in the flow network have unit capacity. We now have a flow network whose maximum flow will allow us to find MinCut$(G, \circ_\Phi, \bullet_\Phi)$.

To find the maximum flow, we use the Edmonds–Karp algorithm [Dinic 1970, Edmonds and Karp 1972], which is a special case of the Ford–Fulkerson method [Fulkerson and Ford 1956]. This is not the fastest maximum flow algorithm in general, but it is fast when the value of the flow is small, as it is the case here where the value of the maximum flow is bounded by $k := |\circ_\Phi| = |\bullet_\Phi|$.

The Ford–Fulkerson method works with the *residual network*, which consists of the arcs that can admit more flow, that is, arcs (u,v) for which $f(u,v) < c(u,v)$. Note that in particular if there is flow from u to v, then $f(v,u)$ is negative and (v,u) is part of the residual network. The method works by repeatedly finding an *augmenting path*, which is a path from s to t in the residual network. It then increases the flow along each arc of the path by the maximum *residual capacity* (that is, the maximum value of $c(u,v) - f(u,v)$ for an arc (u,v) of

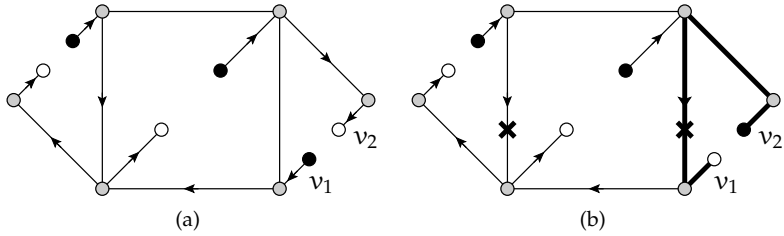


Figure 4.16: Example for reusing flow networks

the path), which will give a flow consistent with (4.3)–(4.5) and a higher flow value. In our case, the increase is always by at least 1, since initially all residual capacities are 1, and they stay integral after increasing the flow. Therefore, we need at most k flow augmentations. It remains to specify how to find an augmenting path. We use breadth-first search in $O(m)$ time, which yields the Edmonds–Karp algorithm. Therefore, each subproblem takes $O(km)$ time.

Reusing flows. The idea is now not to discard the flow of a subproblem, but to "recycle" it for the next subproblem. Many subproblems are very similar; consider for example two valid partitions that only differ in the assignment of one vertex pair u, u'. The corresponding flow networks differ in only two arcs: in one (call it F_1) u is connected to s, and in the other (F_2) u is connected to t; the equivalent holds with respect to u'. As we show now, if we already have a maximum flow for F_1, we can calculate a maximum flow for F_2 with a constant number of augmentation operations.

In a first step, we need to drain the flow along the arc (s, u). Note that there is always flow along this arc, since otherwise the flow value of F_1 is below $k := |X|$ and we would have successfully returned in line 5 of Figure 4.14. For this, we find a "diminuting path" from u to t, that is, a path that uses only arcs (v, w) that have positive flow $f(v, w)$. Because of flow conservation, such a path must always exist. We then clear the flow along the path and the edge (s, u). By the same arguments as those for an augmenting path, this yields a valid flow. We can then safely delete (s, u) and insert (u, t). If the diminuting path happened to pass through u', we already got rid of the flow along the edge (u', t); otherwise, we drain it with an analog diminution operation. Finally, by finding one or two augmenting paths, we can obtain a maximum flow for F_2.

As an example, consider Figure 4.16. In Figure 4.16a, we have a maximum flow for a particular valid partition. The flow can be easily seen to be maximum, since there is no augmenting path. We now wish to calculate a maximum flow

for a modified valid partition, where v_1 and v_2 switch their assignment. For this, we first find a diminuting path from v_2 to v_1 (Figure 4.16b, bold lines) and decrease the flow along the path. Next, we try to increase the flow again by finding an augmenting path between the black and the white vertices. In this case, no such augmenting path can be found; therefore, the flow value is below $|X|$ and we can retrieve the smaller bipartization set as minimum cut (crossed edges).

Both finding a diminuting and an augmenting path can be done in $O(m)$ time by breadth-first search, meaning the update of the allocation of the endpoints of a single edge in X to ○ or ● can be done in $O(m)$ time.

We would now like to order the subproblems such that there is a minimal amount of change between successive subproblems. In fact, we can order the subproblems such there is only a single change between successive subproblems by using a Gray code (see Knuth [2004, Section 7.2.1.1] for the history of Gray codes). A Gray code is a binary numeral system where two successive values differ in only one digit. For example, the numbers 0 to 7 in a 3-bit Gray code are 000, 001, 011, 010, 110, 111, 101, and 100. We use a k-bit Gray code and associate each edge in X with one bit position. For an edge $\{u, v\} \in A$, we associate "u start vertex, v target vertex" with 0 and "v start vertex, u target vertex" with 1. If we now enumerate the 2^k subproblems in the order of the Gray code, then there will always be only one allocation of an edge changing, meaning that we can solve each subproblem (but the first) in $O(m)$ time, yielding the following theorem.

Theorem 4.11. Edge Bipartization *can be solved in* $O(2^k \cdot m^2)$ *time.*

4.5.3 Heuristic speedup

In this section, we show how sometimes we can reduce the number of valid partitions we have to enumerate. Recall that to ensure solution disjointness, we subdivided each edge in X into 3 segments (Graph Transformation 4.1). The idea now is to choose one of the two end segments into the solution instead of the middle segment. The proof of Lemma 4.7 still holds with this change. However, it can happen then that two edges in the solution set X have the same endpoint. This can be exploited. Consider the example in Figure 4.17 (left). We have $|X| = 3$, and therefore we would normally need to enumerate $2^{|X|} = 8$ valid partitions. If we choose an end segment instead of the middle segment, we can obtain a graph as in Figure 4.17 (right). Here, up to symmetry only a single valid partition is possible (black and white vertices), which immediately gives us the smaller solution (crossed edges) as minimum cut between the black and the white vertices.

The graph induced by the edges in X after the thus modified transformation

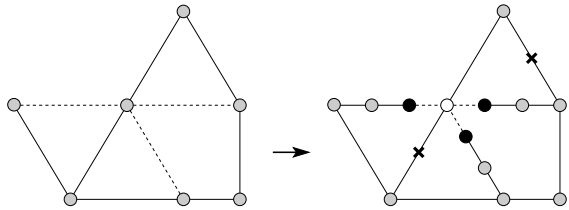


Figure 4.17: Example for the reduction in the number of valid partitions to enumerate

is always a disjoint union of stars (a star is a graph where every vertex but one has degree 1). This is because only one endpoint of an edge in X can be incident on another edge of X, since one endpoint is always a degree-2 vertex incident only on a newly inserted edge. We quantify the gain in the following lemma.

Lemma 4.9. *When the edges in* X *induce* $s \leqslant k$ *stars, we need to examine only* 2^s *valid partitions.*

Proof. There are only two assignments for the center vertex of a star, which determines the assignments of all the other vertices in the star. □

To obtain the maximum gain from this, we have to choose carefully which of the two end segments of a subdivided edge we want to take into the new X. The goal is to minimize the number of stars that result. We can think of this as marking one of the endpoints of each edge in X such that the total number of marked vertices is minimized. This is exactly the well-known Vertex Cover problem in the subgraph induced by X. While Vertex Cover is NP-hard, in our experiments (detailed in Section 3.2.4) the resulting instances are small and sparse (e. g., 80 vertices and 80 edges), and can be easily solved by a simple branching strategy: choose an edge, and include either one endpoint or the other endpoint in the cover. Even if the resulting instances were too large to solve them exactly, we could solve them heuristically and still gain, although possibly not as much as when solving them optimally.

Clearly, Lemma 4.9 is only a heuristic improvement; when the edges of X before the transformation have no endpoints in common, we cannot gain anything. However, since already e. g. 5 pairs of edges each having one endpoint in common yield a speedup of a factor of $2^5 = 32$, the gain can be quite large in practice. In a Balanced Subgraph instance (see Section 3.2.4) where without the heuristic one would have checked 2^{74} valid partitions of the endpoints of the edges in X, due to the trick only 2^{34} valid partitions had to be considered. Thus, only due to the trick iterative compression became feasible, saving a factor

of $2^{40} \approx 10^{12}$ in the running time. As expected, the trick works particularly well in dense graphs.

4.5.4 Generalization to Balanced Subgraph

The Balanced Subgraph problem, introduced in Section 2.3.2, is a generalization of Edge Bipartization. We quickly recall the definition. Balanced Subgraph is defined on *signed graphs*, that is, graphs where every edge is annotated with $=$ or $\neq$. A signed graph is *balanced* if its vertices can be colored with two colors such that the relation at each edge holds with respect to the colors of its endpoints. The Balanced Subgraph problem is then defined as follows:

> Balanced Subgraph
> **Instance:** A signed graph $G = (V, E)$ and an integer $k \geqslant 0$.
> **Question:** Can G be transformed by up to k edge deletions into a balanced graph?

In several applications, one may assume that only a small fraction of the graph edges has to be omitted; for example, it has been observed that many biochemical networks are close to being balanced [Sontag 2007b]. Therefore, it is attractive to use the number of edges that need to be deleted to obtain a balanced graph as a parameter.

We have shown that there is a simple reduction from Balanced Subgraph to Edge Bipartization (Proposition 2.1). Combining this with the iterative compression algorithm for Edge Bipartization (Theorem 4.11) gives us the following theorem, which establishes the fixed-parameter tractability of Balanced Subgraph.

Theorem 4.12. Balanced Subgraph *can be solved in* $O(2^k \cdot m^2)$ *time.*

Theorem 4.12 improves an $O(n^{2L} \cdot (nm)^3)$ time exact algorithm by DasGupta et al. [2007, Remark 1], where L is the number of $\neq$-edges (since clearly $k \leqslant L$).

It turns out that we do not actually need the reduction, which might double the size of the instance, but rather can do with a small modification to the algorithm itself. The reason is that Lemma 4.8 still holds if we replace "edge bipartization set" by "balancing set", where a balancing set is a set of edges whose deletion makes a graph balanced. Intuitively, the reason is that for the edge cut between $\bigcirc_\Phi$ and $\bullet_\Phi$ to exist, it does not matter whether we applied the reduction that subdivides edges.

Lemma 4.10. *Consider a graph* $G = (V, E)$ *and a minimal balancing set* X *for* G*. For a set of edges* $Y \subseteq E$ *with* $X \cap Y = \emptyset$*, the following are equivalent:*

(1) Y *is a balancing set for* G*.*

(2) *There is a valid partition* Φ *of* $V(X)$ *such that* Y *is an edge cut in* $G \setminus X$ *between* $\bigcirc_\Phi := \Phi^{-1}(\bigcirc)$ *and* $\bullet_\Phi := \Phi^{-1}(\bullet)$.

Proof. Consider the following statements for the graph G' obtained by Graph Transformation 2.1 from G and the sets X' and Y' obtained from X and Y, respectively, by replacing $e = \{v, w\} \in E_=$ with $\{v, e\}$.

(3) Y' is a balancing set for G'.

(4) There is a valid partition Φ of $V(X')$ such that Y' is an edge cut in $G' \setminus X$ between $\bigcirc_\Phi := \Phi^{-1}(\bigcirc)$ and $\bullet_\Phi := \Phi^{-1}(\bullet)$.

Clearly, (1) $\iff$ (3). By Lemma 4.8, (3) $\iff$ (4). Finally, it is not hard to see that (4) $\iff$ (2). □

Because of Lemma 4.10, surprisingly, we can use CompressEB completely unchanged for Balanced Subgraph; in particular, it can ignore edge signs. The single change to the overall algorithm required is in line 5 of Figure 4.11: To ensure inclusion minimality of the set X to be compressed, we need to check whether X is a balancing set instead of a bipartization set.

This means that in practice, there is basically no overhead involved in solving Balanced Subgraph instead of Edge Bipartization. Further, the speedups by exploiting subproblem similarity (Section 4.5.2) and by exploiting neighboring vertices in the solution set (Section 4.5.3) can still be applied unchanged.

The iterative compression for Balanced Subgraph was implemented and evaluated together with data reduction rules for Balanced Subgraph as shown in Section 3.2. This combination was used successfully to solve instances with up to 678 vertices and 1582 edges. We refer to Section 3.2.4 for details.

4.5.5 Outlook

There are several ways in which the results of this section can be expanded.

Algorithm modifications. At the core of our iterative compression algorithm, a large number of cut problems in almost identical networks needs to be solved. It is tempting to simplify the graph before doing this, thereby achieving at least a heuristic speedup. Misiołek and Chen [2006] presented several data reduction rules for minimum cut problems; however, these are rather weak, because they could not afford a large polynomial running time. Since we are solving exponentially many related instances, we could invest much more time into data reduction.

In Section 4.3, we showed that the iterative compression algorithm for Cluster Vertex Deletion can be used to also solve weighted instances. It would be

nice if this also holds for Edge Bipartization. Further, iterative compression algorithms for some other problems have been extended to enumerate all solutions, instead of finding just one [Guo et al. 2006, Chen et al. 2007a]. This is also an interesting goal for Edge Bipartization.

For the Vertex Bipartization problem in planar graphs, Fiorini et al. [2005] gave a fixed-parameter algorithm that runs in linear time for fixed k, at the cost of a much worse combinatorial explosion with respect to k compared to the iterative compression approach presented in Section 4.6. It is open to obtain a similar result for Edge Bipartization.

Related problems. Sontag [2007a] suggested to examine the following directed variant of Balanced Subgraph. The motivation is the same as for the examination of Balanced Subgraph, namely, to model stability in dynamic networks of biological origin [DasGupta et al. 2007, Sontag 2007b].

Directed Balanced Subgraph
Instance: A directed signed graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k edge deletions into a graph without negative cycles?

Here, a negative cycle is a directed cycle with an odd number of negative signs. Using an analog of Proposition 2.1, by inserting an extra vertex into $=$-edges, we can get rid of the signs and obtain the equivalent Directed Edge Bipartization problem.

Directed Edge Bipartization
Instance: A directed signed graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k edge deletions into a graph without odd-length cycles?

Although in addition to its application, this problem seems quite natural, I was not able to find any reference in the literature. By a reduction from the undirected case, it is easy to see that Directed Edge Bipartization is NP-hard. With a general result by Raman and Sikdar [2007], the parametric dual of this question (that is, the question whether a directed graph contains an odd-cycle-free subgraph of size at least k) is W[1]-hard. The parameterized complexity of Directed Edge Bipartization is open, though. Iterative compression seems like a good candidate in tackling this problem, since it was helpful for showing fixed-parameter tractability of the related problems Edge Bipartization (Section 4.5) and Directed Feedback Vertex Set [Chen et al. 2008] (which can be parameter-preserving reduced to and from Directed Feedback Edge Set).

4.6 Iterative compression for Vertex Bipartization

In this section, we show how to solve the Vertex Bipartization problem, introduced in Section 2.3.3, by iterative compression. Vertex Bipartization was the first problem for which iterative compression was applied [Reed et al. 2004]. Based on new structural insights, we give a simplified and more intuitive exposition and proof of this result, and slightly improve the worst-case time complexity. This also allows to establish a heuristic improvement that in particular speeds up the search on dense graphs. Our implementation can solve all problems from a testbed from computational biology within minutes, whereas established methods are only able to solve about half of the problems within reasonable time (Section 4.6.3). We recall the definition of Vertex Bipartization and refer to Section 2.3.3 for previous results and applications.

Vertex Bipartization
Instance: An undirected graph $G = (V, E)$ and an integer $k \geqslant 0$.
Question: Can G be transformed by up to k vertex deletions into a bipartite graph?

We call a set of vertices whose deletion makes a graph bipartite an *odd cycle cover* (the name is justified by Lemma 1.1).

4.6.1 Iterative compression algorithm

In this section we give a novel, more intuitive presentation of the algorithm for Vertex Bipartization by Reed et al. [2004]. As opposed to their proof, our presentation does not employ case distinction or contradiction, and gives a better intuition of why the algorithm actually works. This also allows to establish several improvements in Section 4.6.2.2 and Section 4.6.2.3. Our presentation follows the same lines as that for Edge Bipartization (Section 4.5).

We use the same approach as for our introductory example 3-Hitting Set (Section 4.2): The graph is built up vertex-by-vertex, and an optimal solution is held current by compression; the pseudo-code in Figure 4.1 can be used unchanged.

For the compression routine itself, we again use the reduction of the compression task to a *disjoint* compression task: we enumerate all possibilities how a smaller solution X' might reuse vertices of the known solution X (Figure 4.3). This gives us the following property, at a price of a factor of $O(2^k)$ in the running time.

Property 4.1. *X' is disjoint from X, that is, $X \cap X' = \emptyset$.*

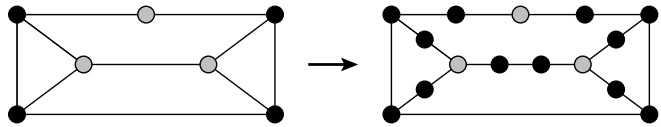


Figure 4.18: Input transformation for Compress-OCC. The *grey vertices* are the elements of the vertex bipartization set X.

This is a notable difference to Edge Bipartization, where we could achieve the equivalent of Property 4.1 by a simple input transformation (Graph Transformation 4.1).

We then impose some additional useful properties on the instance.

Property 4.2. *No two vertices from* X *are neighbors, that is,* $\forall u, v \in X : u \notin N(v)$.

Property 4.3. *No vertex in a smaller odd cycle cover* X' *is neighbor of a vertex in* X*, that is,* $N(X') \cap X = \emptyset$.

Properties 4.2 and 4.3 can be obtained by a simple input transformation: subdivide each edge adjacent to a vertex in X by a new vertex (see Figure 4.18). This is done successively, that is, edges connecting two vertices from X are subdivided by two vertices. This transformation preserves the parity of the length of any cycle C, since for each vertex in C that is in X, two new vertices are inserted into the edges of C. Therefore, after this transformation, X is still an odd cycle cover, and any odd cycle cover for the transformed graph can easily be converted to an odd cycle cover of the same size for the original graph. The transformation allows us to assume without loss of generality that no vertex $v \in X'$ is neighbor of a vertex in X (Property 4.3). This is because any vertex v in an odd cycle cover X' that is neighbor of a vertex in X must be one of the newly inserted degree-2 vertices, and can be replaced by the neighbor of v that is not in X, leading to a solution of the same size.

The task is thus boiled down to the following.

VB Disjoint Compression
Instance: An undirected graph $G = (V, E)$ and a vertex bipartization set X with Property 4.2 for G.
Task: Find a vertex bipartization set X' with $|X'| < |X|$ and Properties 4.1 and 4.3 or prove that this is not possible.

The key to solving VB Disjoint Compression is to compare the two two-colorings of G induced by X and the (yet unknown) X' (see Figure 4.19): some vertices will have the same color in both colorings, and others will get different

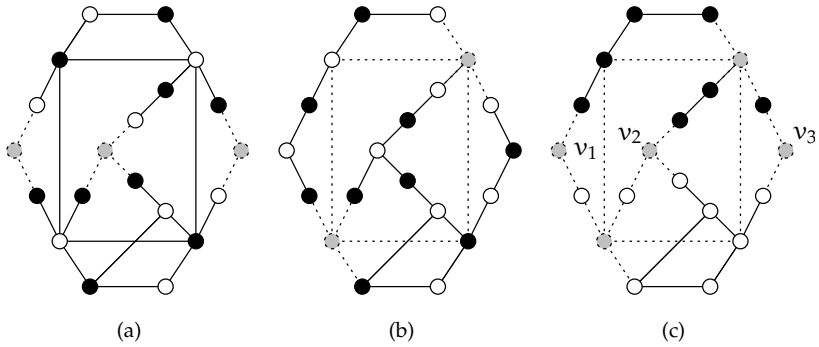


Figure 4.19: Comparing disjoint odd cycle covers: (a) a graph G with an odd cycle cover X (*grey vertices*); a two-coloring C_X of $G \setminus X$ is marked with *black and white vertices*; (b) another odd cycle cover X' of G with Properties 4.1–4.3, and a corresponding two-coloring $C_{X'}$; (c) the comparison function Φ.

colors. More precisely, let C_X and $C_{X'}$ be fixed two-colorings of $G \setminus X$ and $G \setminus X'$, respectively, and define the *comparison function*

$$\Phi : V \setminus (X \cup X') \to \{\bigcirc, \bullet\} : v \mapsto \begin{cases} \bigcirc & \text{if } C_X(v) = C_{X'}(v); \\ \bullet & \text{if } C_X(v) \neq C_{X'}(v). \end{cases} \tag{4.7}$$

The decisive property of Φ is given in the following lemma and illustrated in Figure 4.19c.

Lemma 4.11. *In the setting of* VB Disjoint Compression, *the set* $X \cup X'$ *is a vertex cut between the vertex sets* $\bigcirc_\Phi := \Phi^{-1}(\bigcirc)$ *and* $\bullet_\Phi := \Phi^{-1}(\bullet)$.

Proof. Consider an edge $\{v, w\} \in E$ with $v, w \in V \setminus (X \cup X')$. Since C_X and $C_{X'}$ are two-colorings, we have $C_X(v) \neq C_X(w)$ and $C_{X'}(v) \neq C_{X'}(w)$. Thus, $\Phi(v) = \Phi(w)$, that is, Φ is constant along any edge that has no endpoint in $X \cup X'$. Consequently, there can be no path between two vertices with different values of Φ that does not contain a vertex from X or X'. □

Lemma 4.11 naturally suggests obtaining X' from a vertex cut, which is a polynomial-time task. However, we do not know the value of Φ yet, since it depends on X'. But as we will see, it suffices to guess a small part of Φ by brute force.

For this, consider the value of Φ for the neighbors of some vertex $v \in X$. Because of Properties 4.2 and 4.3, no neighbor of v is in X or in X', so Φ is defined for all neighbors of v. Further, since neither v (by Property 4.1)

nor its neighbors are in X', the value of $C_{X'}$ is equal for all of v's neighbors. Therefore, there are only two possibilities: for all $w \in N(v) : \Phi(w) = C_{X'}(w)$, or for all $w \in N(v) : \Phi(w) \neq C_{X'}(w)$. Figure 4.19c shows an example: for all neighbors w of v_1 and v_2, we have $\Phi(w) = C_X(w)$, and for all neighbors w of v_3, we have $\Phi(w) \neq C_X(w)$.

This motivates the following definition.

Definition 4.4. *Consider a graph* G *and an odd cycle cover* X *for* G*, with* C_X *being a fixed two-coloring of* $G \setminus X$*. Then a coloring* $\Psi : N(X) \to \{\bigcirc, \bullet\}$ *is called* valid *when for all* $v \in X$ *either* $\forall w \in N(v) : \Psi(w) = C_X(w)$ *or* $\forall w \in N(v) : \Psi(w) \neq C_X(w)$.

Thus, there are $2^{|X|}$ valid colorings. We can now state the central lemma of this section, which is analogous to Lemma 4.8 for Edge Bipartization.

Lemma 4.12. *In the setting of* VB Disjoint Compression*, for a vertex set* $X' \subseteq V$*, the following are equivalent:*

(1) X' *is an odd cycle cover for* G*.*

(2) There is a valid coloring Ψ *of* $N(X)$ *such that* X' *is a vertex cut between* $\bigcirc_\Psi := \Psi^{-1}(\bigcirc)$ *and* $\bullet_\Psi := \Psi^{-1}(\bullet)$ *in* $G \setminus X$*.*

Proof. (2) $\Rightarrow$ (1): Consider a vertex set C that induces an odd cycle in G. It suffices to show that $C \cap X' \neq \emptyset$. Since X is an odd cycle cover, there is at least one vertex from X in C. For at least one vertex $v \in C \cap X$, its two cycle neighbors v_l and v_r on C have different colors in C_X, that is, $C_X(v_l) \neq C_X(v_r)$; otherwise, we could two-color the odd cycle C, since no two vertices from X are neighbors by Property 4.2. By the definition of a valid coloring, this implies $\Psi(v_l) \neq \Psi(v_r)$. Since X' is a vertex cut in $G \setminus X$ between the differently colored vertices v_l and v_r, there must be some $v' \in X'$ with $v' \in C$.

(1) $\Rightarrow$ (2): As argued above, $\Psi := \Phi|_{N(X)}$ (that is, Φ restricted to the neighbors of vertices from X) is a valid coloring, and by Lemma 4.11 X' is a vertex cut between $\bigcirc_\Psi$ and $\bullet_\Psi$ in $G \setminus X$. □

With Lemma 4.12, it is now clear that we can solve VB Disjoint Compression by trying all $2^{|X|}$ valid colorings Ψ and determining a minimum vertex cut between $\bigcirc_\Psi$ and $\bullet_\Psi$. We now have everything in place to present the compression routine.

Compress-OCC(G_0, X_0)
1 subdivide edges around each $v \in X_0$
2 **for each** $X \subseteq X_0$**:**
3 $G \leftarrow G_0 \setminus (X_0 \setminus X)$

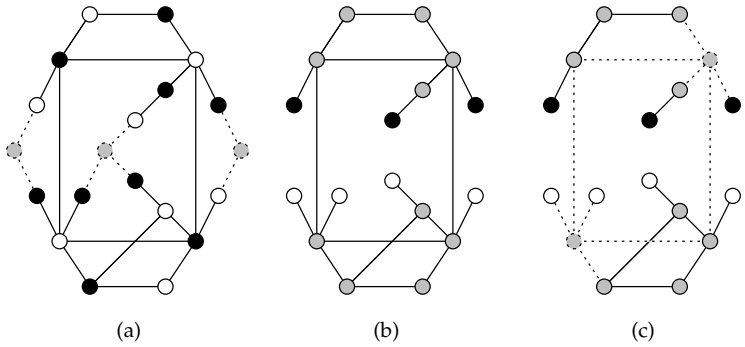


Figure 4.20: Illustration of the algorithm for solving VB Disjoint Compression (Compress-OCC): (a) Graph G with odd cycle cover X (*grey vertices*); (b) $G \setminus X$ with a valid coloring Ψ (*black and white vertices*); (c) a vertex cut X' (*dashed vertices*) between the black and the white vertices is an odd cycle cover for G.

4 **for each** valid coloring Ψ of $N(X)$**:**
5 **if** there is a vertex cut D in $G \setminus X$ between $\circ_\Psi$ and $\bullet_\Psi$ with $|D| < |X|$**:**
6 **return** $(X_0 \setminus X) \cup D$
7 **return** X_0

We now explain Compress-OCC in detail. Given is a graph G_0 with an odd cycle cover X_0. First we ensure Properties 4.2 and 4.3 by a simple input transformation (line 1; see Figure 4.18). We then examine every subset X of the known odd cycle cover X_0 (line 2). For each X, we look for smaller odd cycle covers for G that can be constructed by replacing the vertices of X in X_0 by fewer new vertices from $V \setminus X$ (clearly, for any smaller odd cycle cover, such an X must exist). Since we thereby decided to retain the vertices in $X_0 \setminus X$ in our odd cycle cover, we examine the graph $G = G_0 \setminus (X_0 \setminus X)$ (an example is shown in Figure 4.20a). After line 3, we have Properties 4.2 and 4.1 for G and X. If we now find an odd cycle cover D for G with $|D| < |X|$, we are done, since then $(X_0 \setminus X) \cup D$ is an odd cycle cover smaller than X_0 for G_0. For this, we try all valid colorings for $N(X)$ (Figure 4.20b). By Lemma 4.12, there is some valid coloring where a smaller odd cycle cover forms a vertex cut between $\circ_\Phi$ and $\bullet_\Phi$ in $G \setminus X$; and moreover, any such cut is an odd cycle cover. Therefore, if we find a vertex cut D between $\circ_\Phi$ and $\bullet_\Phi$ that is smaller than X, we are done (Figure 4.20c); conversely, if no valid coloring is successful, it is not possible to compress X.

Running Time. Reed et al. [2004] state the running time of their algorithm as $O(4^k \cdot kmn)$; a slightly more careful analysis reveals it as $O(3^k \cdot kmn)$. For this, note that in effect the two loops in lines 2 and 4 of Compress-OCC iterate over all possible assignments of each $v \in X_0$ to three roles:

- either $v \in X_0 \setminus X$,
- or $\Psi(w) = C_X(w)$ for all neighbors w of v,
- or $\Psi(w) \neq C_X(w)$ for all neighbors w of v.

Therefore, we solve 3^k minimum vertex cut problems, and since we can solve one minimum vertex cut problem in $O(km)$ time by the Edmonds–Karp algorithm [Dinic 1970, Edmonds and Karp 1972], the running time for one invocation of Compress-OCC is $O(3^k \cdot km)$. As the iterative compression main loop (Figure 4.1) calls Compress-OCC n times, we arrive at an overall running time of $O(3^k \cdot kmn)$.

Theorem 4.13. Vertex Bipartization *can be solved in* $O(3^k \cdot kmn)$ *time.*

4.6.2 Algorithmic improvements

We now present several improvements over the algorithm by Reed et al. [2004]. We start with two simple improvements that save a constant factor in the running time. In Section 4.6.2.2 we then show how to save a factor of k in the running time by exploiting the similarity of the subproblems solved. Finally, in Section 4.6.2.3 we present an improvement exploiting the structure of the subgraph induced by the bipartization set. This improvement gave the most pronounced speedups in our experiments presented in Section 4.6.3.

4.6.2.1 Simple improvements

It is easy to see that for each valid coloring Ψ there is a symmetric coloring where the value is inverted at each vertex, leading to the same vertex cuts. Therefore we can arbitrarily fix the allocation of the neighbors of one vertex, saving a factor of 2 in the running time.

The next improvement is justified by the following lemma.

Lemma 4.13. *Consider a graph* $G = (V, E)$*, a vertex* $v \in V$*, and a minimum-size odd cycle cover* X *for* $G \setminus \{v\}$ *with* $|X| = k$*. Then no odd cycle cover of size* k *for* G *contains* v*.*

Proof. If X' is an odd cycle cover of size k for G, then $X' \setminus \{v\}$ is an odd cycle cover of size $k-1$ for $G[V \setminus \{v\}]$, contradicting that $|X|$ is of minimum size. □

With Lemma 4.13 it is clear that the vertex v we add to X in line 5 of the iterative compression main loop (Figure 4.1) cannot be part of a smaller odd cycle cover, and we can omit the case $v \notin X$ in Compress-OCC, saving a third of the cases.

4.6.2.2 Exploiting subproblem similarity

In the "inner loop" of Compress-OCC (line 5), we need to find a minimum size vertex cut between two vertex sets in a graph. This is a classic application for maximum flow techniques: The well-known max-flow min-cut theorem [Cormen et al. 2001] tells us that the size of a minimum edge cut is equal to the maximum flow. Since we are interested in vertex cuts, we create a new, directed graph G' for our input graph $G = (V, E)$: for each vertex $v \in V$, create two vertices v_{in} and v_{out} and a directed edge (v_{in}, v_{out}). For each edge $\{v, w\} \in E$, we add two directed edges (v_{out}, w_{in}) and (w_{out}, v_{in}). It is not hard to see that a maximum flow in G' between $Y_1' := \bigcup_{y \in Y_1} y_{in}$ and $Y_2' := \bigcup_{y \in Y_2} y_{out}$ corresponds to a maximum set of vertex disjoint paths between Y_1 and Y_2. Furthermore, an edge cut D between Y_1' and Y_2' is of the form $\bigcup_{v \in V}\{(v_{in}, v_{out})\}$, and $\bigcup_{(v_{in}, v_{out}) \in D}\{v\}$ is a vertex cut between Y_1 and Y_2 in G.

Since we know that the cut is relatively small (less than or equal to k), we employ the Edmonds–Karp algorithm [Dinic 1970, Edmonds and Karp 1972]. This algorithm repeatedly finds a shortest augmenting path in the flow network and increases the flow along it, until no further increase is possible. We assume in the rest of this section that the reader is familiar with this algorithm.

The idea is then, similar to the trick for Edge Bipartization in Section 4.5.2, that the flow problems solved in Compress-OCC are "similar" in such a way that we can "recycle" the flow networks for each problem. For this, after line 3 of Compress-OCC, we merge all white neighbors of each $v \in X$ into a single vertex v_1, and all black neighbors of each $v \in X$ into a single vertex v_2. Clearly, this does not change the minimum cut found in line 5. Then, each flow problem corresponds to one assignment of the vertices in X to the three roles "v_1 source, v_2 target", "v_2 source, v_1 target", and "not present" ($v \notin X$). Using a so-called $(3, k)$-ary Gray code [Guan 1998], we can enumerate these assignments in such a way that adjacent assignments differ in only one element. For each of these (but the first one), one can solve the flow problem by adapting the previous flow. One or both of the following actions is needed:

- If the vertex v whose assignment was changed was present previously, drain the flow along the path with end point v_1 and the path with end point v_2 (note that they might be identical). Here, "drain the flow" means to find an augmenting path in the flow network (as opposed to the residual network), and zero the flow along this path.

- If v is present in the updated assignment, find an augmenting path from v_1 to v_2 or from v_2 to v_1, depending on the current role of v.

Since each of these operations can be done in $O(m)$ time, we can perform the update in $O(m)$ time, as opposed to $O(km)$ time for solving a flow problem from scratch. This improves the overall worst case running time to $O(3^k \cdot mn)$. We call this algorithm OCC-Gray.

Theorem 4.14. Vertex Bipartization *can be solved in* $O(3^k \cdot mn)$ *time.*

4.6.2.3 Filtering of valid colorings

Lemma 4.12 tells us that for Disjoint Compression, there is a valid coloring for $N(X)$ such that we will find a cut leading to a smaller odd cycle cover. Therefore, simply trying all valid colorings will be successful. However, a more careful examination allows to omit some valid colorings from consideration. For this, consider two vertices $c, d \in X$ that are connected by an edge. After the input transformation, they are connected by a path containing two fresh vertices v_c and v_d. If we now have a valid coloring that assigns different values to v_c and v_d, it is not possible to find a vertex cut that disconnects them, since they are directly connected by an edge. Therefore, any valid coloring that is to be successful must assign them the same colors.

For notational convenience, we now identify a valid coloring Ψ with a coloring C_Ψ of the vertices in X. We identify the choice $\forall w \in N(v) : \Psi(w) = C_X(w)$ with painting v white and the choice $\forall w \in N(v) : \Psi(w) \neq C_X(w)$ with painting v black. Then, the above observation imposes $C_\Psi(v) \neq C_\Psi(w)$ for all $\{v, w\} \in E$. This means that C_Ψ is a two-coloring of $G[X]$. If $G[X]$ is not bipartite, we can immediately give up trying to find a smaller odd cycle cover; otherwise, we only have to try all two-colorings of $G[X]$ (there can be more than one if $G[X]$ is disconnected). This leads to the following algorithm.

```
Compress-OCC-Enum2Col(G_0, X_0)
1  subdivide edges around each v ∈ X
2  for each bipartite subgraph B of G[X]:
3      for each two-coloring C_Ψ of B:
4          if there is a vertex cut D in G \ X between ○_Ψ and ●_Ψ with |D| < |X|:
5              return (X_0 \ X) ∪ D
6  return X_0
```

The worst case for Compress-OCC-Enum2Col is that X is an independent set in G. In this case, every subgraph of $G[X]$ is bipartite and has $2^{|X|}$ two-colorings.

This leads to exactly the same number of flow problems solved as for Compress-OCC. In the best case, X is a clique, and $G[X]$ has only $O(|X|^2)$ bipartite subgraphs, each of which admits (up to symmetry) only one two-coloring.

It is easy to construct a graph where any optimal odd cycle cover is independent; therefore the described modification does not lead to an improvement of the worst-case running time. However, at least in a dense graph, it is "unlikely" that the odd cycle covers are completely independent, and already a few edges between vertices of the odd cycle cover can vastly reduce the required computation.

With a simple branching strategy, one can enumerate all bipartite subgraphs of a graph and all their two-colorings with constant cost per two-coloring. This can also be done in such a way that modifications to the flow graph can be done incrementally, as described in Section 4.6.2.2. The two simple improvements mentioned at the beginning of this section also can still be applied. We call the thus modified algorithm OCC-Enum2Col.

It seems plausible that for dense graphs, an odd cycle cover is "more likely" to be connected, and therefore this heuristic is more profitable. Experiments on random graphs confirm this (see Section 4.6.3.3). This is of particular interest because other strategies (such as reduction rules [Wernicke 2003]) seem to have a harder time with dense graphs than with sparse graphs, making hybrid algorithms appealing.

4.6.3 Implementation and experiments

Before presenting experimental results on our iterative compression algorithm for Vertex Bipartization, we describe two conventional approaches, which we used as a comparison point.

Branch & bound. Wernicke [2003] presented an algorithm for Vertex Bipartization based on branch & bound. A branching decision simply tries for some vertex v the two cases that v is in the odd cycle cover or not. The improvement over the trivial 2^n algorithm comes from the use of good lower and upper bounds and from data reduction rules. Wernicke [2003] presented some experimental results on real-world data.

Integer linear program. Integer linear programs (ILPs) are frequently used in practice to solve hard problems. The reason is that it is often easy to model the problems as ILP, and that powerful solvers are available, which profit from years of research and engineering experience. We refer to the literature [Schrijver 1998, Cormen et al. 2001] for details.

Vertex Bipartization can be formulated as an ILP as follows:

$$
\begin{aligned}
& c_1, \ldots, c_n : \text{binary variables} \quad \textit{(cover)} \\
& s_1, \ldots, s_n : \text{binary variables} \quad \textit{(side)} \\
& \text{minimize} \sum_{i=1}^{n} c_i \\
& \text{s. t. } \forall \{v, w\} \in E : (s_v \neq s_w) \vee (c_v = 1) \vee (c_w = 1)
\end{aligned}
$$

where the constraint can be expressed in canonical ILP form as

$$
\begin{aligned}
\text{s. t. } & \forall \{v, w\} \in E : s_v + s_w + (c_v + c_w) \geqslant 1 \\
& \forall \{v, w\} \in E : s_v + s_w - (c_v + c_w) \leqslant 1.
\end{aligned}
$$

Here, a 1 in c_v models that v is part of the odd cycle cover. The variables s_v model the side of the bipartite graph that remains when deleting the vertices from the odd cycle cover. The first set of constraints enforces that for an edge either one endpoint has color 1, or the other has color 1, or one of them is in the cover. In effect, it forbids that both endpoints have color 0 while none of them is in the cover. Analogously, the second set of constraints forbids that both endpoints have color 1 while none is in the cover.

We first evaluated the conventional approaches. The ILP performed quite well; when solved by GNU GLPK [Makhorin 2004], it consistently outperformed the highly problem-specific branch-and-bound approach by Wernicke [2003] on our test data, sometimes by several orders of magnitude. Therefore, we use the ILP as the comparison point for the performance of our algorithms and do not give details on the branch-and-bound approach.[1]

Implementation Details. The program is written in the C programming language and consists of about 1400 lines of code. The source and the test data are available from http://theinf1.informatik.uni-jena.de/occ/.

Data structures. Over 90% of the time is spent in finding an augmenting path within the flow network; all that this requires from a graph data structure is enumerating the neighbors of a given vertex. The only other frequent operation

[1] We recently found that more sophisticated mathematical programming approaches have been suggested [Fouilhoux and Ridha Mahjoub 2006]. It would be interesting to extend our comparison to these.

is "enabling" or "disabling" vertices as determined by the Gray code (see Section 4.6.2.2). In particular, it is not necessary to quickly add or remove edges, or query whether two vertices are neighbors. Therefore, we chose a very simple data structure, where the graph is represented by an array of neighbor lists, with a null pointer denoting a disabled vertex.

Since the flow simply models a set of vertex-disjoint paths, it is not necessary to store a complete $n \times n$-matrix of flows; it suffices to store the flow predecessor and successor for each node, reducing memory usage to $O(n)$.

Experimental Setup. We tested our implementation on various inputs. The testing machine is an AMD Athlon 64 3700+ with 2.2 GHz, 1 MB cache, and 1 GB main memory, running under the Debian GNU/Linux 3.1 operating system. The source was compiled with the GNU gcc 3.3.4 compiler with option "-O3". Memory requirements are around 3 MB for the iterative compression based algorithms and up to 500 MB for the ILP.

4.6.3.1 Minimum Site Removal

The first test set originates from computational biology (see Section 2.3.3). The instances were constructed by Wernicke [2003] from data of the human genome as a means to solve the so-called Minimum Site Removal problem. We examine these instances to learn about the performance of our algorithms on real-world instances, in particular those modeling a data correction task (the graph should be bipartite, but is distorted, and a most parsimonious reconstruction is sought). The results are shown in Table 4.1.

Runs were cancelled after 2 hours without result. We show only the instance of median size for each value of $|X|$. The column "ILP" gives the running time of the ILP given at the beginning of Section 4.6.3 when solved by GNU GLPK [Makhorin 2004]. The column "Reed" gives the running time of Reed et al.'s algorithm without any of the algorithmic improvements from Section 4.6.2 except for the obvious improvement of omitting symmetric valid partitions. The columns "OCC-Gray" and "OCC-Enum2Col" give the running time for the respective algorithms from Sections 4.6.2.2 and 4.6.2.3.

As expected, the running times of the iterative compression algorithms mainly depend on the size of the odd cycle cover that is to be found. Interestingly, the ILP also shows this behavior; the probable explanation is that it takes the branch-and-bound part of the solver longer to establish inferiority of an assignment of the integer variables to an upper bound found previously. The observed improvement in the running time from "Reed" to "OCC-Gray" is slightly lower than the factor of k gained in the worst-case complexity, but clearly still worthwhile. The heuristic from Section 4.6.2.3 works exceedingly

Table 4.1: Running times in seconds for different algorithms for benchmark instances by Wernicke [2003]

	n	m	density	$\|X\|$	ILP	Reed	OCC-Gray	OCC-Enum2Col
Afr. #31	30	51	11.7	2	0.02	0.00	0.00	0.00
Jap. #19	84	172	4.9	3	0.23	0.00	0.00	0.00
Jap. #24	142	387	3.9	4	1.30	0.00	0.00	0.00
Jap. #11	51	212	16.6	5	0.50	0.00	0.00	0.00
Afr. #10	69	191	8.1	6	3.49	0.00	0.00	0.00
Afr. #36	111	316	5.2	7	16.93	0.01	0.00	0.00
Jap. #18	71	296	11.9	9	73.94	0.09	0.02	0.00
Jap. #17	79	322	10.5	10	100.93	0.27	0.04	0.00
Afr. #11	102	307	6.0	11	3790.99	1.10	0.14	0.02
Afr. #54	89	233	5.9	12		5.13	0.61	0.10
Afr. #34	133	451	5.1	13		9.36	0.98	0.02
Afr. #52	65	231	11.1	14		20.95	2.08	0.02
Afr. #22	167	641	4.6	16		318.95	31.24	0.15
Afr. #48	89	343	8.8	17		1269.01	104.20	0.11
Afr. #50	113	468	7.4	18		5287.68	501.15	0.06
Afr. #19	191	645	3.6	19			1288.85	2.23
Afr. #45	80	386	12.2	20			2774.75	0.23
Afr. #29	276	1058	2.8	21				0.38
Afr. #40	136	620	6.8	22				0.98
Afr. #39	144	692	6.7	23				9.11
Afr. #17	151	633	5.6	25				49.27
Afr. #38	171	862	5.9	26				2.60
Afr. #28	167	854	6.2	27				2.43
Afr. #42	236	1110	4.0	30				78.79
Afr. #41	296	1620	3.7	40				175.14

well and allows to solve even the hardest instances within minutes. In fact, for almost all instances that the ILP was able to solve at all, the running time was below the timer resolution of 100 ms.

For both improvements, the savings in running time can be completely explained by the reduced number of flow augmentations.

4.6.3.2 Synthetic data from computational biology

In this section we examine solving the Minimum Fragment Removal problem [Panconesi and Sozio 2004] with Vertex Bipartization. The motivation is similar as in Section 4.6.3.1, except that the goal is to remove the minimum number of fragments (presumably those that contain read errors) to obtain

Table 4.2: Running times in seconds for different algorithms for synthetic Minimum Fragment Removal instances [Panconesi and Sozio 2004]. Here, c is a model parameter. Each entry is an average over 20 instances.

c	n	m	$\lvert X\rvert$	ILP	Reed	OCC-Gray	OCC-Enum2Col
2	25	23	1.4	0.02	0.00	0.00	0.00
3	50	61	3.5	2.00	0.00	0.00	0.00
4	82	116	6.1	224.33	0.05	0.00	0.00
5	112	173	8.3		1.17	0.05	0.02
6	143	253	10.4		39.86	1.64	0.16
7	174	321	11.6		7245.40	254.10	1.10
8	211	431	14.8			627.84	4.54
9	243	561	17.9				181.48
10	289	710	21.0				186.36
11	328	839	23.3				2493.82

consistent data.

We generate synthetic Vertex Bipartization instances using a model suggested by Panconesi and Sozio [2004]. A random binary string h_1 of length n is generated, and h_2 is generated as a copy of h_1 where a proportion of d bits is randomly flipped. These strings represent the two copies of the haplotype. Next, fragments are generated by breaking h_1 and h_2 each into k pieces by selecting $k-1$ breakpoints randomly. The fragment generation process is repeated c times, such that every position in h_1 or h_2 occurs c times in some fragment. Finally, each fragment is mutated by flipping each bit with probability p. From the fragments, the conflict graph is constructed, where the fragments are the vertices and an edge is drawn between two fragments if they differ at some position.

Following Panconesi and Sozio [2004], we choose the parameters $n = 100$, $d = 0.2$, $k = 20$, $p = 0.02$, and c varying (see Table 4.2).

The results are consistent with those of Section 4.6.3.1. The ILP is outperformed by the iterative compression algorithms; for OCC-Gray, we get a speedup by a factor somewhat below $|X|$ when compared to "Reed". The speedup from employing OCC-Enum2Col is very pronounced, but still far below the speedup observed in Section 4.6.3.1. A plausible explanation is the lower average vertex degree of the input instances; we examine this further in Section 4.6.3.3. Note that even with all model parameters constant, running times varied by a factor of up to several orders of magnitude for all algorithms for different random instances.

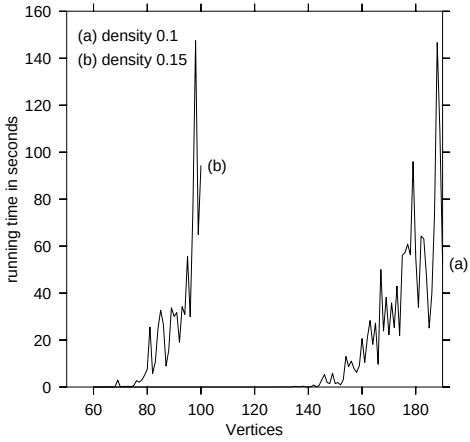


Figure 4.21: Running time of OCC-Enum2Col (Section 4.6.2.3) for random graphs of different density ($n = 300$). Each point is the average over at least 30 runs.

4.6.3.3 Random graphs

The previous experiments have established OCC-Enum2Col as best performing algorithm. Therefore, we now focus on charting its tractability border. We use the following method to generate random graphs with given number of vertices n, edges m, and odd cycle cover size at most k: Pre-allocate the roles "black" and "white" to $(n-k)/2$ vertices each, and "odd cycle cover" to k vertices; select a random vertex and add an edge to another random vertex consistent with the roles until m edges have been added.

In Figure 4.21, we display the running time of OCC-Enum2Col for different sizes of the odd cycle cover and different graph densities for graphs with 300 vertices. Note that the actual optimal odd cycle cover can be smaller than the one "implanted" by our model; the figure refers to the actual odd cycle cover size k.

At an average degree of 3, the growth in the measurements closely matches the one predicted by the worst-case complexity $O(3^k)$. For the average degree 16, the measurements fit a growth of $O(2.6^k)$, and for average degree 64, the growth within the observed range is about $O(1.6^k)$. This demonstrates the effectiveness of OCC-Enum2Col for dense graphs, at least in the range of values of k we examined.

The best way to improve the presented programs further for practical applicability seems the incorporation of data reduction rules. In particular, Wernicke

[2003] reported them to be most effective for sparse graphs. This makes a combination with OCC-Enum2Col (Section 4.6.2.3) attractive, since in contrast, this algorithm displays the worst performance for sparse graphs.

4.6.4 Outlook

As an alternative to the inductive mode of building up a solution, the compression routine can also be employed in a more straight-forward manner by simply trying to compress an initial heuristic solution (e.g., from Abdullah [1992] or Wernicke [2003]) until it cannot be compressed anymore. However, this leads to a much worse combinatorial explosion: even if there was a factor-c approximation, the running time would be $O(4^{ck}) \cdot n^{O(1)}$.

Fernau [2006b] suggests to solve Vertex Bipartization by first using a branching algorithm to get rid of triangles (C_3's), and then using the fact that the remaining instance requires with high probability at most one more vertex deletion to become bipartite. This would allow to solve Vertex Bipartization as fast as 3-Hitting Set, that is, in $2.076^k \cdot n^{O(1)}$ time. Unfortunately, the probability argument only holds over an equal distribution of graphs; for example, it might be wrong for most sparse graphs. Still, it might be interesting to see whether in practice the remaining instances actually have a very small parameter and can thus be solved quickly. Also, a 3-Hitting Set kernelization [Abu-Khzam 2007] might be able to remove some vertices upfront in instances with very many triangles.

It would also be interesting to see whether the quest for better data reduction leads to a problem kernel, as it was recently achieved for the related problem Feedback Vertex Set [Burrage et al. 2006, Bodlaender 2007].

At the Dagstuhl Seminar 07281 (Structure Theory and FPT Algorithmics for Graphs, Digraphs and Hypergraphs, 2007), Henning Fernau (Universität Trier) suggested to examine the parameterization of 3-Coloring by the size of the smallest color partition. Since deleting any color class from a 3-coloring leaves a bipartite graph, we can reformulate this problem as follows: find a vertex bipartization set X of size at most k such that X is an independent set. Clearly, this problem is similar to Vertex Bipartization, and similar techniques might apply.

4.7 Outlook

Up to now, the iterative compression framework has been confined to the setting of graph modification problems for hereditary graph classes. Therefore, one of the most interesting challenges is to find an application that lies outside this area.

Dealing with graph modification problems for non-hereditary graph classes would probably require a more sophisticated induction than simply adding vertex-by-vertex or edge-by-edge. A possibility is to introduce annotations. For example, it has been suggested to try to find an iterative compression algorithm for Cluster Editing with "don't care" edges, that is, edges which may be edited at zero cost [Bodlaender et al. 2006]. We can then use induction by initially having all vertex pairs as "don't care", and then converting them one-by-one to their actual state (edge or non-edge).

An even more interesting challenge is to leave the realm of graph problems completely. For example, recently it has been shown that there is an iterative compression algorithm for Almost 2-SAT, that is, the problem of deleting the minimum number of clauses from a 2CNF-formula such that it becomes satisfiable [Razgon and O'Sullivan 2008]. Similar problems might give good targets for iterative compression.

Chapter 5

Color-coding

The color-coding method was introduced by Alon et al. [1995] as a randomized method for finding certain small subgraphs of size k in a graph. The method can be derandomized with running time $O(c^k m)$ for some c, implying fixed-parameter tractability.

Although the method was lauded for its elegance and potential practicability due to modest running time and no large hidden constants, only recently have implementations been tried. A main application is to find candidates for signaling pathways in protein interaction networks, which can be modeled as a Minimum-Weight Path problem (see Section 2.4). Using color-coding, Scott et al. [2006] could find candidates for signaling pathways with up to 10 proteins within a few hours.

In this chapter, we show how to speed up color-coding both from a worst-case perspective and from a practical viewpoint. Our experiments show a speed-up of several orders of magnitude compared to the work of Scott et al. [2006]. Some of the improvement comes from a simple direct modification of the algorithm (Section 5.3.1); some from the use of heuristic evaluation functions, which help to avoid the search of large parts of the search space (Section 5.3.2); and some come from carefully tuned data structures (Section 5.3.3). With the speedups, for basically all parameter settings relevant to the search for signaling pathways, results can be obtained within seconds (Section 5.4). This allows the interactive exploration of such pathways. For this, the graphical user interface Faspad was implemented (Section 5.5).

In summary, the results of this chapter show that color-coding is a viable method for a wide range of NP-hard motif search problems, that is, problems where one is looking for small subgraphs with certain properties.

5.1 Known results

Introducing color-coding, Alon et al. [1995] have shown that finding a simple path of length k in a graph (Longest Path) can be done with high probability in $O(5.44^k m)$ time. The method can be derandomized with running time $O(c^k m)$ for some (very large) c, implying that this NP-hard problem is fixed-parameter tractable. They further gave FPT algorithms for finding a k-vertex cycle or a given subgraph of bounded treewidth.

Although the technique was described as elegant and fundamental for constructing FPT algorithms, the field lay dormant for some time, and has only recently seen a revival. Several authors used color-coding to get improved FPT results for set packing (given a number of sets, finding a collection of disjoint sets whose union has maximum size) [Fellows et al. 2004, Koutis 2005] and graph packing (packing a maximum number of vertex-disjoint copies of a graph into another graph) [Fellows et al. 2004, Mathieson et al. 2004, Prieto and Sloper 2006, Liu et al. 2006]. Marx [2005] used color-coding to get FPT results for certain constraint satisfaction problems. Betzler [2006] showed how to use color-coding to find a subtree of minimum weight isomorphic to a query tree of k vertices in randomized $O(8.16^k km)$ time. Dost et al. [2007] extended this setting by taking vertex matching weights, insertions, and deletions into account. They further give some details on how to implement color-coding for query graphs with small treewidth. Fellows et al. [2007a] used color-coding to find in a vertex colored graph a connected set of vertices whose colors match a specified set of colors.

Alon et al. [1995] describe how to derandomize color-coding using a "k-perfect family of hash functions" of size c^k for some $c > 8000$. The size of this family is a factor in the exponential base of the running time. Recently, Chen et al. [2007b] have improved c to 6.4. Unfortunately, there is also a lower bound of e [Nilli 1994], implying that a derandomized algorithm cannot have a better exponential base of the running time than the randomized method.

The color-coding method has also inspired a recent approach called "divide-and-color", which is based on divide-and-conquer, and was independently developed by two groups [Kneis et al. 2006, Chen et al. 2007b]. This method can solve Longest Path with high probability in $O(4^k k^{3.42} m)$ time. In addition to the thus improved exponential part of the running time, the algorithm can be derandomized more efficiently [Chen et al. 2007b]. Divide-and-color has further been applied to the t-Dominating Set problem [Kneis et al. 2007]. Another color-coding-like technique termed "random separation" has been suggested by Cai et al. [2006]. This approach seems not quite as generally employable, since it requires assumptions about the size of the neighborhood of a subgraph sought for.

Only recently has it been attempted to implement color-coding methods.

Raymann [2004] gave an implementation for Longest Path. Unlike later works, his implementation cannot deal with edge weights. Just finding a path of length k is a much easier problem than finding a minimum-weight path, since there are usually many paths of length k in real-world instances. Therefore, it is hard to compare this implementation to other works. The use of color-coding in bioinformatics has been initiated by Scott et al. [2006]. They modeled the problem of finding signaling pathways in protein interaction networks as Minimum-Weight Path problem (see Section 2.4) and further gave an algorithm for k-Cardinality Tree (finding a subtree of minimum weight with k edges) running in $8.16^k \cdot n^{O(1)}$ time. Shlomi et al. [2006] extend the approach to Pathway Query (given a pathway, find one that best matches it in the given graph, allowing for deletions and insertions). A very similar setting was applied by Mayrose et al. [2007] to find where antibodies dock to proteins. Dost et al. [2007] gave results for querying tree structures in protein interaction networks. Cappanera and Scutellà [2007] used color-coding for the Balanced Paths problem, that is, to find p paths such that the difference in cost between the longest and the shortest is minimized. Borndörfer et al. [2007] used color-coding to get feasible LP formulations for line planning problems in public transport. Finally, Björklund et al. [2007] gave an elegant method how to speed up the dynamic programming step of color-coding for k-Cardinality Tree from $3^k \cdot n^{O(1)}$ to $2^k \cdot n^{O(1)}$.

Deshpande et al. [2007] independently discovered the trick of increasing the number of colors used, as it is explained in Section 5.3.1. They also recommend using $1.3k$ colors; however, they only derived an upper bound on the exponential running time of 4.5^k, while we show a bound of 4.32^k (Theorem 5.3).

5.2 Basic method

The central idea of color-coding is to randomly color each vertex of the graph with one of several colors and to "hope" that the vertices in the subgraph searched for obtain a particular color pattern; for example, that each vertex obtains a different color. Under the assumption that this happens, the task of finding the subgraph is greatly simplified; in particular, it can be found in FPT time. Of course, most of the time the target structure will not actually be colored favorably. Therefore, we have to repeat the process of randomly coloring and then searching (called *trial*) many times with a fresh coloring until with sufficiently high probability at least once our target structure is colored favorably. Since the number of trials also depends only on k (albeit exponentially), the complete algorithm runs in FPT time. Figure 5.1 shows pseudo-code for this algorithm skeleton.

We now present two simple color-coding algorithms for the Longest Path

Algorithm: ColorCoding($G = (V, E)$)
1 **repeat** a sufficient number of times**:**
2 **for each** $v \in V$**:**
3 color v randomly
4 **if** Trial(G)**:**
5 **return true**
6 **return false**

Figure 5.1: Algorithm skeleton for color-coding

problem. They do not have competitive running times, but rather serve as introductory examples. To make the presentation simpler, in this chapter, when we talk about paths, we always mean *simple* paths, that is, paths that do not contain a vertex more than once. Further, the *length* of a path is its number of vertices (not edges).

Longest Path
Instance: An undirected graph $G = (V, E)$ and an integer $k > 0$.
Question: Does G contain a simple path of k vertices?

The difficulty here comes from the demand of *simple* paths. Without this requirement, we could just traverse a single edge $k - 1$ times.

The probably most simple color-coding algorithm for Longest Path is as follows:

Color-coding Scheme 5.1. *For a trial, color each vertex randomly with a number from* 1 *to* k*. The hope is that the vertices of a* k*-vertex path obtain exactly the colors* 1 *through* k*, from start vertex to end vertex.*

Assuming we have such a coloring, it is easy to check whether there is a path of length k: from $c = k - 1$ down to 1, delete each vertex with color c that is not connected to a vertex of color $c + 1$. It is easy to see that after this, there is a vertex of color 1 left iff there is a path with colors 1 through k.

It remains to state what a sufficient number t of trials is. The following lemma shows how to calculate the number of trials required to achieve a desired error probability ε from the success probability of a trial.

Lemma 5.1. *If a trial of color-coding succeeds with probability* p*, then* $t \geqslant -\ln \varepsilon / p$ *trials are needed to achieve an error probability of at most* ε*.*

Proof. The algorithm only fails if all trials fail. Thus, to get an error probability of at most ε, we need

$$(1 - p)^t \leqslant \varepsilon. \tag{5.1}$$

Using $1 + x \leqslant \exp(x)$, which holds for all x and is a very good approximation for x of small absolute value, we get

$$\exp(-pt) \leqslant \varepsilon \tag{5.2}$$
$$\iff \qquad t \geqslant -\ln \varepsilon / p. \tag{5.3}$$

□

We arrive at the following proposition.

Proposition 5.1. *Using Color-coding Scheme 5.1,* Longest Path *can be solved in* $O(k^k \cdot m \cdot (-\ln \varepsilon))$ *time with error probability at most* ε.

Proof. The probability that with a particular coloring we find a path of length k if there is one is at least $p := 1/k^k$ (higher if there is more than one path of length k). By Lemma Lemma 5.1, $t = \lceil k^k \cdot (-\ln \varepsilon) \rceil$ trials suffice to solve Longest Path with error probability at most ε. A single trial can be done in linear time. □

We have thus already obtained a (randomized) fixed-parameter running time, albeit with a quite bad combinatorial explosion with respect to k. Note, however, that the error probability affects the running time only logarithmically. We can thus afford very small error probabilities at the cost of only a small extra factor in the running time. For example, demanding the error probability to be as low as that of a random hardware memory error caused by cosmic rays imposes a factor of about 4 in a plausible setting (no error-correcting memory; 10^{-12} errors per bit per hour [Tezzaron Semiconductor 2004]; computation running for 10 minutes using 64 MB). Thus, it seems quite unlikely that one would want to use a derandomized version in practice.

We can improve the running time by hoping for a more likely favorable coloring, at the cost of a slightly more involved trial routine.

Color-coding Scheme 5.2. *For a trial, randomly assign to the vertices of* G *a permutation of the numbers* $1, \ldots, n$*, such that each permutation is equally likely. The hope is that the vertices in a* k*-vertex path* $v_1, v_2, \ldots, v_k$ *will receive increasing values, that is, we have* $\text{color}(v_1) < \text{color}(v_2) < \cdots < \text{color}(v_k)$.

The difference to Color-coding Scheme 5.1 is that the colors of the path do not have to be contiguous, but any increasing sequence will do. Finding paths of length k that are colored by increasing colors is still much easier than the general problem. If we are only interested in such paths, we can give each edge a direction going from the vertex with the lower color to the vertex with the higher color. This directed graph is clearly acyclic. We thus have to solve the Longest Path problem in directed acyclic graphs, which can be done in polynomial time using dynamic programming.

Since dynamic programming easily allows us to consider weighted problems, and important applications ask for this, we extend our focus to Minimum-Weight Path (see Section 2.4 for applications and previous work on Minimum-Weight Path). Note that vertex weights can be dealt with by a reduction to edge weights in directed graphs in the same way as for minimum cut (Section 4.6.2.2); all algorithms in this section are easily adaptable to the directed case.

Minimum-Weight Path
Instance: An undirected graph $G = (V, E)$ with edges weighted by $w : E \to \mathbb{Q}+$ and an integer $k > 0$.
Task: Find a simple path $v_1, \ldots, v_k$ of k vertices with minimum weight, that is, that minimizes $\sum_{i=1}^{k-1} w(\{v_i, v_{i+1}\})$.

For ease of presentation, we only deal with determining the weight of an optimal solution; we examine some ways to retrieve the path that realizes this weight in Section 5.3.3. We use the following obvious decomposition.

Observation 5.1. *A path with $k > 1$ vertices that ends in v can be decomposed into a path of $k-1$ vertices that does not contain v and a single edge connecting v to one of its neighbors.*

Observation 5.1 suggests to use a dynamic programming table $T[v, l] : v \in V, 1 \leqslant l \leqslant k$, which stores for each $v \in V$ the weight of a minimum-weight path that has l vertices and ends in v. We can then use the following recurrence:

$$T[v, l] = \min_{u|(u,v)\in E} T[u, l-1] + w(\{u, v\}). \tag{5.4}$$

The base case is simply $T[v, 1] = 0$ for all $v \in V$. After the table has been filled in, the optimum value can be retrieved as $\min_{v\in V} T[v, k]$. For the recurrence to be workable, we must be able to fill the table in such an order that accessed values have already been calculated. This can be ensured if we fill in the table in the order of a topological sort of G. Further, we must make sure that we do not access a value corresponding to a path that already contains v. Here, this needs no further work, because only vertices distinct from v have been considered before filling in an entry for v.

Example 5.1. *Figure 5.2 shows an example. In the trial shown on the right, the random coloring is a lucky coloring, since for $k = 4$, the minimum-weight path* cbda *is colored with ascending colors. The calculations that lead to the discovery of this optimal path*

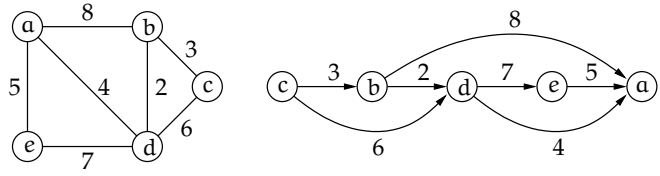


Figure 5.2: Example for Color-coding Scheme 5.2 (left: input instance; right: directed acyclic graph generated in a particular trial, which sorted the vertices as $c < b < d < e < a$)

of weight 9 are (other table entries not shown):

$$T[c,1] = 0 \tag{5.5}$$
$$T[b,2] = \min\{T[c,1] + w((c,b))\} = 3 \tag{5.6}$$
$$T[d,3] = \min\{T[c,2] + w((c,d)), T[b,2] + w((b,d))\} = 5 \tag{5.7}$$
$$T[a,4] = \min\{T[b,3] + w((b,a)), T[d,3] + w((d,a)), T[e,3] + w((e,a))\} = 9. \tag{5.8}$$

Proposition 5.2. *Using Color-coding Scheme 5.2,* Minimum-Weight Path *can be solved in* $O(k! \cdot km \cdot (-\ln \varepsilon))$ *time with error probability at most* ε.

Proof. There are only two colorings of a k-vertex path that are favorable, out of $k!$ possible. Thus, the trial success probability is $p = 2/k!$. Since Lemma 5.1 also holds here, $t = \lceil k!/2 \cdot (-\ln \varepsilon) \rceil$ trials suffice. The dynamic programming table has kn entries, and amortized over all entries, km lookups of older entries need to be done. □

We now have an exponential part of $k!$ for color-coding. It is possible to further improve this to 5.44^k, again by weakening the demand put on the coloring. This is the scheme proposed by Alon et al. [1995].

Color-coding Scheme 5.3. *For a trial, color each vertex randomly with a number from* 1 *to* k. *The hope is that a minimum-weight path becomes* colorful, *meaning all of its vertices obtain different colors.*

Here, we will not be able to give a polynomial-time algorithm for a single trial. Instead, the problem of finding a colorful k-vertex path is solved in modestly exponential time by dynamic programming, following the path decomposition Observation 5.1. The extra complication is because we cannot as easily ensure that when accessing shorter paths to form paths ending in v, they do not already use v (that is, that the path stays simple). For the general (uncolored) problem,

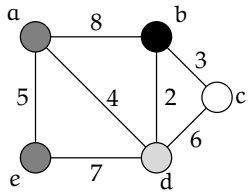


Figure 5.3: Example for Color-coding Scheme 5.3

we would need one table entry for each possible combination of vertices in a path and each end vertex, resulting in a table size of roughly $O(n^k)$. When looking only for a colorful path, however, it suffices to store the set of colors a path uses: if the color of v does not occur in some path, we can safely append v without introducing loops. That is, we use a table $T[v, S] : v \in V, S \subseteq \mathcal{P}(\{1, \ldots, k\})$, where an entry $T[v, S]$ stores the minimum weight of a path that ends in v and whose vertices have exactly the colors in S. The length of the path does not need to be stored explicitly; since we look only for colorful paths, it is $|S|$.

To fill in the table, we can use the following recurrence, analogous to (5.4):

$$T[v, S] = \min_{\{u,v\} \in E} T[u, S - \text{color}(v)] + w(\{u, v\}). \tag{5.9}$$

The base case is easy: for each $v \in V$ and $c \in \{1, \ldots, k\}$, set

$$T[v, \emptyset] = \infty \tag{5.10}$$

$$T[v, \{c\}] = \begin{cases} 0 & \text{if color}(v) = c \\ \infty & \text{otherwise.} \end{cases} \tag{5.11}$$

Further, if we fill the table in order of increasing size of S, we only need to access entries already filled in. After the table is filled in completely, we can retrieve the optimum value as $\min_{v \in V, |S|=k} T[v, S]$.

We give an example of a trial (Figure 5.3). Here, also a lucky coloring is shown, that colors the optimal path $cbda$ of weight 9 colorful. First, the base case is handled:

$$T[a, \{\bullet\}] = 0; T[b, \{\bullet\}] = 0; T[c, \{\circ\}] = 0; T[d, \{\circ\}] = 0; T[e, \{\bullet\}] = 0. \tag{5.12}$$

Then, the dynamic programming recursion is applied, in increasing order

of $|S|$ (we show only non-∞ entries):

$$
\begin{array}{lll}
T[a,\{\circ,\bullet\}] = 4; & T[a,\{\bullet,\bullet\}] = 8; & \\
T[b,\{\circ,\bullet\}] = 3; & T[b,\{\circ,\bullet\}] = 2; & T[b,\{\bullet,\bullet\}] = 8; \\
T[c,\{\circ,\circ\}] = 6; & T[c,\{\circ,\bullet\}] = 3; & \\
T[d,\{\circ,\circ\}] = 6; & T[d,\{\circ,\bullet\}] = 4; & T[d,\{\circ,\bullet\}] = 2; \\
T[e,\{\circ,\bullet\}] = 7; & & \\
T[a,\{\circ,\circ,\bullet\}] = 10; & T[a,\{\circ,\bullet,\bullet\}] = 11; & T[a,\{\circ,\bullet,\bullet\}] = 6; \\
T[b,\{\circ,\circ,\bullet\}] = 8; & T[b,\{\circ,\bullet,\bullet\}] = 6; & \\
T[c,\{\circ,\circ,\bullet\}] = 10; & T[c,\{\circ,\circ,\bullet\}] = 5; & T[c,\{\circ,\bullet,\bullet\}] = 11; \\
T[d,\{\circ,\circ,\bullet\}] = 5; & T[d,\{\circ,\bullet,\bullet\}] = 10; & \\
T[e,\{\circ,\circ,\bullet\}] = 13; & T[e,\{\circ,\bullet,\bullet\}] = 9; & \\
T[a,\{\circ,\circ,\bullet,\bullet\}] = 9; & & \\
T[b,\{\circ,\circ,\bullet,\bullet\}] = 13; & & \\
T[c,\{\circ,\circ,\bullet,\bullet\}] = 9; & & \\
T[d,\{\circ,\circ,\bullet,\bullet\}] = 15; & & \\
T[e,\{\circ,\circ,\bullet,\bullet\}] = 12. & &
\end{array}
$$

The weight of the optimal solution is then retrieved from $T[a,\{\circ,\circ,\bullet,\bullet\}]$ or $T[c,\{\circ,\circ,\bullet,\bullet\}]$ (the optimal path will always be found twice, once from start to end and once from end to start).

Theorem 5.1 (Alon et al. [1995])**.** *Using Color-coding Scheme 5.3,* Minimum-Weight Path *can be solved in* $O(1/\sqrt{k}\exp(k)\cdot 2^k\cdot m\cdot(-\ln\varepsilon)) = O(5.44^k\cdot m\cdot(-\ln\varepsilon))$ *time with error probability at most* ε*.*

Proof. We first need to estimate the probability p that a path is colorful in a trial. There are k^k ways to color a k-vertex path, of which $k!$ are favorable. Thus, using the asymptotically very close Stirling approximation

$$k! > \sqrt{2\pi k}\left(\frac{k}{e}\right)^k, \tag{5.13}$$

we get

$$p = \frac{k!}{k^k} > \frac{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k}{k^k} = \sqrt{2\pi k}\exp(-k). \tag{5.14}$$

By Lemma 5.1, we thus need $t = \lceil 1/(\sqrt{2\pi k}\exp(k)\cdot(-\ln\varepsilon))\rceil$ trials.

To estimate the time needed for a trial, we note that for each of the 2^k possible values of S, there are m terms evaluated as argument of the minimum. The time for evaluating one term depends on the data structure and machine model: if we assume that the table is stored in "flat" format and address offset calculations can be done in constant time, then it can be done in constant time; otherwise (as stated, e. g., by Alon et al. [1995]) we get an additional factor of k. □

5.2.1 Variations of Minimum-Weight Path

A particularly appealing aspect of the color-coding method is that it can be easily adapted to many practically relevant variations of Minimum-Weight Path. In particular, extra constraints on the path can be taken into account. For example, the set of vertices where a path can start and end can be restricted (such as to force it to start in a membrane protein and end in a transcription factor [Scott et al. 2006]). This can be done by restricting the base case (5.11) to only set entries to 0 for vertices in the start set, and by only examining vertices from the target set in the final loop that retrieves the optimal value. A variant with more involved extra constraints is presented in Section 5.2.2; another example is given by Mayrose et al. [2007].

Further, not only a minimum-weight path can be sought after but rather a collection of low-weight paths. This is useful when modeling scenarios where more than one path is sought for, or when there is additional information not modeled in the instance, which requires a number of good candidates for manual inspection. This can be simply done by storing not just one optimal path, but a fixed number of paths with the highest score. It seems difficult, though, to provably find with high probability the set of the b best paths: while a suboptimal path p will also be colorful in some trial with high probability, there might be other paths with lower weight that have many vertices in common with p and are therefore also colorful with high probability, which would make the dynamic programming not find p.

However, typically, one is not interested in small variations of a single path anyway; rather, one wants a diverse set of candidates for manual inspection. For example, Scott et al. [2006] demand that the paths must differ in, say, 30 % of the vertices. This can be done by not accepting a path if there is already a path with better score and more than 30 % similarity, and by discarding all paths with more than 30 % similarity to a newly inserted path. While it still seems hard to prove anything about this scenario (in particular since the solution is no longer necessarily unique, even if all weights are distinct), it is quite plausible that it will with high probability produce optimal solution sets in the sense that no path could be inserted that would improve the average score.

	a	b	c	d	e
q_1	7	5	4	5	1
q_2	4	4	7	2	6
q_3	8	1	8	4	5
q_4	6	6	3	6	4

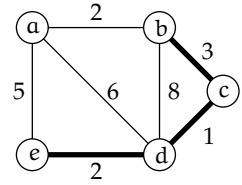


Figure 5.4: Example for Pathway Query with $l = 4$ and $N_{ins} = N_{del} = 1$. Query sequence is q_1, q_2, q_3, q_4. Left: vertex match table h. Right: weighted graph G. The optimal path is $P = edcb$ (*bold edges*) with matching $M : q_1 \mapsto e, q_2 \mapsto d, q_3 \mapsto b, q_4 \mapsto \perp$. The cost of this solution is $2 + 1 + 3 + 1 + 2 + 1 = 10$.

5.2.2 Pathway Query

Recently, it has been demonstrated that pathway queries to a network, that is, the task of finding a pathway in a network that is as similar as possible to a query pathway, can be handled with color-coding [Shlomi et al. 2006] (see Section 2.4 for details on the motivation). We recall the somewhat involved, but natural formalization:

Pathway Query
Instance: An undirected graph $G = (V, E)$ with edges weighted by $w : E \to \mathbb{Q}_+$, a length-l query sequence $Q = q_1, \ldots, q_l$, a match weight function $h : \{q_1, \ldots, q_l\} \times V \to \mathbb{Q}_+$, and two nonnegative integers N_{ins} and N_{del}.
Task: Find an *alignment*, that is, a path $P = p_1, \ldots, p_k$ in G together with a mapping M from $\{q_1, \ldots, q_l\}$ to $\{p_1, \ldots, p_k\} \cup \{\perp\}$ such that no vertex in P has more than one preimage. The alignment must have at most N_{ins} *insertions* (that is, vertices in P that have no preimage in Q) and at most N_{del} *deletions* (that is, vertices in Q that are mapped to $\perp$). Further, the weight of the alignment must be minimal, that is, one must minimize

$$\sum_{i=1}^{l-1} w(p_i, p_{i+1}) + \sum_{\substack{1 \leqslant i \leqslant l \\ M(q_i) \neq \perp}} h(q_i, M(q_i)). \tag{5.15}$$

Figure 5.4 shows an example. Note that Pathway Query is a generalization of Minimum-Weight Path and becomes equivalent to this problem in the special case where the match weight function h is unit and $N_{ins} = N_{del} = 0$.

Shlomi et al. [2006] show how to solve Pathway Query by color-coding. The basic process is the same as for Minimum-Weight Path (Color-coding

Scheme 5.3): the input graph is randomly colored with $k := l + N_{ins}$ colors and it is hoped that the optimal path becomes colorful in the process. However, the dynamic programming recurrence (5.9) from solving Minimum-Weight Path needs to be adapted in several ways to account for the more general problem formulation of Pathway Query:

- New dimensions are added to the dynamic programming table to track the number of deletions θ and the number of matched vertices i.
- The vertex match weights are taken into account.
- New recurrences for the process of insertion and deletion are added.

Thus, a table entry $T[v, i, \theta, S]$ contains the minimum weight of a partial alignment that matches $q_1, \ldots, q_i$, ends at v, contains θ deletions, and uses exactly the colors in S for the matching targets of $q_1, \ldots, q_i$. The precise recurrences are:

$$T[v, i, \theta, S] = \min \begin{cases} T[u, i-1, \theta, S - \text{color}(v)] + w(u, v) + h(q_i, u) & \{u, v\} \in E \\ T[u, i, \theta, S - \text{color}(v)] + w(u, v) & \{u, v\} \in E, |S| - i < N_{ins} \\ T[v, i-1, \theta-1, S] & \theta < N_{del}. \end{cases} \tag{5.16}$$

The first case corresponds to a match. The number of matched vertices i is increased, and the costs for the matched edge and the matching of the two vertices are added. The second case describes an insertion; this can only be done if the number of already inserted vertices ($|S| - i$) is not too large. Finally, the third case describes a deletion.

Theorem 5.2 (Shlomi et al. [2006])**.** *Using Color-coding Scheme 5.3,* Pathway Query *can be solved in* $O(1/\sqrt{k}\exp(k) \cdot 2^k \cdot N_{ins}N_{del}m \cdot (-\ln \varepsilon)) = O(5.44^k \cdot N_{ins}N_{del}m \cdot (-\ln \varepsilon))$ *time with error probability at most* ε*.*

Proof. The analysis is the same as in the proof of Theorem 5.1, except that any color set S can occur in combination with $\theta = 0, \ldots, N_{del}$ and with $i = |S| - N_{ins}, \ldots, |S|$. □

Details of the implementation of calculating (5.16) are given in Section 5.3.4.

5.3 Algorithm engineering

This section presents several algorithmic improvements for color-coding that lead to large savings in time and memory consumption. Whereas most of

the improvements in Sections 5.3.2 and 5.3.4 are of a heuristic nature, the improvement in Section 5.3.1 makes color-coding also more efficient in a worst-case scenario. Most of our improvements are applicable to color-coding in general and not restricted to the Minimum-Weight Path and Pathway Query problems that our implementation covers.

As is standard practice in dynamic programming, we do not allocate memory for the complete table and evaluate the recurrence (5.9) recursively; rather, we work inductively starting from an initial set of entries. This way, we do not need to allocate memory for the whole table, but can use a sparse data structure where only entries that were actually calculated are stored; all others are implicitly ∞. This saves both time and memory.

More precisely, based on (5.11), we seed the table with all entries corresponding to a one-vertex path ending at a vertex v:

for each $v \in V$**:** $T[v, \text{color}(v)] \leftarrow 0$

Layer i contains the table entries with color sets of cardinality i. Thus, we already have the first layer. Then, for $i = 2, \ldots, k$, we generate layer i from layer $i-1$:

for each $T[v, S]$ in the $(i-1)$-th layer**:**
  for each $u \mid \{v, u\} \in E$**:**
    $T[v, S + \text{color}(u)] \leftarrow T[v, S] + w(\{v, u\})$

The running time of this implementation is $O(\sum_{i=1}^{k} 2^i m) = O(2^k m)$, and thus the same as a direct implementation of (5.9).

We can discard a layer after the next layer has been computed to save memory. However, this requires extra memory to carry along enough information in each entry to reconstruct a solution; we show in Section 5.3.3 how to do this efficiently.

5.3.1 Increasing the number of colors

Like the improvements in going from Color-coding Scheme 5.1 to Color-coding Scheme 5.2 and from Color-coding Scheme 5.2 to Color-coding Scheme 5.3, this improvement comes from a trade-off between the number of trials and the time required for one trial. The idea is to increase the number of colors used. More colors means a path is more likely to become colorful, which means fewer trials. On the other hand, the dynamic programming step of each single trial takes longer: the table size (and thus the running time) goes from $O(2^k m)$ to $O(2^{k+x} m)$ for x extra colors, that is, it doubles for every extra color.

Table 5.1 illustrates the gain from adding more colors. For example, when adding a single color, it becomes 3.5 times more likely that a path is colorful, which (by Lemma 5.1) means 3.5 times fewer trials. This well overcompensates

Table 5.1: Probability p for obtaining a colorful path using $k + x$ colors ($k = 8$, $\varepsilon = 0.001$)

x	0	1	2	3	4	5
p	0.0024	0.0084	0.0181	0.0310	0.0464	0.0636
trials	2875	820	381	223	149	109

the increase by a factor of 2 for the dynamic programming. However, there are diminishing returns: when going from 4 extra colors to 5 extra colors, we only gain a factor of about 1.4, not sufficient to compensate the extra time needed for the dynamic programming. Somewhere in-between is an optimum (in this case, $x = 2$).

To determine this optimum, we need to look at the probability p_x that a path of length k colored with $k + x$ colors is colorful. The probability is

$$p_x = \frac{\binom{k+x}{k} \cdot k!}{(k+x)^k} = \frac{(k+x)!}{x!(k+x)^k} = \prod_{i=1}^{k} \frac{i+x}{k+x}, \tag{5.17}$$

because there are $(k + x)^k$ ways to color k vertices with $k + x$ colors and of these exactly $\binom{k+x}{k} \cdot k!$ use mutually different colors. The optimal x is then the lowest x where p_{x+1}/p_x drops below 2, because if this value is below 2, going from x to $x + 1$ colors cannot compensate the factor of 2 increase from the larger dynamic programming table.

It seems hard to determine this point analytically; numerical experiments show that asymptotically, this happens for $x \approx 0.3k$. Using $x = 0.3k$, the exponential part of 5.44^k in Theorem 5.1 can be improved:

Theorem 5.3. *Using Color-coding Scheme 5.3 with* $1.3k$ *colors,* Minimum-Weight Path *can be solved in* $O(4.32^k \cdot m \cdot (-\ln \varepsilon))$ *time with error probability at most* ε.

For the proof, we refer to Wernicke [2006, Theorem 4.3].

Analogously to this theorem for Minimum-Weight Path, the worst-case running time that is required to solve Pathway Query for a $(k - N_{ins})$-vertex query path can be improved to $O(4.32^k \cdot N_{ins} N_{del} m \cdot (-\ln \varepsilon))$ by setting the number of colors close to $1.3k$.

This improvement brings the exponential part of color-coding for Minimum-Weight Path from 5.44^k down to 4.32^k, and thus much closer to the 4^k of the divide-and-color method [Kneis et al. 2006, Chen et al. 2007b]. For a practical implementation, we can expect even more speedup than suggested by the worst-case estimation. While we could fix the number of colors at the worst-case

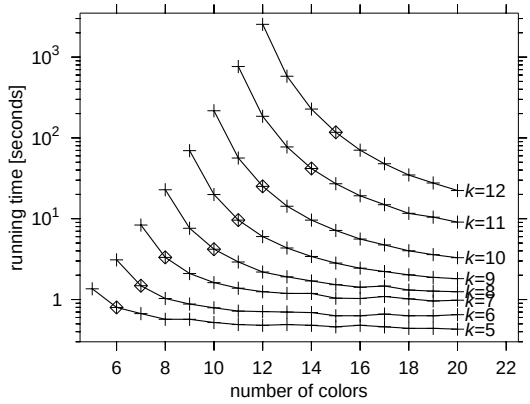


Figure 5.5: Running times for finding the 20 minimum-weight paths of different lengths k in the yeast protein interaction network of Scott et al. [2006]. Increasing the number of colors yields a speedup of up to two orders of magnitude. No heuristic evaluation function (Section 5.3.2) was used; the highlighted point of each curve marks the optimal choice when assuming worst-case trial running time.

optimum 1.3k, it is most likely beneficial to use even more colors, because various algorithmic tweaks and the underlying graph structure can keep the dynamic programming table very sparse, and thus the running time of a single trial stays substantially below the worst-case estimate. This in turn causes the increase in running time per trial by choosing more colors to be even more overcompensated by a decrease in the total number of trials needed, as is exemplified in Figure 5.5 for the case of Minimum-Weight Path. In fact, for a small path length of 8–10 we can choose the number of colors to be the maximum our implementation allows (that is, 31, due to the data structures explained in Section 5.3.3), and get by with a very small number of trials (≈15–30). Based on such observations, our implementation uses an adaptive approach to the number of colors, starting with the maximum of 31 and decreasing this in case a trial runs out of memory.

A disadvantage of this technique is the increased memory usage: in the worst case, memory usage increases by a factor of 2^x. However, this problem can be much mitigated using the techniques of the next section.

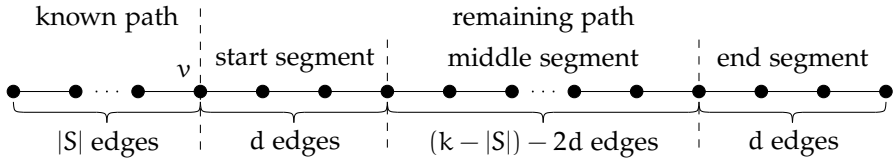


Figure 5.6: Calculation of a lower weight bound for a path with k edges when already $|S|$ of these edges are given

5.3.2 Heuristic evaluation functions

In a color-coding trial for solving Minimum-Weight Path, every vertex carries entries for up to 2^{k+x} color sets, each representing a partial colorful path with a certain weight. As explained at the beginning of Section 5.3, we fill in the table not recursively, but inductively, starting with the base case entries (5.11), and *expanding* entries to form entries in the next layer. Because each entry may get expanded, over several layers, to an exponentially large collection of new entries, pruning even a small fraction of them can lead to a significant speedup. The pruning strategy that we employ makes use of the fact that we are only looking for one minimum-weight path. As soon as we have found a solution candidate, we can always remove table entries where the weight of the corresponding partial path, when completed, is certain to exceed the weight of an already known length-k path.

Consider an entry $T[v, S]$ corresponding to some partial path. To obtain a length-k path, we need to append another $k - |S|$ edges. Thus, a trivial lower bound for the total weight of a length-k path expanded from this entry would be $T[v, S] + (k - |S|)w_{\min}$, where $w_{\min}$ is the minimum weight of any edge in the graph. We improve upon this simple bound by dividing the remaining path length not into single edges, but rather—as illustrated in Figure 5.6—into three segments, calculating a lower bound separately for each of them and summing up these bounds.

The lower bound calculation is prepared in a preprocessing phase by dynamic programming on the uncolored graph. There, we determine by dynamic programming for every vertex v and a range of lengths $1 \leqslant i \leqslant d$ the minimum weight $w_{\min}(v, i)$ of a path of i edges that starts at the vertex v. If the paths are restricted to end only in a certain set of "goal vertices" (for example, when the signaling pathway candidates are restricted to end in a transcription factor), we additionally determine the minimum weight $g_{\min}(v, i)$ of a path of i edges starting in v and ending in a goal vertex. After this preprocessing, to get a lower bound for the minimum weight of a path with $l < k$ edges starting in v and ending in a goal vertex, we can directly look up $g_{\min}(v, l)$ whenever $l \leqslant d$. Since

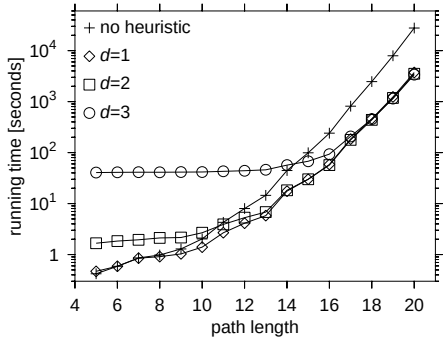


Figure 5.7: Running time comparison with heuristic evaluation functions for different values of d (seeking the 20 lowest-weight paths in the yeast network of Scott et al. [2006] that differ in at least 30% of participating vertices).

calculating $w_{\min}$ and $g_{\min}$ takes $O(n^d)$ time and space in the worst case, we generally have to choose $d < k$. We can still get a lower bound if $l > d$. For example, if $l = c \cdot d$ for some $c \geqslant 2$, we calculate

$$w_{\min}(v, d) + \frac{l - 2d}{d} \cdot \min_{u \in V} w_{\min}(u, d) + \min_{u \in V} g_{\min}(u, d). \tag{5.18}$$

If d does not evenly divide l, we add a suitable correction term for the middle segment. If $l < 2d$, we additionally try all ways of dividing the bound between $w_{\min}(v, l_1)$ and $\min_{u \in V} g_{\min}(u, l_2)$ for $l_1 + l_2 = l$.

Clearly, there is a trade-off between the time invested in the preprocessing (depending on d) and the time saved in the main algorithm. For the yeast network of Scott et al. [2006], setting $d = 2$ seems to be a good choice with an additional second of preprocessing time. For $d = 3$, the preprocessing time increases to 38 seconds, an amount of time that is only recovered when searching for paths of length at least 19 (see Figure 5.7).

Using lower bounds is only effective once we have already found as many paths as we are looking for. Therefore, it is important to quickly find some low-weight paths early in the process. We achieve this "preheating" by prepending a number of trials with a thinned-out graph, that is, for some $0 < t < 1$, we consider a graph that contains only the tm lightest edges of the input graph. Trials for a certain value of t are repeated with different random colorings until the lower bound does not improve any more. By default, t is increased in steps of $1/10$; should we run out of memory, this step size is halved. This allows to successfully complete trials in the thinned out graphs, making trials feasible on the original graphs by providing them with powerful bounds for pruning.

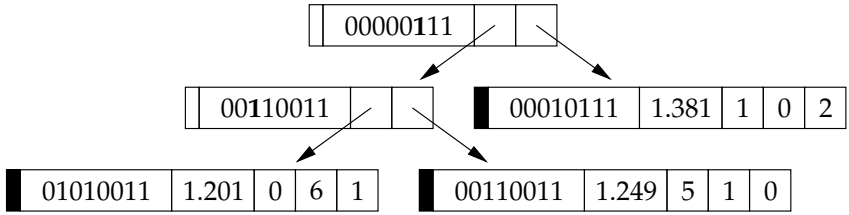


Figure 5.8: Representation of color sets at some node u with color 4. Inner nodes contain the marker bit (leftmost box), the common suffix (branch bit in bold), and two pointers to children. Leaf nodes contain the marker bit, the color set, the weight, and the vertices of the path.

5.3.3 Data structures

For each vertex, we need a data structure that maps color sets (keys) to a partial colorful path together with its weight (value). We represent a color set as a bit string of fixed length, that is, bit i is set to 1 iff color i is an element of the color set. This allows to use a Patricia tree, that is, a compact representation of a radix tree [Cormen et al. 2001] where any node which is an only child is merged with its parent (see Figure 5.8 for an example). In radix trees, only the leaves carry key/value pairs, while inner nodes serve for navigation.

Inner nodes of the tree contain a color set. The highest 1-bit of this color set is the *branch bit*. For all leaf nodes below an inner node, the bits below the branch bits are equal to the corresponding bits in this inner node. The left subtree contains color sets where the branch bit is 0, and the right subtree those where it is 1. We additionally need a marker bit to distinguish inner nodes and leaves. A leaf stores the complete color set, the weight of the corresponding partial path, and the vertices in the order of occurrence on the path (except for the last one, which is redundant).

The height of the tree is naturally limited by the number of colors, so no balancing is needed. This data structure allows for very quick insertions and iterations with a moderate memory overhead of, e.g., 12 bytes per color set on a 32-bit system. Memory allocation time and space overhead is minimized by using a memory pool.

The data structure has the additional advantage that it is possible to quickly skip over color sets containing a certain color by noting that the corresponding bit is set in the suffix at some inner node.

The storage of the vertices of the partial paths accounts for the bulk of the memory requirement, because $k\lceil \log n \rceil$ bits per stored path are required. We can save memory here by noting that it suffices to store only the order in which

the colors appear on a path: after completing a color set at some vertex u, the path can be recovered by running a shortest path algorithm (e.g., Dijkstra's algorithm [Cormen et al. 2001]) for the source vertex u while allowing it to only travel edges that match the color order. This reduces the memory cost per entry to $k\lceil \log k \rceil$ bits, which, for our application, amounts to a saving factor of about 2–4. Because of the resulting increase in computer cache effectiveness, this usually also leads to a speedup except when either short path lengths are used (where memory is not an issue anyway) or when many solution paths are found and have to be reconstructed.

5.3.4 Improvements for Pathway Query

If we wish to exploit the heuristic cutoffs and the resulting sparseness of the dynamic programming table when solving an instance of Pathway Query, we cannot use recurrence (5.16) from Section 5.2.2 directly; rather, the entries must be built up inductively. For this purpose, the dimension of i is represented implicitly by working layerwise from $i = 1$ to l and accessing only the previous layer. The dimensions of v and θ are represented explicitly as an array, while the values of S are covered by one Patricia tree (see Section 5.3.3) per combination of v and θ. The calculation of layer $i + 1$ from layer i starts by expand each entry $W(v, i, \theta, S)$ with weight w_i in layer i by possible matchings or deletions of a single vertex:

$$\text{if } \{u, v\} \in E \wedge \text{color}(u) \notin S : \tag{5.19}$$

$$T[u, i + 1, \theta, S \cup \{\text{color}(u)\}] \leftarrow w_i + w(u, v) + h(q_{i+1}, u) \tag{5.20}$$

$$\text{if } \theta < N_{\text{del}} : \tag{5.21}$$

$$T[v, i + 1, \theta + 1, S] \leftarrow w_i. \tag{5.22}$$

The update is skipped if an entry with lower weight is already present. Since insertions do not increment i, we then have to update the table for layer $i + 1$ by entries with an arbitrary additional number of insertions. Each entry $W(v, i + 1, \theta, S)$ with weight w_{i+1} (including those generated in this process) is expanded:

$$\text{if } \{u, v\} \in E \wedge \text{color}(u) \notin S \wedge |S| - (i + 1) < N_{\text{ins}} : \tag{5.23}$$

$$T[u, i + 1, \theta, S \cup \{\text{color}(u)\}] \leftarrow w_{i+1} + w(u, v). \tag{5.24}$$

Fortunately, the Patricia tree structure allows to do the insertion updates in a straightforward way: a single in-order walk of the tree will do, since any newly inserted entry contains one more color and will therefore be encountered later in the walk.

Table 5.2: Basic properties of the network instances YEAST [Scott et al. 2006] and DROSOPHILA [Giot et al. 2003])

	n	m	clustering coeff.	avg. degree	max. degree
YEAST	4 389	14 319	0.067	6.5	237
DROSOPHILA	7 009	20 440	0.030	5.8	175

For initialization, all possibilities to delete $0, \ldots, N_{\text{del}}$ query vertices and then match the next with $v \in V$ have to be entered into the table. Alignments starting with insertions are not considered, since they cannot be optimal.

A deletion is not allowed after an insertion, since alignments that only differ in the order of deletions and insertions between two actual matches will have the same score, and are not reasonable to differentiate for our application.

To use the heuristic lower bounds that we use for MINIMUM-WEIGHT PATH (Section 5.3.2) also for PATHWAY QUERY, these have to be slightly adapted: First, possible deletions have to be taken into account when considering the minimum additional edge that must be incurred. Second, we improve the heuristic by adding the minimum match weight of all query vertices that are yet to be matched (also considering possible deletions).

5.4 Implementation and experiments

Method and Results. We have implemented the color-coding technique with the improvements described in Section 5.3. The source code of the program is available under the GNU public license (GPL) from http://theinf1.informatik.uni-jena.de/colorcoding/; it is written in the C++ programming language and consists of approximately 1500 lines of code. The testing machine is an AMD 3400+ with 2.4 GHz, 512 KB cache, and 1 GB main memory running under the Debian GNU/Linux 3.1 operating system. The program was compiled with the GNU g++ 4.2 compiler using the options "-O3 -march=athlon".

5.4.1 Minimum-Weight Path

The real-world network instances used for speed measurements were the *Saccharomyces cerevisiae* interaction network used by Scott et al. [2006] and the *Drosophila melanogaster* interaction network described by Giot et al. [2003]. Due to the difficulty of obtaining reliable interaction probabilities, other networks that are currently available are much smaller and thus not suited as benchmarks. Some properties of these networks, which we will refer to as YEAST

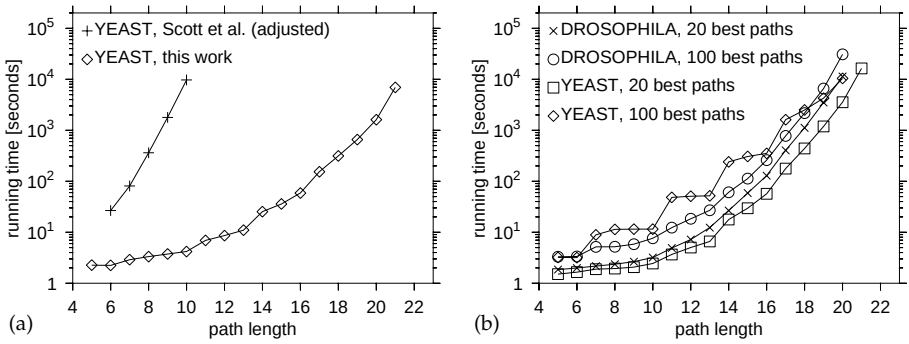


Figure 5.9: Running times for protein interaction networks

and DROSOPHILA, are summarized in Table 5.2. The *clustering coefficient* is the probability that $\{u, v\} \in E$ for $u, v, x \in V$ with $\{u, x\} \in E$ and $\{x, v\} \in E$. The clustering coefficient measures how much a graph resembles a *small-world network*. Small-world networks are networks frequently encountered in a variety of real-world instances such as social networks or power grids, which are characterized by some common properties such as the existence of large dense subgraphs, vertices with high degree, and short connections between pairs of vertices [Watts and Strogatz 1998]. A graph is considered to be a small-world network if its clustering coefficient is substantially below the clustering coefficient of a random network with the same number of vertices and edges, which is the case for both YEAST and DROSOPHILA.

To explore the sensitivity of the running time to various graph parameters (namely, the number of vertices, the clustering coefficient, the degree distribution, and the distribution of edge weights), the implementation was also run on a testbed of random graph instances that were generated with the algorithm described by Volz [2004]. The results of all experiments and details as to the experimental setting are given in Figures 5.9 and 5.10. Scott et al. [2006] obtained their running times on a 3.0 GHz Intel Xeon processor with 4 GB main memory (but only using one CPU); therefore, in Figure 5.9, we divided the running times they reported by 1.2, based on AMD's claim that the speed of an AMD 3400+ is equivalent to an Intel CPU with 3.4 GHz.

Only a few of the most difficult instances hit the predefined 768 MB memory limit and required additional preheating cycles.

Discussion. Figure 5.9a shows running times for YEAST as reported by Scott et al. [2006] and measured with our implementation. In both cases, paths

must start at a membrane protein, and end at a transcription factor. Memory requirements were, e. g., 3 MB for $k = 10$ and 242 MB for $k = 21$.

Compared to the running times from Scott et al. [2006], our implementation is faster by a factor of 10 to 2 000 on YEAST (see Figure 5.9a). Scott et al. [2006] discuss findings for paths up to a length of 10, which they were able to find in about three hours. These can be found within seconds by our implementation, allowing for interactive queries and displays (see Section 5.5). The range of feasible path lengths is more than doubled.

Figure 5.9b shows a comparison of the running times of our implementation when applied to YEAST and DROSOPHILA for various path lengths, seeking after either 20 or 100 minimum-weight paths that mutually differ in at least 30% of their vertices. There were no restrictions as to the sets of start and end vertices. The figure shows that the running times for YEAST and DROSOPHILA are roughly equal. The only exception is the search for the best 100 paths within YEAST, which not only takes unexpectedly long but also displays step-like structures. Most likely, these two phenomena can be attributed to the fact that certain path lengths allow for much fewer well-scoring paths than others in YEAST, causing the lower-bound heuristic (Section 5.3.2) to be less effective. Figure 5.9b also demonstrates that a major factor in the running time is the number of paths that is sought after. This is because a larger number of paths worsens the lower bound of the heuristic which cannot cut off as many partial solutions, and maintaining the list of paths and checking the "at least 30% of vertices must differ" criterion becomes more involved.

Figure 5.10 shows the average running time for our color-coding implementation on random networks, seeking after 20 minimum-weight paths. Unless a parameter is the variable of a measurement, the following default values are used (we have empirically found them to result in networks that are quite similar to YEAST): 4 000 vertices; degree distribution is a power law with exponential cutoff, that is, the fraction p_k of vertices with degree k satisfies $p_k \sim k^{\alpha} \cdot e^{-k/1.3} \cdot e^{-45/k}$; the default value for α is -1.6; edge weights are distributed as in YEAST; the clustering coefficient is 0.1. The data shown reports the average running time over five runs each. Figures 5.10a–c show the dependency on the number of vertices, the clustering coefficient, and the parameter α of the power law distribution, respectively. Figure 5.10d shows the dependency on the distribution of edge weights for three different distributions: A uniform $[0, 1]$-distribution, the distribution of YEAST, and the distribution of YEAST under consideration of vertex degree.

Figures 5.10a, 5.10b, and 5.10d show that the running time of the color-coding algorithm appears to be insensitive to the size of the graph (increasing linearly with increasing graph size) as well as the clustering coefficient and the distribution of edge weights. The somewhat unexpectedly high running times

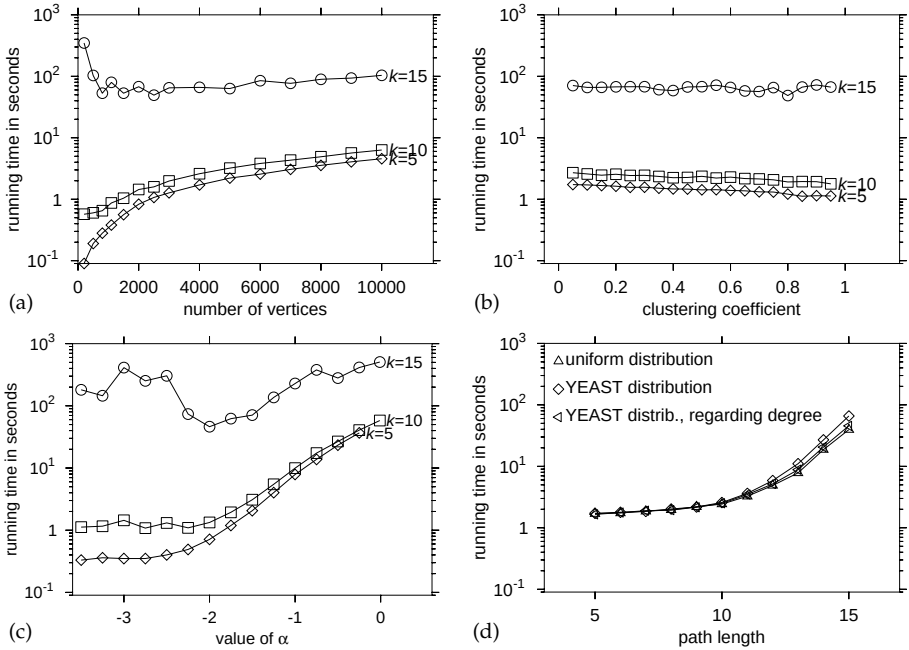


Figure 5.10: Dependency of the running time on various parameters

for graphs with less than 500 vertices in Figure 5.10a are explained by the fact that the number of length-10 and length-15 paths in these networks is very low, causing the heuristic lower bounds to be rather ineffective (this also explains why the effect is worse for $k = 15$ than it is for $k = 10$).

Figure 5.10c shows that the algorithm is generally faster when the vertex degrees are unevenly distributed. This comes as no surprise, because for low-degree vertices, fewer color sets have to be maintained in general and the heuristic lower bounds are often better. For $k = 15$, two points in the curve require further explanation: First, the drop-off in running time for $\alpha < -3$ is explained by the random graph "disintegrating" into small components. Second, the increased running time for $-3 \leqslant \alpha \leqslant -2$ is most likely due to a decrease in the total number of length-15 paths as compared to larger values of α.

5.4.2 Pathway Query

Method and Results. To evaluate the performance of our improved color-coding for Pathway Query, we conducted experiments similar to those of

Table 5.3: Running times for path queries in the *D. melanogaster* network. The rightmost column gives the percentage of query paths for which a matching path in the *D. melanogaster* network could be found.

Path length	Avg. Time [s]	Max. Time [s]	Successful Queries
4	2.24	2.57	98%
5	2.33	3.61	93%
6	3.00	23.02	81%
7	4.52	93.32	52%
8	7.49	225.61	31%
9	11.38	245.78	13%

Shlomi et al. [2006]: The data basis are their protein interaction networks of *S. cerevisiae* and *D. melanogaster* as well as a matrix of protein similarity scores described by Shlomi et al. [2006]. To obtain query paths of various lengths $l = 4, \ldots, 9$, we determined the set of the 100 minimum-weight paths of each length in the *S. cerevisiae* network, using the constraint that no two paths of the same length are allowed to overlap by more than 20 % of their vertices. We then determined the best match for each of these query paths in the *D. melanogaster* network, allowing up to 3 insertions and up to 3 deletions. Table 5.3 shows the obtained running times for these queries.

Discussion. Shlomi et al. [2006] were able to answer path queries for paths of length 7 on average within 8 minutes on a Pentium 4 with 1.7 GHz. On average, our implementation solves these within a few seconds and is able to answer queries even for length $l = 9$ within reasonable time (longer queries are of doubtful relevance on this data set, since matches are rarely found). In general, we found the algorithm to be substantially slowed down if the network contains no path that matches the query, which explains the large deviations between the average and maximum required time as the query path length increases.

5.5 Graphical user interface

The improvements of Section 5.3 have sped up color-coding to a point where basically all relevant queries for signaling pathways in protein interaction networks [Scott et al. 2006] can be done within seconds. This makes it attractive to have a graphical user interface FASPAD (Fast Signaling Pathway Detection) based on our color-coding implementation, which by visualizing pathway candidates

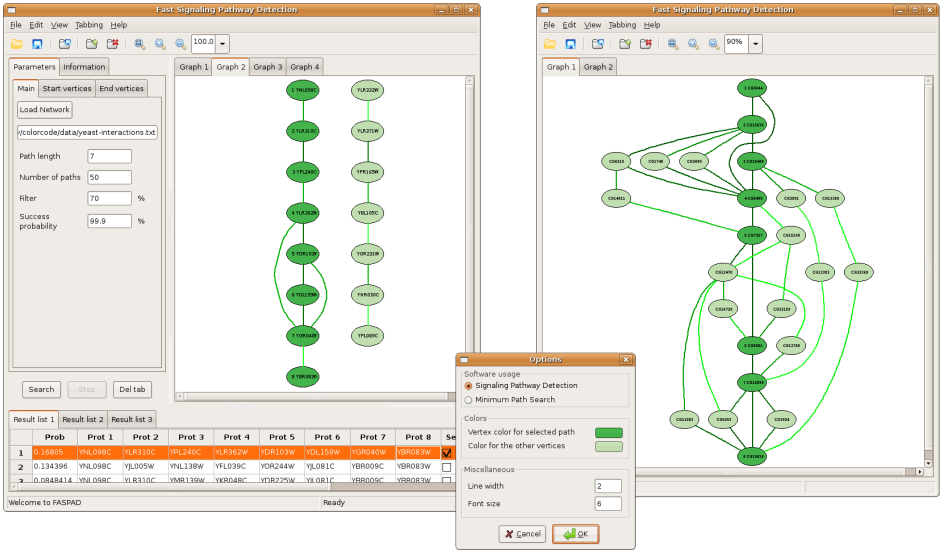


Figure 5.11: Two of the pathway candidates found by FASPAD in the yeast network by Scott et al. [2006] are displayed on the left: they correspond to the cell wall integrity pathway and the filamentous growth pathway known from literature. The right window shows a path in the Drosophila network (dark nodes) with its context. Also shown is the options dialog.

allows the user to rapidly evaluate them.

FASPAD is available as free software under the GPL license at http://theinf1.informatik.uni-jena.de/faspad/. The graphical user interface is based on the free wxWidgets library, which makes it portable to several operating systems; the web page provide Linux and Windows binaries.

The workflow with FASPAD is divided into two main phases: First, a set of parameters must be specified that describes the paths the user is interested in. Second, following the actual calculation of pathway candidates, the user can graphically display them and examine their interactions with each other. The results can be saved both graphically and in text format. Both workflow phases are detailed in the following subsections, the interface is exemplified in Figure 5.11.

The use of FASPAD is not limited to protein interaction networks; optionally, it can search paths that minimize the sum of edge weights in arbitrary networks.

5.5.1 Search parameters

The user first specifies a file that contains the actual protein interaction network. The file format that is used by FASPAD is a simple text format, which facilitates the conversion of various data sources to it. Various parameters can be set for a candidate search:

Path length. FASPAD supports path lengths up to 31 proteins. (Considering that signaling pathways quite rarely consist of more than 15 proteins, this should not pose any restrictions for practical use.) Depending on the size and structure of the network, starting at 15–20 the search might become rather slow.

Number of paths. Typically, one is not only interested in a single "best" path, but rather wants a set of "good" paths to further examine them. The size of this set can be specified with a parameter. Requesting more than ~100 pathway candidates can slow down the search substantialy.

Filter. Looking at a number of paths with the best scores, one will typically find many small variations of the top scoring path. These are not interesting for manual examination. Therefore, it is possible to request that paths must not have more than a certain percentage of their proteins in common. Using a very low value here while looking for many paths can slow down the search substantially.

Success probability. The search algorithm that FASPAD uses is randomized, that is, the results are optimal only with a certain probability. This probability can be specified by the user in order to balance accuracy with running time. (Increasing the probability slows down the algorithm by a logarithmic factor.) The default probability is 99.9%; higher values are probably not useful, but lower values can give moderate running time improvements.

Start and end nodes. Often, it is interesting to restrict the search to candidates that start and end with certain groups of proteins, e. g., candidates that start with a membrane protein and end with a transcription factor. Any such restriction can be specified in FASPAD.

With most parameter choices that are relevant in practice, the actual calculation of pathway candidates takes only a few seconds. The candidates that are found are displayed in table form.

5.5.2 Graphical display

In the most simple form, a single pathway candidate is displayed graphically. The user can examine the nodes and the edges and their probability. The nodes are arranged automatically using the *dot* algorithm provided by the *graphviz* graph drawing package [Gansner and North 2000]. The arrangement can be tweaked in a drag&drop manner. It is also possible to zoom in to examine details of a complicated network.

Often, one would like to examine whether there are small variations of a path that "make more sense", even if their score is not as good, for example due to measuring errors. For this, it is possible to display also some surrounding context of the path. For example, one can display all proteins that interact with the displayed proteins with a probability higher than 70%, or display possible shortcuts within the path.

In order to compare several pathway candidates found by FASPAD, it is possible to select more than one path for display at the same time. This can be combined with the context display option.

To make it possible to generate different views on the same data set, or to compare results for different parameter settings or even different data sets, several "tabs" can be used, each displaying a combination of paths with a certain layout. The result list can be saved and restored later.

Finally, the generated paths can be exported in a simple text format. The path display can be exported as PostScript vector graphics, as .png graphics file, or in the "dot" format, which is a network layout description format used by graphviz.

5.6 Outlook

In this chapter, we have shown how color-coding can be used to develop viable tools for detecting signaling pathways in protein interaction networks. Despite the hardness of this problem, our tool FASPAD can handle most practical tasks without longer delays. This underlines the utility of color-coding as a tool for designing FPT algorithms for subgraph isomorphism problems.

We point out some concrete open questions for further research.

- Dost et al. [2007] use a heuristic method to reduce the number of color-coding trials in querying problems. For this, they exploit that a query vertex can typically only map to a few graph vertices. Can this improvement be combined with the reduction in trials by using more colors (Section 5.3.1)?
- Using color-coding, the k-CARDINALITY TREE problem, which asks for a subtree of minimum weight with k edges in a graph, can be solved

in $8.16^k \cdot n^{O(1)}$ time [Scott et al. 2006]. The dynamic programming for a trial takes $3^k \cdot n^{O(1)}$ time. Björklund et al. [2007] show how to improve this to $2^k \cdot n^{O(1)}$ time, resulting in the same $5.44^k \cdot n^{O(1)}$ running time as the Minimum-Weight Path algorithm. The problem with the latter approach is that the dynamic programming table uses numbers with $n \log M$ bits, where M is the maximum weight; thus, for example for the yeast network [Scott et al. 2006], a single table entry would use about 16 KB memory. Further, the table cannot be made sparse easily. Still, if it was possible to reduce the memory requirement, this might be a viable approach to k-Cardinality Tree, which has many applications [Bruglieri et al. 2006].

- The divide-and-color method [Kneis et al. 2006, Chen et al. 2007b] with $4^k \cdot n^{O(1)}$ for Minimum-Weight Path provides a better worst-case running time than that of $4.32^k \cdot n^{O(1)}$ provided by Theorem 5.3. However, as mentioned in Section 2.4, independent of the actual graph structure, we actually need $\Omega(4^k) \cdot n^{O(1)}$ operations, whereas for color-coding, we typically calculate only a small fraction of the dynamic programming table. Can this disadvantage of divide-and-color be mitigated such that it becomes a viable alternative to color-coding?

- In principle, color-coding needs not be limited to graph problems, but should be applicable to any problem where a small substructure is sought. An example might be Longest Common Substring or related problems.

Chapter 6

Outlook

In this chapter, we point out some possible future research directions that are less directly connected to the problems we considered. We further give some recommendations for algorithm engineering for parameterized approaches to NP-hard problems, based on our experiences.

In Chapter 3, we have added two more sections to the success story of data reduction. The identification of useful parameters and kernelizations can be a good guideline in the design of data reductions. For example, the *extremal method* [Prieto Rodríguez 2005] has been used to find several kernelizations, and it is an interesting open task to experimentally evaluate these.

Chapter 4 and Chapter 5 demonstrate the successful application of two unconventional techniques (iterative compression and color-coding) that were designed with the goal of obtaining fixed-parameter algorithms. This clearly demonstrates the usefulness of the parameterized approach in guiding algorithm design. It is of course tempting to look for other such techniques that might have untapped practical potential. One such technique is *greedy localization* [Chen et al. 2004, Dehne et al. 2004], which seems to be mostly applicable to maximization problems that have a "packing" character (finding a large disjoint collection of objects). The idea is to greedily find an initial solution. If this solution is large enough, we are done; otherwise, the structure of the solution can be used for further search. Greedy localization has for example been used to find an $8^k \cdot n^{O(1)}$ time algorithm for Set Splitting [Dehne et al. 2004].

Further, it is an interesting endeavour to apply some of the new viewpoints and tools of algorithm engineering to FPT algorithms. This includes more realistic machine models that for example consider memory hierarchies; further, "exotic" computation models such as multi-core processing or FPGAs (field-programmable gate arrays, see Abu-Khzam et al. [2004b] for an application), or

even GPUs (graphics processing units) are promising fields.

Another important task is to open up new fields of deployment for fixed-parameter algorithmics. A particularly well-suited field is bioinformatics (see Hüffner et al. [2007b] for a survey on the use of FPT in bioinformatics). Here, a large number of new problem settings are emerging, and many of them are NP-hard; however, the instances typically have structural properties that can be exploited by parameterization. Another prospective field is that of computational social choice, which includes tasks such as voting winner determination or fair resource allocation (see e. g. Endriss and Lang [2006]); here, few results in the parameterized context are known (a pioneering work is by Christian et al. [2007]).

We will now try to give some general recommendations on how to go about applying FPT techniques to NP-hard problems encountered in the real world. In this, we follow a step-by-step program.

Identification of parameters. The first task is to identify useful parameters. Usually, there is a "natural" parameter such as the solution size. But it is also useful to consider other parameters, such as the distance from tractable instances. The choice of the parameter clearly depends on whether it will be small in the instances to be expected. At this point, it is useful to determine whether the problem is fixed-parameter tractable or W[1]-hard (see e. g. Niedermeier [2006]); a hardness result encourages to look for another parameter.

Implementation of brute-force search. The first thing to do then is to implement a brute-force search that is as simple as possible. There are several reasons for this: It gives some first impression on what solutions look like (for example, can we use their size as parameter?). Also, they are invaluable in shaking out bugs from later, more sophisticated implementations, in particular if results for random instances are systematically compared. Possibly the best way to get a simple brute-force result is an integer linear program (ILP, see e. g. Section 4.6.3); they sometimes need only a few lines when using a modelling language such as MathProg [Makhorin 2004], but are often surprisingly effective, due to the decades of engineering that went into the solvers. The second method of choice is a simple search tree (Section 1.3.1).

Implementation of data reduction. Since data reduction is valuable in combination with any other algorithmic technique such as approximation, heuristics, or fixed-parameter algorithms, and in some cases can even completely solve instances without further effort, it can be considered as essential for the treatment of NP-hard problems, and should always be the first nontrivial technique to be

developed and implemented. When combined with even a naive brute-force approach, it can often already solve instances of notable size.

Tuned search trees. After this, the easiest speedups typically come from a more carefully tuned search tree algorithm. Case distinction can help to improve provable running time bounds, although it has often been reported that a too complicated branching actually leads to a slowdown. Heuristic branching priorities can help, as well as admissible heuristic evaluation functions [Felner et al. 2004]. Further, interleaving with data reduction can lead to a speedup [Niedermeier and Rossmanith 2000].

More "exotic" techniques. When search trees are not applicable or too slow, less clear instructions can be given. The best thing to do is to look at other FPT algorithms and techniques for inspiration: Does induction help, like with iterative compression (Chapter 4)? Does randomization help, like with color-coding (Chapter 5)? In this way, possibly using some of the more exotic approaches, it might be possible to come up with a fixed-parameter algorithm. Here, one should be wary of exponential-space algorithms; they can often fill the memory within seconds and therefore become unusable in practice.

Heuristics. Some of the largest speedups we experienced in our experiments came from techniques that can only be considered heuristic in the sense that they do not improve worst-case time bounds or the kernel size. Therefore, one should not easily dismiss such heuristics. The general idea of most heuristics is to recognize early that some branch or some subcase cannot lead to an optimal solution, and to skip those. Particularly useful is it to dismiss entire classes of branches or cases without explicitly enumerating them. This kind of heuristics should always be kept in mind when implementing algorithms.

Programming language. Finally, some words about the programming language to use: we recommend to use very high-level programming languages such as Haskell or Objective Caml. The reason is that these languages allow for a very rapid development, because of powerful abstractions, and because a large class of bugs such as buffer overflows are avoided. The drawback is that execution is typically slower than that of code written in C or C++. However, this speed difference is usually a small constant factor (e. g., it is the stated goal of the Objective Caml developers that the produced code is never more than a factor of 2 slower than compiled C code). In contrast, the speedups gained from more effective algorithms for NP-hard problems can be tremendous; we regularly reported speedups of several orders of magnitude in our experiments. Using

a high-level programming language will allow to code more improvements within the same time, and will thus likely result in faster algorithms. Further, after an algorithm has stabilized, one can replace "inner loops" by optimized C routines, as we did for the iterative compression routine for Balanced Subgraph (Section 3.2.4) and regain pretty much all of the speed.

Bibliography

Abdullah, Ahsan. On graph bipartization. In *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems (ISCAS '92)*, volume 4, pages 1847–1850. 1992. Cited on p. 115.

Abu-Khzam, Faisal N. Kernelization algorithms for d-hitting set problems. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS '07)*, volume 4619 of *LNCS*, pages 434–445. Springer, 2007. Cited on pp. 65, 72, 78, 81, 85, and 115.

Abu-Khzam, Faisal N., Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christopher T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX '04)*, pages 62–69. SIAM, 2004a. Cited on pp. 31, 45, and 68.

Abu-Khzam, Faisal N., Michael R. Fellows, Michael A. Langston, and W. Henry Suters. Crown structures for vertex cover kernelization. *Theory of Computing Systems*, 41(3):411–430, 2007. Cited on pp. 31 and 45.

Abu-Khzam, Faisal N. and Henning Fernau. Kernels: Annotated, proper and induced. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 264–275. Springer, 2006. Cited on pp. 73 and 78.

Abu-Khzam, Faisal N., Michael A. Langston, and Pushkar Shanbhag. Scalable parallel algorithms for difficult combinatorial problems: A case study in optimization. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '04)*, pages 649–654. IASTED/ACTA Press, 2004b. Cited on p. 145.

Abu-Khzam, Faisal N., Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable parallel algorithms for FPT problems. *Algorithmica*, 45(3):269–284, 2006. Cited on p. 4.

Agarwal, Amit, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $O(\sqrt{\log n})$ approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems. In *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC '05)*, pages 573–581. ACM, 2005. Cited on pp. 18 and 93.

Agarwal, Pankaj K., Noga Alon, Boris Aronov, and Subhash Suri. Can visibility graphs be represented compactly? *Discrete and Computational Geometry*, 12(1):347–365, 1994. Cited on p. 16.

Ahuja, Ravindra K., Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. Cited on p. 93.

Ailon, Nir, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: ranking and clustering. In *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC '05)*, pages 684–693. ACM, 2005a. Cited on pp. 80 and 81.

Ailon, Nir, Moses Charikar, and Alantha Newman. Proofs of conjectures in "Aggregating inconsistent information: Ranking and clustering". Technical Report TR-719-05, Department of Computer Science, Princeton University, 2005b. Cited on p. 71.

Alber, Jochen, Nadja Betzler, and Rolf Niedermeier. Experiments on data reduction for optimal domination in networks. *Annals of Operations Research*, 146(1):105–117, 2006. Cited on p. 31.

Alber, Jochen, Frederic Dorn, and Rolf Niedermeier. Experimental evaluation of a tree decomposition based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, 145(2):219–231, 2005. Cited on p. 7.

Alber, Jochen, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *Journal of the ACM*, 51(3):363–384, 2004. Cited on pp. 31 and 34.

Alber, Jochen and Rolf Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN '02)*, volume 2286 of *LNCS*, pages 613–628. Springer, 2002. Cited on p. 7.

Alon, Noga. Ranking tournaments. *SIAM Journal on Discrete Mathematics*, 20(1):137–142, 2006. Cited on p. 81.

Alon, Noga, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995. Cited on pp. 27, 117, 118, 123, 125, and 126.

Amilhastre, Jérôme, Marie-Catherine Vilarem, and Philippe Janssen. Complexity of minimum biclique cover and minimum biclique decomposition for bipartite domino-free graphs. *Discrete Applied Mathematics*, 86(2–3):125–144, 1998. Cited on p. 44.

Antal, Tibor, Paul L. Krapivsky, and Sidney Redner. Social balance on networks: The dynamics of friendship and enmity. *Physica D: Nonlinear Phenomena*, 224(1–2):130–136, 2006. Cited on p. 23.

Asano, Takao and Tomio Hirata. Edge-deletion and edge-contraction problems. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC '82)*, pages 245–254. ACM, 1982. Cited on p. 14.

Ausiello, Giorgio, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999. Cited on pp. 1 and 15.

Avidor, Adi and Michael Langberg. The multi-multiway cut problem. *Theoretical Computer Science*, 377(1–3):35–42, 2007. Cited on pp. 18 and 93.

Bader, Joel S., Amitabha Chaudhuri, Jonathan M. Rothberg, and John Chant. Gaining confidence in high-throughput protein interaction networks. *Nature Biotechnology*, 22(1):78–85, 2003. Cited on p. 26.

Bang-Jensen, Jørgen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2002. Cited on p. 11.

Bansal, Nikhil, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1–3):89–113, 2004. Cited on p. 71.

Barahona, Francisco. On the complexity of max cut. Technical Report 186, IMAG, Université Joseph Fourier, Grenoble, France, 1980. Cited on pp. 19 and 59.

Barahona, Francisco. On the computational complexity of Ising spin glass models. *Journal of Physics A: Mathematics and General*, 15(3241–3253), 1982. Cited on p. 23.

Barahona, Francisco, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, 1988. Cited on p. 20.

Barahona, Francisco, Martin Grötschel, and Ali Ridha Mahjoub. Facets of the bipartite subgraph polytope. *Mathematics of Operations Research*, 10:340–358, 1985. Cited on p. 20.

Barahona, Francisco and Ali Ridha Mahjoub. Facets of the balanced (acyclic) induced subgraph polytope. *Mathematical Programming*, 45(1–3):21–33, 1989. Cited on p. 24.

Barvinok, Alexander I. and Kevin Woods. Short rational generating functions for lattice point problems. *Journal of the American Mathematical Society*, 16(4):957–979, 2003. Cited on p. 54.

Bast, Holger, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007. Cited on p. 9.

Behrisch, Michael and Anusch Taraz. Efficiently covering complex networks with cliques of similar vertices. *Theoretical Computer Science*, 355(1):37–47, 2006. Cited on pp. 15 and 42.

Bellman, Richard. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962. Cited on p. 27.

Betzler, Nadja. *Steiner Tree Problems in the Analysis of Biological Networks*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2006. Cited on pp. 5 and 118.

Bixby, Robert E. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002. Cited on p. 29.

Björklund, Andreas, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets Möbius: fast subset convolution. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC '07)*, pages 67–74. ACM, 2007. Cited on pp. 5, 119, and 144.

Böcker, Sebastian, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. PEACE: Parameterized and exact algorithms for cluster editing, 2007. Manuscript, Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena. Cited on pp. 4, 5, 71, and 72.

Bodlaender, Hans L. On linear time minor tests with depth-first search. *Journal of Algorithms*, 14(1):1–23, 1993. Cited on p. 27.

Bodlaender, Hans L. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. Cited on p. 7.

Bodlaender, Hans L. A cubic kernel for feedback vertex set. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Science (STACS '07)*, volume 4393 of *LNCS*, pages 320–331. Springer, 2007. Cited on p. 115.

Bodlaender, Hans L., Leizhen Cai, Jianer Chen, Michael R. Fellows, Jan Arne Telle, and Dániel Marx. Open problems in parameterized and exact computation — IWPEC 2006. Technical Report UU-CS-2006-052, Department of Information and Computing

Sciences, Utrecht University, 2006. Cited on p. 116.

Bodlaender, Hans L., Celina M. Herrera de Figueiredo, Marisa Gutierrez, Ton Kloks, and Rolf Niedermeier. Simple max-cut for split-indifference graphs and graphs with few P_4's. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA '05)*, volume 3503 of *LNCS*, pages 87–99. Springer, 2005. Cited on p. 19.

Bodlaender, Hans L. and Klaus Jansen. On the complexity of the maximum cut problem. *Nordic Journal of Computing*, 7(1):14–31, 2000. Cited on p. 19.

Bodlaender, Hans L. and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 2007. To appear. Cited on p. 7.

Borndörfer, Ralf, Martin Grötschel, and Marc E. Pfetsch. A column-generation approach to line planning in public transport. *Transportation Science*, 41(1):123–132, 2007. Cited on p. 119.

Boros, Endre and Peter L. Hammer. The max-cut problem and quadratic 0–1 optimization; polyhedral aspects, relaxations and bounds. *Annals of Operations Research*, 33(3):151–180, 1991. Cited on p. 24.

Brandstädt, Andreas, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*, volume 3 of *SIAM Monographs on Discrete Mathematics and Applications*. SIAM, 1999. Cited on p. 11.

Bron, Coen and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973. Cited on p. 38.

Brosemann, Matthias, Jochen Alber, Falk Hüffner, and Rolf Niedermeier. Matrix robustness, with an application to power system observability. In *Proceedings of the 2nd Algorithms and Complexity in Durham Workshop (ACiD '06)*, volume 7 of *Texts in Algorithmics*, pages 37–48. College Publications, 2006. Cited on pp. xi and 170.

Bruglieri, Maurizio, Matthias Ehrgott, Horst W. Hamacher, and Francesco Maffioli. An annotated bibliography of combinatorial optimization problems with fixed cardinality constraints. *Discrete Applied Mathematics*, 154(9):1344–1357, 2006. Cited on p. 144.

Burrage, Kevin, Vladimir Estivill-Castro, Michael R. Fellows, Michael A. Langston, Shev Mac, and Frances A. Rosamond. The undirected feedback vertex set problem has a poly(k) kernel. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 192–202. Springer, 2006. Cited on p. 115.

Burzyn, Pablo, Flavia Bonomo, and Guillermo Durán. NP-completeness results for edge modification problems. *Discrete Applied Mathematics*, 154(13):1824–1844, 2006. Cited on p. 14.

Cai, Leizhen. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, 1996. Cited on p. 14.

Cai, Leizhen, Siu Man Chan, and Siu On Chan. Random separation: A new method for solving fixed-cardinality optimization problems. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 239–250. Springer, 2006. Cited on p. 118.

Cai, Liming, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84(1):119–138, 1997. Cited

on p. 31.

Cai, Liming, Xiuzhen Huang, Chunmei Liu, Frances A. Rosamond, and Yinglei Song. Parameterized complexity and biopolymer sequence comparison. *The Computer Journal*, 2007. To appear. Cited on p. 7.

Cai, Liming and David W. Juedes. On the existence of subexponential parameterized algorithms. *Journal of Computer and System Sciences*, 64(4):789–807, 2003. Cited on p. 28.

Cai, Mao-Cheng, Xiaotie Deng, and Wenan Zang. An approximation algorithm for feedback vertex sets in tournaments. *SIAM Journal on Computing*, 30(6):1993–2007, 2001. Cited on pp. 79 and 80.

Cai, Mao-Cheng, Xiaotie Deng, and Wenan Zang. A min-max theorem on feedback vertex sets. *Mathematics of Operations Research*, 27(2):361–371, 2002. Cited on p. 87.

Cappanera, Paola and Maria Grazia Scutellà. Balanced paths in telecommunication networks: some computational results. In *Proceedings of the 3rd International Network Optimization Conference (INOC '07)*. 2007. Cited on p. 119.

Chang, Maw-Shang and Haiko Müller. On the tree-degree of graphs. In *Proceedings of the 27th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '01)*, volume 2204 of *LNCS*, pages 44–54. Springer, 2001. Cited on p. 15.

Chang, Maw-Shang and Fu-Hsing Wang. Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. *Information Processing Letters*, 43(6):293–295, 1992. Cited on p. 86.

Charbit, Pierre, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is NP-hard for tournaments. *Combinatorics, Probability and Computing*, 16:1–4, 2007. Cited on p. 81.

Charikar, Moses, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. *Journal of Computer and System Sciences*, 71(3):360–383, 2005. Cited on p. 72.

Charon, Irène and Olivier Hudry. A survey on the linear ordering problem for weighted or unweighted tournaments. *4OR: A Quarterly Journal of Operations Research*, 5(1):5–60, 2007. Cited on p. 80.

Cheetham, James, Frank K. H. A. Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J. Taillon. Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, 67(4):691–706, 2003. Cited on p. 4.

Chen, Jianer, Fedor V. Fomin, Yang Liu, Songjian Lu, and Yngve Villanger. Improved algorithms for the feedback vertex set problems. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS '07)*, volume 4619 of *LNCS*, pages 422–433. Springer, 2007a. Cited on pp. 62, 63, 68, 70, 77, and 100.

Chen, Jianer, Donald K. Friesen, Weijia Jia, and Iyad A. Kanj. Using nondeterminism to design efficient deterministic algorithms. *Algorithmica*, 40(2):83–97, 2004. Cited on p. 145.

Chen, Jianer, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001. Cited on p. 45.

Chen, Jianer, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science (MFCS '06)*, volume 4162 of *LNCS*, pages 238–249. Springer, 2006. Cited on pp. 3, 4, 63, and 67.

Chen, Jianer, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. In *Proceedings of the 40th ACM Symposium on Theory of Computing (STOC '08)*. ACM, 2008. Cited on pp. 61, 62, 69, 79, and 100.

Chen, Jianer, Songjian Lu, Sing-Hoi Sze, and Fenghui Zhang. Improved algorithms for path, matching, and packing problems. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*, pages 298–307. ACM-SIAM, 2007b. Cited on pp. 27, 118, 130, and 144.

Chiang, Charles, Andrew B. Kahng, Subarna Sinha, Xu Xu, and Alexander Z. Zelikovsky. Fast and efficient bright-field AAPSM conflict detection and correction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(1):115–126, 2007. Cited on pp. 24 and 59.

Choi, Hyeong-Ah, Kazuo Nakajima, and Chong S. Rim. Graph bipartization and via minimization. *SIAM Journal on Discrete Mathematics*, 2(1):38–47, 1989. Cited on pp. 24 and 25.

Chor, Benny, Michael R. Fellows, and David W. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In *Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '04)*, volume 3353 of *LNCS*, pages 257–269. Springer, 2004. Cited on p. 45.

Christian, Robin, Mike Fellows, Frances A. Rosamond, and Arkadii Slinko. On complexity of lobbying in multiple referenda. *Review of Economic Design*, 11(3):217–224, 2007. Cited on p. 146.

Conitzer, Vincent. Computing Slater rankings using similarities among candidates. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI '06)*. AAAI Press, 2006. Cited on p. 81.

Coppersmith, Don and Uzi Vishkin. Solving NP-hard problems in 'almost trees': Vertex cover. *Discrete Applied Mathematics*, 10(1):27–45, 1985. Cited on p. 6.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. Cited on pp. 5, 76, 93, 107, 109, 134, and 135.

Cornaz, Denis and Ali Ridha Mahjoub. The maximum induced bipartite subgraph problem with edge weights. *SIAM Journal on Discrete Mathematics*, 21(3):662–675, 2007. Cited on p. 24.

Damaschke, Peter. On the fixed-parameter enumerability of cluster editing. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG '05)*, volume 3787 of *LNCS*, pages 283–294. Springer, 2005. Cited on p. 79.

DasGupta, Bhaskar, German Andres Enciso, Eduardo D. Sontag, and Yi Zhang. Algorithmic and complexity results for decompositions of biological networks into monotone subsystems. *Biosystems*, 90(1):161–178, 2007. Cited on pp. 22, 23, 24, 45, 54, 55, 59, 98, and 100.

Davis, Martin, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. Cited on p. 4.

Dehne, Frank K. H. A., Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Kim Stevens. An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Theory of Computing Systems*, 41(3):479–492, 2007. Cited on pp. 61, 62, 68,

and 82.

Dehne, Frank K. H. A., Michael R. Fellows, Frances A. Rosamond, and Peter Shaw. Greedy localization, iterative compression, and modeled crown reductions: New FPT techniques, an improved algorithm for set splitting, and a novel 2k kernelization for vertex cover. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC '04)*, volume 3162 of *LNCS*, pages 271–280. Springer, 2004. Cited on p. 145.

Dehne, Frank K. H. A., Michael A. Langston, Xuemei Luo, Sylvain Pitre, Peter Shaw, and Yun Zhang. The cluster editing problem: Implementations and experiments. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 13–24. Springer, 2006. Cited on pp. 5 and 71.

Deolalikar, Vinay, Malena R. Mesarina, John Recker, and Salil Pradhan. Perturbative time and frequency allocations for RFID reader networks. In *Proceedings of the 2006 Workshop on Emerging Directions in Embedded and Ubiquitous Computing (EUC '06)*, volume 4097 of *LNCS*, pages 392–402. Springer, 2006. Cited on p. 25.

Deshpande, Pawan, Regina Barzilay, and David R. Karger. Randomized decoding for selection-and-ordering problems. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL HLT '07)*, pages 444–451. Association for Computational Linguistics, 2007. Cited on pp. 28 and 119.

Díaz, Josep and Marcin J. Kamiński. Max-cut and max-bisection are NP-hard on unit disk graphs. *Theoretical Computer Science*, 377(1–3):271–276, 2007. Cited on p. 19.

Diestel, Reinhard. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 2005. Cited on p. 10.

Dinic, Efim A. Algorithm for solution of a problem of maximum flow in networks with power estimation (in Russian). *Doklady Akademii Nauk SSSR*, 194(4), 1970. English translation in *Soviet Mathematics Doklady*, 11:1277–1280, 1970. Cited on pp. 93, 94, 106, and 107.

Dinur, Irit and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005. Cited on p. 80.

Dolezal, Oliver, Thomas Hofmeister, and Hanno Lefmann. A comparison of approximation algorithms for the maxcut-problem. Technical Report CI-57/99, Universität Dortmund, Informatik 2, 1999. Cited on p. 20.

Dom, Michael, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Extending the tractability border for closest leaf powers. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG '05)*, volume 3787 of *LNCS*, pages 397–408. Springer, 2005. Cited on pp. xi and 170.

Dom, Michael, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Error compensation in leaf power problems. *Algorithmica*, 44(4):363–381, 2006a. Cited on pp. xi and 170.

Dom, Michael, Jiong Guo, Falk Hüffner, Rolf Niedermeier, and Anke Truß. Fixed-parameter tractability results for feedback set problems in tournaments. In *Proceedings of the 6th Italian Conference on Algorithms and Complexity (CIAC '06)*, volume 3998 of *LNCS*, pages 320–331. Springer, 2006b. Cited on pp. xii, 61, 81, 87, and 170.

Dom, Michael, Falk Hüffner, and Rolf Niedermeier. Labyrinth und Tiefensuche. In

Taschenbuch der Algorithmen. Springer, 2007. To appear. Cited on pp. xi and 170.

Došlić, Tomislav and Damir Vukičević. Computing the bipartite edge frustration of fullerene graphs. *Discrete Applied Mathematics*, 155(10):1294–1301, 2007. Cited on p. 20.

Dost, Banu, Tomer Shlomi, Nitin Gupta, Eytan Ruppin, Vineet Bafna, and Roded Sharan. QNet: A tool for querying protein interaction networks. In *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology (RECOMB '07)*, volume 4453 of *Lecture Notes in Bioinformatics*, pages 1–15. Springer, 2007. Cited on pp. 118, 119, and 143.

Downey, Rodney G. and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995. Cited on p. 61.

Downey, Rodney G. and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999. Cited on pp. 2, 3, 4, 36, and 60.

Downey, Rodney G., Michael R. Fellows, and Ulrike Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In Graham, Ronald L., Jan Kratochvíl, Jaroslav Nesetril, and Fred S. Roberts, editors, *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99. AMS Press, 1999. Cited on p. 64.

Dreyfus, Stuart E. and Robert A. Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1972. Cited on p. 5.

Edmonds, Jack and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972. Cited on pp. 93, 94, 106, and 107.

El-Mallah, Ehab S. and Charles J. Colbourn. The complexity of some edge deletion problems. *IEEE Transactions on Circuits and Systems*, 35(3):354–362, 1988. Cited on p. 14.

Elias, Peter, Amiel Feinstein, and Claude E. Shannon. A note on the maximum flow through a network. *IEEE Transactions on Information Theory*, 2(4):117–119, 1956. Cited on p. 94.

Endriss, Ulle and Jérôme Lang, editors. *Proceedings of the 1st International Workshop on Computational Social Choice (COMSOC '06)*. Universiteit van Amsterdam, 2006. Cited on p. 146.

Erdős, Paul, Adolph W. Goodman, and Louis Pósa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18:106–112, 1966. Cited on p. 15.

Estivill-Castro, Vladimir, Michael R. Fellows, Michael A. Langston, and Frances A. Rosamond. FPT is P-time extremal structure I. In *Proceedings of the 1st Algorithms and Complexity in Durham Workshop (ACiD '06)*, volume 4 of *Texts in Algorithmics*, pages 1–41. College Publications, 2006. Cited on p. 60.

Feder, Tomás and Rajeev Motwani. Finding large cycles in Hamiltonian graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pages 166–175. SIAM, 2005. Cited on p. 27.

Feist, Adam M., Johannes C. M. Scholten, Bernhard Ø. Palsson, Fred J. Brockman, and Trey Ideker. Modeling methanogenesis with a genome-scale metabolic reconstruction of *Methanosarcina barkeri*. *Molecular Systems Biology*, 2:2006.0004, 2006. Cited on p. 58.

Fellows, Michael R., Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Sharp tractability borderlines for finding connected motifs in vertex-colored graphs. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP '07)*, volume 4596 of *LNCS*, pages 340–351. Springer, 2007a. Cited on p. 118.

Fellows, Michael R., Christian Knauer, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Ulrike Stege, Dimitrios M. Thilikos, and Sue Whitesides. Faster fixed-parameter tractable algorithms for matching and packing problems. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA '04)*, volume 3221 of *LNCS*, pages 311–322. Springer, 2004. Extended version to appear in *Algorithmica*. Cited on p. 118.

Fellows, Michael R. and Michael A. Langston. An analogue of the myhill-nerode theorem and its use in computing finite-basis characterizations. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS '89)*, pages 520–525. IEEE, 1989. Cited on p. 60.

Fellows, Michael R., Michael A. Langston, Frances A. Rosamond, and Peter Shaw. Efficient parameterized preprocessing for cluster editing. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory (FCT '07)*, volume 4639 of *LNCS*, pages 312–321. Springer, 2007b. Cited on pp. 72 and 79.

Felner, Ariel, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 21:1–39, 2004. Cited on pp. 68 and 147.

Fernau, Henning. A top-down approach to search-trees: Improved algorithmics for 3-hitting set. Technical Report 073, Electronic Colloquium on Computational Complexity (ECCC), 2004. Cited on p. 65.

Fernau, Henning. *Parameterized Algorithmics: A Graph-Theoretic Approach*. Habilitationsschrift, Wilhelm-Schickard Institut für Informatik, Universität Tübingen, 2005. Cited on pp. 65 and 68.

Fernau, Henning. Parameterized algorithms for hitting set: The weighted case. In *Proceedings of the 6th Italian Conference on Algorithms and Complexity (CIAC '06)*, volume 3998 of *LNCS*, pages 332–343. Springer, 2006a. Cited on p. 72.

Fernau, Henning. Speeding up exact algorithms with high probability. In *Proceedings of the 5th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW '06)*, volume 25 of *Electronic Notes in Discrete Mathematics*, pages 57–59. Elsevier, 2006b. Cited on p. 115.

Festa, Paola, Panos M. Pardalos, and Mauricio G. C. Resende. Feedback set problems. In Du, Ding-Zhu and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume Supplement A, pages 209–259. Kluwer, 1999. Cited on p. 62.

Festa, Paola, Panos M. Pardalos, Mauricio G. C. Resende, and Celso C. Ribeiro. Randomized heuristics for the max-cut problem. *Optimization Methods and Software*, 17(6):1033–1058, 2002. Cited on p. 20.

Fiorini, Samuel, Nadia Hardy, Bruce Reed, and Adrian Vetta. Planar graph bipartization in linear time. In *Proceedings of the 2nd Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO '05)*, volume 19 of *Electronic Notes in Discrete Mathematics*, pages 265–271. Elsevier, 2005. Extended version to appear in *Discrete Applied Mathematics*. Cited on p. 100.

Fleischner, Herbert, Egbert Mujuni, Daniël Paulusma, and Stefan Szeider. Covering graphs with few complete bipartite subgraphs. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '07)*, volume 4855 of *LNCS*, pages 340–351. Springer, 2007. Cited on p. 44.

Flum, Jörg and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006. Cited on p. 3.

Fouilhoux, Pierre and Ali Ridha Mahjoub. Polyhedral results for the bipartite induced subgraph problem. *Discrete Applied Mathematics*, 154(15):2128–2149, 2006. Cited on pp. 25 and 110.

Fredman, Michael L. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975. Cited on p. 85.

Fredman, Michael L. and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. Cited on p. 77.

Frieze, Alan M. and Bruce Reed. Covering the edges of a random graph by cliques. *Combinatorica*, 15(4):489–497, 1995. Cited on p. 40.

Fuchs, Bernhard, Walter Kern, Daniel Mölle, Stefan Richter, Peter Rossmanith, and Xinhui Wang. Dynamic programming for minimum Steiner trees. *Theory of Computing Systems*, 41(3):493–500, 2007. Cited on p. 5.

Fulkerson, Delbert R. and Lester R. Ford, Jr. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. Cited on p. 94.

Gabow, Harold N. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000. Cited on p. 47.

Gabow, Harold N. Finding paths and cycles of superpolylogarithmic length. *SIAM Journal on Computing*, 36(6):1648–1671, 2007. Cited on p. 27.

Gabow, Harold N. and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. Cited on p. 77.

Galluccio, Anna, Martin Loebl, and Jan Vondrák. Optimization via enumeration: a new algorithm for the max cut problem. *Mathematical Programming*, 90(2):273–290, 2001. Cited on p. 19.

Ganley, Joseph L. Computing optimal rectilinear Steiner trees: A survey and experimental evaluation. *Discrete Applied Mathematics*, 90(1–3):161–171, 1999. Cited on p. 5.

Gansner, Emden R. and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000. Cited on p. 143.

Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. Cited on pp. 1, 14, 15, 36, and 64.

Garg, Naveen, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway cuts in directed and node weighted graphs. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming (ICALP '94)*, volume 820 of *LNCS*, pages 487–498. Springer, 1994. Cited on p. 24.

Giot, L., J. S. Bader, C. Brouwer, A. Chaudhuri, et al. A protein interaction map of Drosophila melanogaster. *Science*, 302(5651):1727–1736, 2003. Cited on p. 136.

Goemans, Michel X. and David P. Williamson. Improved approximation algorithms for

maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995. Cited on pp. 19 and 22.

Goemans, Michel X. and David P. Williamson. Primal-dual approximation algorithms for feedback problems in planar graphs. *Combinatorica*, 18(1):37–59, 1998. Cited on p. 24.

Gramm, Jens, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004. Cited on pp. 5, 71, 72, and 170.

Gramm, Jens, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Graph-modeled data clustering: Exact algorithms for clique generation. *Theory of Computing Systems*, 38(4):373–392, 2005. Cited on pp. 5, 71, 72, and 170.

Gramm, Jens, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction, exact, and heuristic algorithms for clique cover. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX '06)*, pages 86–94. SIAM, 2006. Extended version to appear under the title "Data reduction and exact algorithms for clique cover" in *ACM Journal of Experimental Algorithmics*. Cited on p. xi.

Gramm, Jens, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *ACM Journal of Experimental Algorithmics*, 2007a. To appear. Cited on pp. xii and 170.

Gramm, Jens, Jiong Guo, Falk Hüffner, Rolf Niedermeier, Hans-Peter Piepho, and Ramona Schmid. Algorithms for compact letter displays: Comparison and evaluation. *Computational Statistics & Data Analysis*, 52(2):725–736, 2007b. Cited on pp. xi, xii, 15, 40, 44, and 170.

Gramm, Jens, Falk Hüffner, and Rolf Niedermeier. Closest strings, primer design, and motif search. Presented at *6th Annual International Conference on Computational Molecular Biology (RECOMB '02)*, poster session, 2002. Cited on p. 170.

Greenwell, Don L., Robert L. Hemminger, and Joseph B. Klerlein. Forbidden subgraphs. In *Proceedings of the 4th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pages 389–394. 1973. Cited on p. 14.

Grötschel, Martin, Michael Jünger, and Gerhard Reinelt. Calculating exact ground states of spin glasses: A polyhedral approach. In *Heidelberg Colloquium on Glassy Dynamics*, volume 275 of *Lecture Notes in Physics*, pages 325–353. Springer, 1987. Cited on p. 20.

Grötschel, Martin and George L. Nemhauser. A polynomial algorithm for the max-cut problem on graphs without long odd cycles. *Mathematical Programming*, 29(1):28–40, 1984. Cited on p. 19.

Grötschel, Martin and William R. Pulleyblank. Weakly bipartite graphs and the max-cut problem. *Operations Research Letters*, 1(1):23–27, 1981. Cited on p. 19.

Guan, Dah-Jyh. Generalized Gray codes with applications. *Proceedings of the National Science Council, Republic of China (A)*, 22(6):841–848, 1998. Cited on p. 107.

Guenin, Bertrand. A characterization of weakly bipartite graphs. *Journal of Combinatorial Theory, Series B*, 83(1):112–168, 2001. Cited on p. 19.

Guillaume, Jean-Loup and Matthieu Latapy. Bipartite structure of all complex networks. *Information Processing Letters*, 90(5):215–221, 2004. Cited on p. 15.

Gülpinar, Nalân, Gregory Gutin, Gautam Mitra, and Alexey Zverovich. Extracting pure network submatrices in linear programs using signed graphs. *Discrete Applied*

Mathematics, 137(3):359–372, 2004. Cited on p. 25.

Guo, Jiong. *Algorithm Design Techniques for Parameterized Graph Modification Problems*. Ph.D. thesis, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2006. Cited on pp. 13, 63, 69, 77, and 93.

Guo, Jiong. A more effective linear kernelization for cluster editing. In *Proceedings of the 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE '07)*, volume 4614 of *LNCS*, pages 36–47. Springer, 2007. Cited on pp. 72 and 79.

Guo, Jiong, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Improved fixed-parameter algorithms for two feedback set problems. In *Proceedings of the 9th Workshop on Algorithms and Data Structures (WADS '05)*, volume 3503 of *LNCS*, pages 158–168. Springer, 2005. Cited on p. xii.

Guo, Jiong, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, 72(8):1386–1396, 2006. Cited on pp. xi, xii, 61, 62, 63, 68, 82, 100, and 170.

Guo, Jiong, Falk Hüffner, Erhan Kenar, Rolf Niedermeier, and Johannes Uhlmann. Complexity and exact algorithms for vertex multicut in interval and bounded treewidth graphs. *European Journal of Operational Research*, 2007a. To appear. Cited on pp. xi and 170.

Guo, Jiong, Falk Hüffner, Christian Komusiewicz, and Yong Zhang. Improved algorithms for bicluster editing. In *Proceedings of the 5th Annual Conference on Theory and Applications of Models of Computation (TAMC '08)*, volume 4978 of *LNCS*. Springer, 2008. To appear. Cited on pp. xi and 170.

Guo, Jiong, Falk Hüffner, and Hannes Moser. Feedback arc set in bipartite tournaments is NP-complete. *Information Processing Letters*, 102(2–3):62–65, 2007b. Cited on pp. xi, 87, and 170.

Guo, Jiong, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC '04)*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004. Cited on pp. xi, 3, and 170.

Guo, Jiong and Rolf Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News*, 38(1):31–45, 2007. Cited on p. 29.

Gupta, Sushmita. Feedback arc set problem in bipartite tournaments. *Information Processing Letters*, 105(5):150–154, 2008. Cited on p. 87.

Gutin, Gregory and Anders Yeo. Some parameterized problems on digraphs. *The Computer Journal*, 2007. To appear. Cited on p. 79.

Gutwenger, Carsten and Petra Mutzel. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing (GD '00)*, volume 1984 of *LNCS*, pages 77–90. Springer, 2000. Cited on p. 47.

Gyárfás, András. A simple lower bound on edge coverings by cliques. *Discrete Mathematics*, 85(1):103–104, 1990. Cited on p. 35.

Hadlock, Frank O. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4(3):221–225, 1975. Cited on p. 19.

Harary, Frank. On the notion of balance of a signed graph. *Michigan Mathematical Journal*, 2(2):143–146, 1953. Cited on p. 20.

Harary, Frank. On the measurement of structural balance. *Behavioral Science*, 4(4):316–323, 1959. Cited on p. 22.

Harary, Frank, Meng-Hiot Lim, and Donald C. Wunsch. Signed graphs for portfolio analysis in risk management. *IMA Journal of Management Mathematics*, 13(3):201–210, 2002. Cited on p. 23.

Håstad, Johan. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001. Cited on p. 19.

Held, Michael and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. Cited on p. 27.

Helwig, Sabine, Falk Hüffner, Ivo Rössling, and Maik Weinard. Algorithm design. In *Algorithm Engineering*, LNCS. Springer, 2007. To appear. Cited on pp. xi and 170.

Henzinger, Monika R., Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 43(2):222–250, 2000. Cited on p. 48.

Hoover, D. N. Complexity of graph covering problems for graphs of low degree. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 11:187–208, 1992. Cited on p. 15.

Hopcroft, John E. and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973. Cited on p. 47.

Hsu, Wen-Lian and Kuo-Hui Tsai. Linear time algorithms on circular-arc graphs. *Information Processing Letters*, 40(3):123–129, 1991. Cited on p. 15.

Hüffner, Falk. *Finding Optimal Solutions to Atomix*. Studienarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2002. Cited on p. 170.

Hüffner, Falk. *Graph Modification Problems and Automated Search Tree Generation*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2003. Cited on p. 170.

Hüffner, Falk. Algorithm engineering for optimal graph bipartization. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA '05)*, volume 3503 of *LNCS*, pages 240–252. Springer, 2005. Cited on pp. xii and 170.

Hüffner, Falk. Automated search tree generation. In *Encyclopedia of Algorithms*. Springer, 2007. To appear. Cited on pp. xi and 170.

Hüffner, Falk, Nadja Betzler, and Rolf Niedermeier. Optimal edge deletions for signed graph balancing. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA '07)*, volume 4525 of *LNCS*, pages 297–310. Springer, 2007a. Cited on pp. xii and 170.

Hüffner, Falk, Stefan Edelkamp, Henning Fernau, and Rolf Niedermeier. Finding optimal solutions to Atomix. In *Proceedings of the German Conference on Artificial Intelligence (KI '01)*, volume 2174 of *LNCS*, pages 229–243. Springer, 2001. Cited on p. 170.

Hüffner, Falk, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier. Fixed-parameter algorithms for cluster vertex deletion. In *Proceedings of the 8th Latin American Theoretical Informatics Symposium (LATIN '08)*, LNCS. Springer, 2008a. To appear. Cited on pp. xi, xii, and 170.

Hüffner, Falk, Rolf Niedermeier, and Sebastian Wernicke. Developing fixed-parameter algorithms to solve combinatorially explosive biological problems. In *Bioinformatics*, Methods in Molecular Biology Series. Humana Press, 2007b. To appear. Cited on pp. xi, 146, and 170.

Hüffner, Falk, Rolf Niedermeier, and Sebastian Wernicke. Fixed-parameter algorithms for graph-modeled data clustering. In *Clustering Challenges in Biological Networks*. World Scientific, 2007c. To appear. Cited on pp. xi, 70, and 170.

Hüffner, Falk, Rolf Niedermeier, and Sebastian Wernicke. Techniques for practical fixed-parameter algorithms. *The Computer Journal*, 51(1):7–25, 2008b. Cited on pp. xi, 3, and 170.

Hüffner, Falk, Sebastian Wernicke, and Thomas Zichner. Algorithm engineering for color-coding to facilitate signaling pathway detection. In *Proceedings of the 5th Asia–Pacific Bioinformatics Conference (APBC '07)*, volume 5 of *Advances in Bioinformatics and Computational Biology*, pages 277–286. Imperial College Press, 2007d. Extended version to appear in *Algorithmica*. Cited on p. xiii.

Hüffner, Falk, Sebastian Wernicke, and Thomas Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 2007e. To appear. Cited on pp. xiii and 170.

Hüffner, Falk, Sebastian Wernicke, and Thomas Zichner. FASPAD: fast signaling pathway detection. *Bioinformatics*, 23(13):1708–1709, 2007f. Cited on pp. xiii and 170.

Hunt, James W. and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977. Cited on p. 85.

Ideker, Trey, Vesteinn Thorsson, Jeffrey A. Ranish, Rowan Christmas, Jeremy Buhler, Jimmy K. Eng, Roger Bumgarner, David R. Goodlett, Ruedi Aebersold, and Leroy Hood. Integrated genomic and proteomic analyses of a systematically perturbed metabolic network. *Science*, 292(5518):929–934, 2001. Cited on p. 26.

Ideker, Trey and Alfonso Valencia. Bioinformatics in the human interactome project. *Bioinformatics*, 22(24):2973–2974, 2006. Cited on p. 26.

Kahng, Andrew B., Shailesh Vaya, and Alexander Z. Zelikovsky. New graph bipartizations for double-exposure, bright field alternating phase-shift mask layout. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 133–138. 2001. Cited on pp. 20 and 25.

Karger, David R., Rajeev Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997. Cited on p. 27.

Karp, Richard M. Reducibility among combinatorial problems. In Miller, Raymond E. and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. Cited on pp. 18 and 80.

Kellerman, E. Determination of keyword conflict. *IBM Technical Disclosure Bulletin*, 16(2):544–546, 1973. Cited on pp. 14, 15, 32, and 40.

Kenyon-Mathieu, Claire and Warren Schudy. How to rank with few errors: A PTAS for weighted feedback arc set on tournaments. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC '07)*, pages 95–103. ACM, 2007. Cited on p. 81.

Khomenko, Victor. Efficient automatic resolution of encoding conflicts using STG unfoldings. In *Proceedings of the 7th International Conference on Application of Concurrency to*

System Design (ACSD '07), pages 137–146. IEEE, 2007. Cited on pp. 16 and 44.

Khot, Subhash. On the power of unique 2-prover 1-round games. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC '02)*, pages 767–775. ACM, 2002. Cited on p. 18.

Khot, Subhash, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable CSPs? *SIAM Journal on Computing*, 37(1):319–357, 2007. Cited on p. 19.

Khot, Subhash and Venkatesh Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science*, 289(2):997–1008, 2002. Cited on pp. 14, 19, and 25.

Khot, Subhash and Oded Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008. Cited on p. 64.

Kneis, Joachim, Daniel Mölle, Stefan Richter, and Peter Rossmanith. Algorithms based on the treewidth of sparse graphs. In *Proceedings of the 31st International Workshop on Graph-Theoretic Concepts in Computer Science (WG '05)*, volume 3787 of *LNCS*, pages 385–396. Springer, 2005. Cited on p. 20.

Kneis, Joachim, Daniel Mölle, Stefan Richter, and Peter Rossmanith. Divide-and-color. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG '06)*, volume 4271 of *LNCS*, pages 58–67. Springer, 2006. Cited on pp. 27, 118, 130, and 144.

Kneis, Joachim, Daniel Mölle, and Peter Rossmanith. Partial vs. complete domination: t-dominating set. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '07)*, volume 4362 of *LNCS*, pages 367–376. Springer, 2007. Cited on p. 118.

Knuth, Donald E. *The Art of Computer Programming*, volume 4. Addison–Wesley, 2004. In preparation. Cited on pp. 67 and 96.

Koch, Ina. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1–2):1–30, 2001. Cited on p. 38.

Komusiewicz, Christian, Falk Hüffner, Hannes Moser, and Rolf Niedermeier. Isolation concepts for enumerating dense subgraphs. In *Proceedings of the 13th Annual International Conference on Computing and Combinatorics (COCOON '07)*, volume 4598 of *LNCS*, pages 140–150. Springer, 2007. Cited on pp. xi and 170.

Kőnig, Dénes. *Theorie der endlichen und unendlichen Graphen*. Akademische Verlagsgesellschaft, Leipzig, 1936. English translation: *Theory of finite and infinite graphs*, Birkhäuser, 1990. Cited on pp. 11 and 21.

Kou, Lawrence T., Larry J. Stockmeyer, and Chak-Kuen Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Communications of the ACM*, 21(2):135–139, 1978. Cited on pp. 15, 16, 32, and 40.

Koutis, Ioannis. A faster parameterized algorithm for set packing. *Information Processing Letters*, 94(1):7–9, 2005. Cited on p. 118.

Leroy, Xavier, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Available on the web, 1996. `http://caml.inria.fr/ocaml/`. Cited on pp. 39 and 54.

Lewis, John M. On the complexity of the maximum subgraph problem. In *Proceedings of the 10th ACM Symposium on Theory of Computing (STOC '78)*, pages 265–274. ACM,

1978. Cited on p. 14.

Lewis, John M. and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980. Cited on pp. 13 and 24.

Liers, Frauke, Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. Computing exact ground states of hard Ising spin glass problems by branch-and-cut. In Hartmann, Alexander K. and Heiko Rieger, editors, *New Optimization Algorithms in Physics*, pages 47–70. Wiley-VCH, 2004. Cited on p. 20.

Liu, Yang, Songjian Lu, Jianer Chen, and Sing-Hoi Sze. Greedy localization and color-coding: Improved matching and packing algorithms. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 84–95. Springer, 2006. Cited on p. 118.

Lovász, László and Michael D. Plummer. *Matching Theory*, volume 29 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1986. Cited on p. 76.

Lu, Songjian, Fenghui Zhang, Jianer Chen, and Sing-Hoi Sze. Finding pathway structures in protein interaction networks. *Algorithmica*, 48(8):363–374, 2007. Cited on p. 28.

Lund, Carsten and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP '93)*, volume 700 of *LNCS*, pages 40–51. Springer, 1993. Cited on pp. 14 and 24.

Lund, Carsten and Mihalis Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41(5):960–981, 1994. Cited on p. 15.

Ma, S., Walter D. Wallis, and J. Wu. Clique covering of chordal graphs. *Utilitas Mathematica*, 36:151–152, 1989. Cited on p. 15.

Mahajan, Meena and Venkatesh Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999. Cited on pp. 25 and 61.

Makhorin, Andrew. *GNU Linear Programming Kit Reference Manual Version 4.8*. Department of Applied Informatics, Moscow Aviation Institute, 2004. Cited on pp. 55, 110, 111, and 146.

Marx, Dániel. Parameterized complexity of constraint satisfaction problems. *Computational Complexity*, 14(2):153–183, 2005. Cited on p. 118.

Marx, Dániel. Chordal deletion is fixed-parameter tractable. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG '06)*, volume 4271 of *LNCS*, pages 37–48. Springer, 2006. Cited on pp. 61, 62, and 68.

Mathieson, Luke, Elena Prieto, and Peter Shaw. Packing edge disjoint triangles: A parameterized view. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC '04)*, volume 3162 of *LNCS*, pages 127–137. Springer, 2004. Cited on p. 118.

Mayrose, Itay, Tomer Shlomi, Nimrod D. Rubinstein, Jonathan M. Gershoni, Eytan Ruppin, Roded Sharan, and Tal Pupko. Epitope mapping using combinatorial phage-display libraries: a graph-based algorithm. *Nucleic Acids Research*, 35(1):69–78, 2007. Cited on pp. 119 and 126.

McKee, Terry A. and Fred R. McMorris. *Topics in Intersection Graph Theory*, volume 2 of *SIAM Monographs on Discrete Mathematics and Applications*. SIAM, 1999. Cited on p. 15.

Mehlhorn, Kurt, Rolf Möhring, Burkhard Monien, Petra Mutzel, Peter Sanders, and Dorothea Wagner. Beschreibung des DFG-Schwerpunktprogramms zum Thema Algorithm Engineering. Call for proposals of the Deutsche Forschungsgemeinschaft (DFG) (in German), 2006. Cited on p. 8.

von Mering, Christian, Roland Krause, Berend Snel, Michael Cornell, Stephen G. Oliver, Stanley Fields, and Peer Bork. Comparative assessment of large-scale data sets of protein–protein interactions. *Nature*, 417:399–403, 2002. Cited on p. 26.

Mi, Huaiyu, Betty Lazareva-Ulitsky, Rozina Loo, Anish Kejariwal, Jody Vandergriff, Steven Rabkin, Nan Guo, Anushya Muruganujan, Olivier Doremieux, Michael J. Campbell, Hiroaki Kitano, and Paul D. Thomas. The PANTHER database of protein families, subfamilies, functions and pathways. *Nucleic Acids Research*, 33(Supplement 1):284–288, 2005. Cited on p. 58.

Michalewicz, Zbigniew and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2005. Cited on p. 1.

Misiołek, Ewa and Danny Z. Chen. Two flow network simplification algorithms. *Information Processing Letters*, 97(5):197–202, 2006. Cited on p. 99.

Monien, Burkhard. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985. Cited on p. 27.

Moon, John W. On maximal transitive subtournaments. *Proceedings of the Edinburgh Mathematical Society*, 17:345–349, 1971. Cited on p. 86.

Moser, Hannes. 2007. Private communication. Cited on pp. 63 and 69.

Mujuni, Egbert and Frances A. Rosamond. Parameterized complexity of the clique partition problem. In *Proceedings of the 14th Computing: The Australasian Theory Symposium (CATS '08)*, volume 77 of *Conferences in Research and Practice in Information Technology*, pages 75–78. ACS, 2008. Cited on p. 15.

Niedermeier, Rolf. *Invitation to Fixed-Parameter Algorithms*. Number 31 in Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press, 2006. Cited on pp. 3 and 146.

Niedermeier, Rolf and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73(3–4):125–129, 2000. Cited on pp. 31, 38, and 147.

Niedermeier, Rolf and Peter Rossmanith. An efficient fixed-parameter algorithm for 3-hitting set. *Journal of Discrete Algorithms*, 1(1):89–102, 2003. Cited on pp. 65, 67, and 68.

Nilli, Alon. Perfect hashing and probability. *Combinatorics, Probability and Computing*, 3:407–409, 1994. Cited on p. 118.

O'Boyle, Michael and Elena Stöhr. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):529–543, 2002. Cited on p. 16.

Oda, Kanae, Tomomi Kimura, Yukiko Matsuoka, Akira Funahashi, Masaaki Muramatsu, and Hiroaki Kitano. Molecular interaction map of a macrophage. *AfCS Research Reports*, 2:14, 2004. Cited on p. 55.

Oda, Kanae and Hiroaki Kitano. A comprehensive map of the toll-like receptor signaling network. *Molecular Systems Biology*, 2:2006.0015, 2006. Cited on p. 58.

Okasaki, Chris and Andy Gill. Fast mergeable integer maps. In *Proceedings of the ACM SIGPLAN Workshop on ML*, pages 77–86. 1998. Cited on p. 39.

Orlin, James B. Contentment in graph theory: Covering graphs with cliques. *Indagationes Mathematicae (Proceedings)*, 80(5):406–424, 1977. Cited on pp. 15 and 44.

Orlova, G. I. and Ya. G. Dorfman. Finding the maximum cut in a graph (in Russian). *Tekhnicheskaja Kibernetika*, 3:155–159, 1972. English translation in *Engineering Cybernetics*, 10:502–506, 1972. Cited on p. 19.

Panconesi, Alessandro and Mauro Sozio. Fast hare: A fast heuristic for single individual SNP haplotype reconstruction. In *Proceedings of the 4th International Workshop on Algorithms in Bioinformatics (WABI '04)*, volume 3240 of *LNCS*, pages 266–277. Springer, 2004. Cited on pp. 25, 112, and 113.

Papadimitriou, Christos H. and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991. Cited on p. 18.

Peiselt, Thomas. *An Iterative Compression Algorithm for Vertex Cover*. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2007. Cited on pp. 63 and 69.

Piepho, Hans-Peter. An algorithm for a letter-based representation of all-pairwise comparisons. *Journal of Computational and Graphical Statistics*, 13(2):456–466, 2004. Cited on pp. 15, 16, 32, and 40.

Plehn, Jürgen and Bernd Voigt. Finding minimally weighted subgraphs. In *Proceedings of the 16th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '90)*, volume 484 of *LNCS*, pages 18–29. Springer, 1990. Cited on p. 27.

Polzin, Tobias and Siavash Vahdati Daneshmand. Practical partitioning-based methods for the Steiner problem. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA '05)*, volume 4007 of *LNCS*, pages 241–252. Springer, 2006. Cited on pp. 45 and 60.

Ponta, Oriana, Falk Hüffner, and Rolf Niedermeier. Speeding up dynamic programming for some np-hard graph recoloring problems. In *Proceedings of the 5th Annual Conference on Theory and Applications of Models of Computation (TAMC '08)*, volume 4978 of *LNCS*. Springer, 2008. To appear. Cited on pp. xi and 170.

Pop, Mihai, Daniel S. Kosack, and Steven L. Salzberg. Hierarchical scaffolding with Bambus. *Genome Research*, 14(1):149–159, 2004. Cited on p. 20.

Prieto, Elena and Christian Sloper. Looking at the stars. *Theoretical Computer Science*, 351(3):437–445, 2006. Cited on p. 118.

Prieto Rodríguez, Elena. *Systematic Kernelization in FPT Algorithm Design*. Ph.D. thesis, School of Electrical Engineering and Computer Science, The University of Newcastle, Australia, 2005. Cited on pp. 45 and 145.

Prisner, Erich. Graphs with few cliques. In *Proceedings of the 7th Conference on the Theory and Applications of Graphs*, pages 945–956. Wiley, 1995. Cited on p. 38.

Prömel, Hans-Jürgen and Angelika Steger. *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity*. Advanced Lectures in Mathematics. Vieweg, 2002. Cited on p. 5.

Protti, Fábio, Maise Dantas da Silva, and Jayme L. Szwarcfiter. Applying modular decomposition to parameterized bicluster editing. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 1–12. Springer, 2006. To appear under the title "Applying modular decomposition

to parameterized cluster editing problems" in *Theory of Computing Systems*. Cited on pp. 72 and 79.

Rahmann, Sven, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. Exact and heuristic algorithms for weighted cluster editing. In *Proceedings of the 6th International Conference on Computational Systems Bioinformatics (CSB '07)*, volume 6 of *Computational Systems Bioinformatics*, pages 391–401. Imperial College Press, 2007. Cited on pp. 5, 71, and 72.

Rajagopalan, Subramanian, Sreeranga P. Rajan, Sharad Malik, Sandro Rigo, Guido Araujo, and Koichiro Takayama. A re-targetable VLIW compiler framework for DSPs with instruction level parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1319–1328, 2001. Cited on p. 16.

Rajagopalan, Subramanian, Manish Vachharajani, and Sharad Malik. Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00)*, pages 157–164. ACM Press, 2000. Cited on pp. 15 and 16.

Raman, Venkatesh and Saket Saurabh. Parameterized algorithms for feedback set problems and their duals in tournaments. *Theoretical Computer Science*, 351(3):446–458, 2006. Cited on pp. 80 and 81.

Raman, Venkatesh and Saket Saurabh. Improved fixed parameter tractable algorithms for two "edge" problems: MAXCUT and MAXDAG. *Information Processing Letters*, 104(2):65–72, 2007. Cited on p. 19.

Raman, Venkatesh, Saket Saurabh, and Somnath Sikdar. Efficient exact algorithms through enumerating maximal independent sets and other techniques. *Theory of Computing Systems*, 41(3):563–587, 2007. Cited on pp. 25, 81, and 86.

Raman, Venkatesh and Somnath Sikdar. Parameterized complexity of the induced subgraph problem in directed graphs. *Information Processing Letters*, 104(3):79–85, 2007. Cited on p. 100.

Raymann, Dominik. *Implementation of Alon–Yuster–Zwick's Color-Coding Algorithm*. Diplomarbeit, Institut für Theoretische Informatik, ETH Zürich, Switzerland, 2004. Cited on p. 119.

Razgon, Igor and Barry O'Sullivan. Almost 2-SAT is fixed-parameter tractable. Technical Report arXiv:0801.1300, arxiv.org, 2008. Cited on pp. 63, 69, and 116.

Reed, Bruce, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, 2004. Cited on pp. xii, 19, 24, 61, 62, 101, 105, 106, and 111.

Rendl, Franz, Giovanni Rinaldi, and Angelika Wiegele. A branch and bound algorithm for max-cut based on combining semidefinite and polyhedral relaxations. In *Proceedings of the 12th International Conference on Integer Programming and Combinatorial Optimization (IPCO '07)*, volume 4513 of *LNCS*, pages 295–309. Springer, 2007. Cited on p. 20.

Rizzi, Romeo, Vineet Bafna, Sorin Istrail, and Giuseppe Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *Proceedings of the 2nd International Workshop on Algorithms in Bioinformatics (WABI '02)*, volume 2452 of *LNCS*, pages 29–43. Springer, 2002. Cited on p. 25.

Sasatte, Prashant. Improved FPT algorithm for feedback vertex set problem in bipartite tournament. *Information Processing Letters*, 105(3):79–82, 2007. Cited on p. 87.

Schrijver, Alexander. *Theory of Linear and Integer Programming*. Wiley, 1998. Cited on p. 109.

Schwikowski, Benno and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1–3):253–265, 2002. Cited on p. 86.

Scott, Alexander D. and Gregory B. Sorkin. Linear-programming design and analysis of fast algorithms for Max 2-CSP. *Discrete Optimization*, 4(3–4):260–287, 2007. Cited on p. 20.

Scott, Jacob, Trey Ideker, Richard M. Karp, and Roded Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *Journal of Computational Biology*, 13(2):133–144, 2006. Cited on pp. 26, 28, 117, 119, 126, 131, 133, 136, 137, 138, 140, 141, and 144.

Scott, Michelle S., Theodore Perkins, Scott Bunnell, François Pepin, David Y. Thomas, and Michael Hallett. Identifying regulatory subnetworks for a set of genes. *Molecular & Cellular Proteomics*, 4(5):683–692, 2005. Cited on p. 5.

Seymour, Paul D. and Robin Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993. Cited on p. 7.

Shamir, Ron, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144(1–2):173–182, 2004. Cited on p. 71.

Sharan, Roded. *Graph Modification Problems and their Applications to Genomic Research*. Ph.D. thesis, Sackler Faculty of Exact Sciences, School of Computer Science, Tel-Aviv University, 2002. Cited on p. 13.

Shlomi, Tomer, Daniel Segal, Eytan Ruppin, and Roded Sharan. QPath: a method for querying pathways in a protein–protein interaction network. *BMC Bioinformatics*, 7:199, 2006. Cited on pp. 119, 127, 128, and 140.

Sontag, Eduardo D. 2007a. Private communication. Cited on p. 100.

Sontag, Eduardo D. Monotone and near-monotone biochemical networks. *Systems and Synthetic Biology*, 1(2):59–87, 2007b. Cited on pp. 98 and 100.

Speckenmeyer, Ewald. On feedback problems in digraphs. In *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '89)*, volume 411 of *LNCS*, pages 218–231. Springer, 1989. Cited on p. 80.

Steffen, Martin, Allegra Petti, John Aach, Patrik D'haeseleer, and George Church. Automated modelling of signal transduction networks. *BMC Bioinformatics*, 2:34, 2002. Cited on pp. 26 and 28.

Sturmfels, Bernd. *Gröbner Bases and Convex Polytopes*, volume 8 of *University Lecture Series*. American Mathematical Society, 1996. Cited on p. 54.

Suthram, Silpa, Tomer Shlomi, Eytan Ruppin, Roded Sharan, and Trey Ideker. A direct comparison of protein interaction confidence assignment schemes. *BMC Bioinformatics*, 7:360, 2006. Cited on p. 26.

Tezzaron Semiconductor. Soft errors in electronic memory. White paper, 2004. Cited on p. 121.

Thagard, Paul and Karsten Verbeurgt. Coherence as constraint satisfaction. *Cognitive*

Science, 22(1):1–24, 1998. Cited on p. 22.

Truß, Anke. *Parametrisierte Algorithmen für Feedback-Set-Probleme auf Turniergraphen*. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2005. (Parameterized algorithms for feedback set problems in tournaments, in German). Cited on p. xii.

Vassilevska, Virginia, Ryan Williams, and Shan Leung Maverick Woo. Confronting hardness using a hybrid approach. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, pages 1–10. ACM, 2006. Cited on p. 28.

Vazirani, Vijay V. *Approximation Algorithms*. Springer, 2001. Cited on p. 1.

Volz, Erik. Random networks with tunable degree distribution and clustering. *Physical Review E*, 70(5):056115, 2004. Cited on pp. 58 and 137.

Wahlström, Magnus. *Algorithms, Measures and Upper Bounds for Satisfiability and Related Problems*. Ph.D. thesis, Department of Computer and Information Science, Linköpings universitet, Sweden, 2007. Cited on pp. 65, 68, 72, 81, and 87.

Watts, Duncan J. and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998. Cited on p. 137.

Weihe, Karsten. Covering trains by stations or the power of data reduction. In *Proceedings of the 1st Workshop on Algorithms and Experiments (ALEX '98)*, pages 1–8. 1998. Cited on p. 29.

Wernicke, Sebastian. *On the Algorithmic Tractability of Single Nucleotide Polymorphism (SNP) Analysis and Related Problems*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2003. Cited on pp. 19, 45, 87, 109, 110, 111, 112, 114, and 115.

Wernicke, Sebastian. *Combinatorial Algorithms to Cope with the Complexity of Biological Networks*. Ph.D. thesis, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2006. Cited on pp. 63 and 130.

Williams, Ryan. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2–3):357–365, 2005. Cited on p. 19.

Woeginger, Gerhard J. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156(3):397–405, 2008. Cited on pp. 81 and 86.

Xu, Jinbo, Feng Jiao, and Bonnie Berger. A tree-decomposition approach to protein structure prediction. In *Proceedings of the 4th International Conference on Computational Systems Bioinformatics (CSB '05)*, pages 247–256. IEEE, 2005. Cited on p. 7.

Yannakakis, Mihalis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981. Cited on pp. 18 and 24.

Zaslavsky, Thomas. Bibliography of signed and gain graphs. *Electronic Journal of Combinatorics*, DS8, 1998. Updated version available at http://www.math.binghamton.edu/zaslav/Bsg/. Cited on p. 20.

Zhang, Xiang-Sun, Rui-Sheng Wang, Ling-Yun Wu, and Luonan Chen. Models and algorithms for haplotyping problem. *Current Bioinformatics*, 1(1):104–114, 2006. Cited on p. 25.

Zhuang, Xiaotong and Santosh Pande. Resolving register bank conflicts for a network processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 269–278. IEEE Press, 2003. Cited on p. 25.

van Zuylen, Anke and David P. Williamson. Deterministic algorithms for rank aggregation

and other ranking and clustering problems. In *Proceedings of the 5th Workshop on Approximation and Online Algorithms (WAOA '07)*, volume 4927 of *LNCS*. Springer, 2007. To appear. Cited on p. 71.

Ehrenwörtliche Erklärung

Hiermit erkläre ich,

- dass mir die Promotionsordnung der Fakultät bekannt ist;
- dass ich die Dissertation selbst angefertigt und alle von mir benutzten Hilfsmittel, persönliche Mitteilungen sowie Quellen in meiner Arbeit angegeben habe;
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und dass Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen;
- dass ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe;
- dass ich weder die gleiche, eine in wesentlichen Teilen ähnliche, noch eine andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht habe.

Jena, Februar 2008 *Falk Hüffner*

Lebenslauf

Falk Hüffner
Geboren am 25. Februar 1976 in Oldenburg (Oldenburg)

2005–	**Doktorand** Friedrich-Schiller-Universität Jena Lehrstuhl Theoretische Informatik I/Komplexitätstheorie, Prof. Dr. Rolf Niedermeier
2004–2005	**Doktorand** Eberhard-Karls-Universität Tübingen Emmy-Noether-Nachwuchsgruppe „Kleine Parameter in schwierigen Problemen“, PD Dr. Rolf Niedermeier
2004	**Diplom** Diplomarbeitsthema: „Graph modification problems and automated search tree generation“ [Hüffner 2003] Betreuer: PD Dr. Rolf Niedermeier
1996–2004	**Diplom-Studiengang Informatik**, Nebenfach Mathematik Eberhard-Karls-Universität Tübingen
1995–1996	**Zivildienst** Institut für Vogelforschung, Wilhelmshaven
1988–1995	**Gymnasium** Lothar-Meyer-Gymnasium Varel, Germany Leistungsfächer: Mathematik und Physik
1986–1988	**Orientierungsstufe** Hauptschule/Orientierungsstufe Obenstrohe
1982–1986	**Grundschule** Grundschule Altjührden, Germany

Begutachtete Publikationen (bei Veröffentlichungen mit Konferenz- und Zeitschriftenversion ist nur die Zeitschriftenversion aufgeführt):

Hüffner et al. [2001], Gramm et al. [2002], Hüffner [2002], Guo et al. [2004], Gramm et al. [2004], Dom et al. [2005], Gramm et al. [2005], Hüffner [2005], Brosemann et al. [2006], Dom et al. [2006a,b], Guo et al. [2006], Dom et al. [2007], Gramm et al. [2007a,b], Guo et al. [2007a,b], Hüffner [2007], Hüffner et al. [2007a], Helwig et al. [2007], Hüffner et al. [2007c,b,f,e], Komusiewicz et al. [2007], Guo et al. [2008], Hüffner et al. [2008a,b], Ponta et al. [2008]

Jena, Februar 2008 *Falk Hüffner*