Exact Algorithms for Generalizations of Vertex Cover

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom-Informatiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

eingereicht von Hannes Moser

Betreuer: Prof. Dr. Rolf Niedermeier
Dipl.-Inf. Jiong Guo
Dipl.-Inf. Sebastian Wernicke

Jena, 09.11.2005

Zusammenfassung

Das Vertex Cover Problem ist ein Graphenproblem, das in der theoretischen Informatik intensiv untersucht wurde. Vertex Cover ist wie folgt definiert. Für einen gegebenen Graphen und eine positive ganze Zahl k ist zu bestimmen, ob eine Knotenmenge V' mit maximal k Knoten existiert, so dass jede Kante des Graphen zu mindestens einem Knoten aus V' inzident ist. Einige wichtige Generalisierungen dieses Problems sind Partial Vertex Cover, Connected Vertex Cover und Capacitated Vertex Cover, welche sowohl in der Theorie als auch in Anwendungen bedeutend sind. Vertex Cover sowie diese Generalisierungen sind jedoch NP-vollständig, d.h. es sind keine Algorithmen bekannt, die sie in Polynomialzeit lösen können. Wir müssen exponentielles Laufzeitverhalten grundsätzlich in Kauf nehmen, um optimale Lösungen dieser Probleme zu finden.

Wir verfolgen in dieser Arbeit den Ansatz sogenannter *parametrisierter Algorithmen*. Hierbei wird die Laufzeit im Gegensatz zur klassischen Komplexitätstheorie in Abhängigkeit der Eingabegröße *und* eines sogenannten *Problemparameters* gemessen. Es gibt verschiedene sinnvolle Problemparameter, wie zum Beispiel bei Vertex Cover die maximale Lösungsgröße k. Ein Problem heisst *festparameter-handhabbar* bezüglich Parameter k, wenn sich ein Lösungsalgorithmus mit Laufzeit $f(k) \cdot n^{O(1)}$ finden läßt, wobei n die Größe der Eingabeinstanz darstellt und die berechenbare Funktion f nur von k abhängt. Dies bedeutet, dass für kleine Werte von k und einem $f(k)$ wie z.B. 2^k der exponentielle Laufzeitanteil klein gehalten werden kann. Die Klasse solcher *parametrisierter Probleme*, welche festparameter-handhabbar sind, wird mit FPT bezeichnet.

Diese Arbeit knüpft an eine Arbeit an, in welcher gezeigt wurde, dass mit der Lösungsgröße als Parameter sowohl Connected Vertex Cover als auch Capacitated Vertex Cover festparameter-handhabbar sind, während Partial Vertex Cover es wohl nicht ist. Hier werden nun diese drei Generalisierungen unter einer anderen Parametrisierung betrachtet. Der verwendete Parameter ist die sogenannte *Baumweite*, welche die Ähnlichkeit eines Graphen zu einem Baum beschreibt. Die Arbeit ist dabei folgendermaßen aufgebaut.

Im ersten Kapitel geben wir einen Überblick zu Vertex Cover und seinen Generalisierungen und erwähnen die wichtigsten Begriffe, die in dieser Arbeit Anwendung finden.

Im zweiten Kapitel geben wir zuerst eine kurze Einführung in wichtige Begriffe aus der Graphentheorie. Danach erklären wir die Grundlagen der parametrisierten Komplexitätstheorie anhand von einfachen Beispielen.

Im dritten Kapitel geben wir die genauen Definitionen von Partial Vertex Cover, Connected Vertex Cover und Capacitated Vertex Cover an. Für diese Probleme präsentieren wir bekannte und neue Resultate und stellen typische Anwendungen vor.

Das vierte Kapitel ist das Hauptkapitel der Arbeit. In diesem Kapitel betrachten wir die Festparameter-Handhabbarkeit von Partial Vertex Cover, Connected Vertex Cover und Capacitated Vertex Cover bezüglich der Baumweite als Parameter. Die Baumweite und die damit zusammenhängenden sogenannten *Baumzerlegungen* für Graphen werden eingeführt, desweiteren erläutern wir die Technik des "dynamischen Programmierens auf Baumzerlegungen". Mit Hilfe von dynamischer Programmierung auf Baumzerlegungen geben wir für jede dieser Generalisierungen einen Lösungsalgorithmus an und analysieren dessen Laufzeit in Abhängigkeit der Baumweite. Damit können wir zeigen, dass Partial Vertex Cover sowie Connected Vertex Cover mit Baumweite als Parameter festparameterhandhabbar sind. Für Capacitated Vertex Cover zeigen wir, dass es für Graphen mit beschränktem Knotengrad festparameter-handhabbar ist, wenn der Parameter die Baumweite ist.

Im Anschluss dazu betrachten wir im fünften Kapitel eine Variante von Capacitated Vertex Cover, welche auch auf Bäumen NP-vollständig ist. Für diese Variante zeigen wir, dass sie mit dem maximalen Knotengrad als Parameter festparameter-handhabbar ist.

Zum Abschluss der Arbeit geben wir einen Überblick über die gewonnenen Erkenntnisse. Für die wichtigste verbleibende offene Frage, nämlich die Festparameter-Handhabbarkeit mit der Baumweite als Parameter für Capacitated Vertex Cover (im allgemeinen Fall), geben wir einige mögliche Ansätze an, die in dieser Arbeit nicht weiter untersucht worden sind.

Abstract

The NP-complete Vertex Cover problem has been intensively studied in the field of parameterized complexity theory. However, there exists only little work concerning important generalizations of Vertex Cover like Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover which are of high interest in theory as well as in real-world applications. So far research was mainly focused on the approximability of these problems. It was shown recently that, with the size of the vertex cover as parameter, Connected Vertex Cover and Capacitated Vertex Cover are both fixed-parameter tractable whereas Partial Vertex Cover is $W[1]$-hard. We will study the fixed-parameter tractability of these problems using another parameter, called the treewidth, which describes the "tree-likeness" of the input graph. Our dynamic programming approaches lead to exact algorithms for graph classes with small treewidth. With these algorithms we show that Partial Vertex Cover and Connected Vertex Cover are fixed-parameter tractable using treewidth as a parameter, and that Capacitated Vertex Cover is fixed-parameter tractable with respect to treewidth for graphs with bounded vertex degree. Additionally, we will consider a variant of Capacitated Vertex Cover which is NP-complete for trees. For this problem we show that it is fixed-parameter tractable when parameterized by the vertex degree.

Contents

Chapter 1

Introduction

1.1 Motivation

The Vertex Cover problem is one of the best-studied graph problems in theoretical computer science.

Vertex Cover is defined as follows:

> **Input:** An undirected graph $G = (V, E)$ and a nonnegative integer k.
>
> **Question:** Can we find a subset V' of at most k vertices such that each edge in E has at least one of its endpoints in V'?

In other words, if we define that a vertex *covers* all incident edges, then a *vertex cover* is a subset of vertices that covers all edges. The Vertex Cover problem is to decide whether there exists a vertex cover of size at most k.

Vertex Cover has many real-world applications, e.g., in network design. For instance, monitoring a communication network by placing devices at selected sites. Such a device can monitor the communication links incident to the site where it is located. The optimization criterion is to minimize the number of devices. Obviously, this is exactly the Vertex Cover problem [AKLSS05]. Another interesting field of application is bioinformatics [AKLSS05]. Vertex Cover finds applications in the construction of phylogenetic trees, in phenotype identification, and in analysis of microarray data [AKCF+04].

Unfortunately, the Vertex Cover problem is NP-complete [Kar72]. This means that, unless P = NP, there is no hope for a polynomial-time algorithm for Vertex Cover.

The next question is how to solve this NP-complete problem in practice. In other words, how can we deal with the presumable computational intractability of the problem? There are several general approaches for attacking NP-complete problems, among them approximation algorithms, fixed-parameter algorithms, and heuristics. The first two, approximation and fixed-parameter algorithms, are the approaches developed in theoretical computer science to address the Vertex Cover problem.

Approximation algorithms compute (in polynomial time) non-optimal solutions with a guaranteed performance. The performance of approximation algorithms is usually measured by a factor f, where the non-optimal solution differs at most f times from the optimal solution. For instance, Vertex Cover has a factor-2 approximation (or "2-approximation"), which means that there exists a polynomial-time algorithm that computes a vertex set that covers all edges with at most twice as many vertices as an optimal vertex cover. This can be shown with different methods, an approach is, e.g., to repeatedly select an arbitrary edge of the graph, adding its endpoints to the vertex cover and removing every edge incident to these endpoints in the graph, until there is no edge left. Since we know that there is at least one endpoint of each edge in a minimum vertex cover, the vertex cover obtained with this method has at most twice the number of vertices than an optimal solution. However, it was shown that, unless P = NP, the lower bound of the approximation factor for Vertex Cover is 1.36 [DS02].

Another interesting and promising alternative used to compute (optimal) solutions of NP-hard problems are *fixed-parameter algorithms* introduced by Downey and Fellows [DF99]. The idea is to restrict the unavoidable exponential running time of exactly solving algorithms, sometimes referred to as "*combinatorial explosion*", to a parameter, such that the problem can be efficiently solved in practice as long as the parameter is small. If there exists an algorithm with running time $f(k) \cdot n^{O(1)}$, where f is a computable function only depending on k, and n is the size of the input, then we call the problem *fixed-parameter tractable*.

Vertex Cover is one of the best-studied problems concerning fixed-parameter tractability. Many techniques in parameterized complexity, as for instance data reduction, depth-bounded search trees, and dynamic programming, were successfully applied to Vertex Cover, see e.g. in [ABF$^+$02, AFN04, AKCF$^+$04, CDRC$^+$03, CKJ01, Fel03, NR99, NR03]. Moreover, studies of Vertex Cover even led to new research directions within parameterized complexity [AR02, CDRC$^+$03, PS03].

Vertex Cover is fixed-parameter tractable with respect to the size k of the vertex cover. There exists a long list of continuous improvements on the combinatorial explosion [BFR98, CG05, CKJ01, NR99, NR03]. Beginning

from 1.32^k [BFR98], the best bound is now below 1.28^k [CG05].

Despite of the intensive studies of Vertex Cover (and also Weighted Vertex Cover [NR03]) in the field of parameterized complexity, there exists little work dealing with parameterized complexity of the following important generalizations of Vertex Cover:

1. For Partial Vertex Cover (PVC) we want to cover a certain number of edges with at most k vertices.

2. For Connected Vertex Cover (ConVC) we require that the subgraph induced by the vertex cover V' is connected.

3. For Capacitated Vertex Cover (CVC) we assign to each vertex a "covering capacity", such that a vertex can cover only a certain number of incident edges.

For formal definitions of these generalizations we refer to Section 3. These generalizations have many applications, as for instance in wireless network design and computational biology. So far PVC, ConVC, and CVC have been studied intensively concerning their approximability, e.g., in [BB98, CN02, GHK+03, GHKO03, GKS04, HS02]. They all possess polynomial-time factor-2 approximation algorithms. Recently, Guo et al. [GNW05] initiated the study of their fixed-parameter tractability. Considering the size k of the desired vertex cover as parameter, they show that Partial Vertex Cover appears to be fixed-parameter intractable, and that Connected Vertex Cover and Capacitated Vertex Cover are fixed-parameter tractable with combinatorial explosions 6^k and 1.2^{k^2}, respectively, which is still relatively high. Here we continue their research on the parameterized complexity of these generalizations. We aim to complement the results of Guo et al. by analyzing Partial Vertex Cover, Connected Vertex Cover, Capacitated Vertex Cover considering another parameter. Particularly, the fixed-parameter intractability of Partial Vertex Cover with respect to the size of the desired vertex cover as parameter highly motivates to look for some feasible parameterizations. There exist many meaningful parameters, e.g., the maximum vertex degree, or the number of edges covered by a partial vertex cover. The parameterization considered in this thesis is motivated by the observation that all three generalizations (PVC, ConVC, and CVC) are easy to solve on trees.[1] The next logical step is to ask for the

[1] This is trivial for Connected Vertex Cover, here we have to put all non-leaf vertices into the cover set. For Capacitated Vertex Cover see Guha et al. [GHKO03], and Partial Vertex Cover can be solved using dynamic programming on the input tree.

problem's complexity if the input graph is "almost" a tree. The parameter in this case would be a value describing the "tree-likeness" of a graph, which is small for graphs that are similar to trees. We use the concept of *treewidth* to measure the tree-likeness. Treewidth and the corresponding notion of *tree decomposition*, which describes the "tree-like structure" of a graph, were introduced by Robertson and Seymour [RS86] and play an important role in graph theory. Moreover, there exist several practical applications of the notion of treewidth, such as in expert systems, telecommunications, VLSI-design, Cholesky factorization, natural language processing, and programming languages, just to name a few [Bod88b, Bod93, Bod97]. Recently, tree decompositions with small treewidth have been successfully applied in computational biology to speed up significantly the search of RNA structures in genomes [SLM$^+$05, XJB05].

There are several techniques to solve problems on graphs with small treewidth. A standard technique is dynamic programming on tree decompositions [Alb03, Bod88a, Bod97, Nie06]. This technique is used for a vast number of known problems, a comprehensive list can be found, e.g., in [Bod88a]. Other techniques are for instance graph reductions for graphs with small treewidth [BdF96] and the use of monadic second order logic [Bod97].

In this work we will use the dynamic programming on tree decomposition technique to derive algorithms for Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover. In particular, we show that PVC and ConVC are fixed-parameter tractable with respect to the treewidth. Moreover, we show that CVC is fixed-parameter tractable with respect to the treewidth for graphs with bounded vertex degree. Also, we examine a generalization of CVC introduced by Guha et al. [GHKO03] which is already NP-complete for trees, and give a fixed-parameter algorithm to solve this problem restricted to trees, parameterized by the maximum vertex degree of the input tree.

1.2 Overview

The remaining part of this thesis is structured as follows. Chapter 2 is an introduction to several notions we use in this thesis. In Section 2.1 we give a short introduction to some basic notation, mainly from graph theory. We introduce in Section 2.2 the most important definitions of parameterized complexity theory, particularly fixed-parameter algorithms in Section 2.2.1 using Vertex Cover as an example. Also, we will give a brief introduction to parameterized reductions and fixed-parameter intractability in Section 2.2.2, where we use Independent Set as example.

Chapter 3 covers the formal definitions of Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover in more detail. We summarize known results concerning approximability and parameterized complexity. Our results obtained in Chapter 4 are also stated briefly. Moreover, we mention applications of PVC, ConVC, and CVC.

Chapter 4 is the main part of this thesis and contains several new results for Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover. We introduce tree decompositions and treewidth in Section 4.1. Then, we introduce the technique of dynamic programming on tree decompositions in Section 4.2. This technique is used to solve PVC, ConVC, and CVC in Sections 4.3, 4.4, and 4.5, respectively.

Chapter 5 introduces a generalized version of CVC. We study this generalization restricted to trees and we present a fixed-parameter algorithm to solve it with respect to the maximum vertex degree as parameter.

We conclude this thesis with a short summary and present an outlook of further work in Chapter 6.

Chapter 2

Preliminaries

This chapter summarizes some basic notations used throughout this work and provides a brief introduction to parameterized complexity theory.

2.1 Basic Notation from Graph Theory

A *graph* is defined as a pair $G = (V, E)$, where the elements of V are called *vertices* of G, $V := \{v_1, \ldots, v_n\}$, and the elements of E are called *edges*, $E \subseteq \{\{u, v\} : u, v \in V\}$. We denote the set of vertices of a graph G with $V(G)$, and the set of edges with $E(G)$. A vertex $v \in V$ is called *incident* with an edge $e \in E$ if $v \in e$. Two vertices $v, w \in V$ are called *adjacent* if there exists an edge $\{v, w\} \in E$, and v and w then are called *neighbors*. Two edges $e, f \in E$ are called *adjacent* if they share a vertex, that is, $e \cap f \neq \emptyset$. The *degree* $\deg(v)$ of a vertex v is the number of incident edges.

A *subgraph* $G' = (V', E')$ of a graph $G = (V, E)$ is a graph having $V' \subseteq V$ and $E' \subseteq \{e \in E : e \subseteq V'\}$. We also say that G *contains* G'. A subgraph G' of G is called an *induced subgraph* of G if every edge $\{u, v\} \in E$ with $u, v \in V'$ is a member of E'. The vertex set V' then *induces* G' in G. We write $G' = G[V']$.

A *path* is a graph $P = (V, E)$ with

$$V = \{v_1, v_2, \ldots, v_n\}, \;\; E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}\}$$

where V is a set of distinct vertices. The vertices v_1 and v_n are *connected* by P. A path connecting two vertices u, v is denoted by $P_{u,v}$. The *length* of a path is defined as the number of its edges.

A *connected graph* is a graph $G = (V, E)$ such that there exists a path $P_{u,v}$ for every pair $u, v \in V$. A *cycle* is a graph $C = (V, E)$ such that $(V, E \setminus \{e\})$

is a path for an arbitrary edge $e \in E$. A *tree* is a connected graph which does not contain cycles.

For a more detailed introduction to graph theory we refer to [Die05].

2.2 Parameterized Complexity

The purpose of this section is to introduce the general aspects of parameterized complexity. Detailed information can be found in the research monograph of Downey and Fellows [DF99]. The theoretical aspects of fixed-parameter intractability are only stated superficially.

2.2.1 Fixed-Parameter Tractability

Parameterized complexity theory [DF99] offers a two-dimensional framework for studying the computational complexity of problems. One dimension is the input size n and the other dimension the *parameter* k. The basic concept are parameterized problems and the notion of fixed-parameter tractability, which are defined in the following.

Definition 2.2.1. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$ for some finite alphabet Σ. For a problem instance $(x, k) \in L$, the second component denotes the* parameter.

Note that in this work, as in most publications, the parameter is a nonnegative integer. However, the above definition also permits more complicated parameters.

Definition 2.2.2. *A parameterized problem is* fixed-parameter tractable (fpt) *if it can be determined in $f(k) \cdot n^c$ time whether $(x, k) \in L$, where $n := |(x, k)|$ is the input size, f is a computable function only depending on k, and c is a constant.*

Fixed-parameter tractable problems are classified as FPT. Several common approaches exist to show that a problem actually is fixed-parameter tractable, as for instance

- data reduction rules (kernelization),
- depth-bounded search trees,
- dynamic programming, and
- iterative compression.

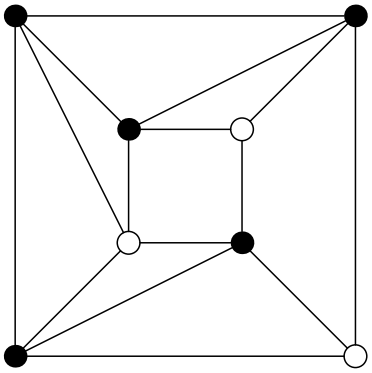


Figure 2.1: Example of a graph with a vertex cover of size 5 (black vertices). Observe that each edge has at least one of its endpoints in the cover. This vertex cover is optimal in the sense that there is no vertex cover with fewer than 5 vertices.

In this work we focus on the dynamic programming approach. One of the best-studied parameterized problems is the well-known Vertex Cover problem. In Figure 2.1 we give an example of a graph with a vertex cover of minimum size. This problem is fixed-parameter tractable with respect to the size of the vertex cover. There are several approaches to show that. Here we give a short description of a simple depth-bounded search tree algorithm with a combinatorial explosion 2^k. The idea of this algorithm is simple: To find a vertex cover of size at most k, we choose an arbitrary edge $e = \{a, b\}$, and then, since we know that at least one of a or b has to be in the vertex cover, we distinguish two cases whether a or b is a cover vertex. For each of these cases we choose an arbitrary uncovered edge and again distinct two cases for choosing an endpoint to be a part of the cover, and so forth, until we have selected k vertices. This recursive algorithm can be described with a tree structure called a *search tree*. The depth of such a tree is bounded by k, since in the k-th step of the algorithm we have chosen k vertices, so among all these possibilities to select k vertices there must be a solution to the problem, if such a solution exists. Obviously, the search tree has 2^k leaves, each of them representing a set of vertices, and we check whether at least one of them covers every edge. So, we have shown that Vertex Cover is fixed-parameter tractable with respect to the vertex cover size.

However, what can be done if it seems that the combinatorial explosion of an NP-complete problem cannot be restricted to a certain parameter? How can we actually show that an algorithm running in $f(k) \cdot n^c$ time is unlikely to be found? These questions are answered briefly in the next section.

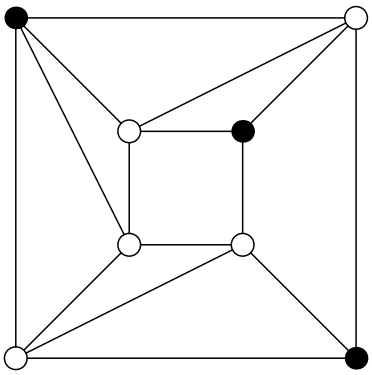


Figure 2.2: Example of a graph with an independent set of size 3 (black vertices). Observe that each black vertex has no other black vertex as neighbor. This independent set is optimal in the sense that there is no other independent set with more than 3 vertices.

2.2.2 Fixed-Parameter Intractability

Downey and Fellows [DF99] developed a formal framework to show that a problem is *fixed-parameter intractable*. We begin with some basic definitions.

To show that a NP-complete problem is unlikely to be fixed-parameter tractable, Downey and Fellows developed a completeness program similar to the classical complexity theory. The basic concept is parameterized reduction.

Definition 2.2.3. *A* parameterized reduction *from a parameterized problem L to another parameterized problem L' is a function that, given an instance (x, k), computes in time $f(k) \cdot n^c$ an instance (x', k') such that*

1. $(x, k) \in L \leftrightarrow (x', k') \in L'$,
2. *and k' only depends on k,*

where f is a function depending only on k, and c is a constant.

The basic complexity class for fixed-parameter intractability is $W[1]$. It can be defined as the class of parameterized languages that are equivalent to the Short Turing Machine Acceptance problem. This is the parameterized analogue to the Turing Machine Acceptance problem, which defines the basic NP-complete class. The conjecture that FPT $\neq W[1]$ is analogous to the conjecture that P $\neq$ NP. There are good reasons to believe that algorithms solving $W[1]$-hard parameterized problems with parameter k in $f(k) \cdot n^c$ time are unlikely to exist [DF99].

The Independent Set problem is an example of a $W[1]$-hard (and also $W[1]$-complete) problem and is closely related to Vertex Cover.
Independent Set:

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

Question: Can we find a subset I of V of size at least k such that I induces a subgraph of G without edges?

In Figure 2.2 we give an example of a graph with an independent set of maximum size. It is a well-known fact that a graph has a vertex cover of size k iff it has an independent set of size $n - k$ (all vertices not in the vertex cover must form an independent set, since an edge between such vertices would not be covered). Hence, the question whether or not a graph has a vertex cover of size k is equivalent to the question whether or not a graph has a independent set of size $n-k$. However, this reduction from Vertex Cover to Independent Set is *not* a parameterized reduction, since $k' = n - k$ is not only depending on k but also on n (see Definition 2.2.3).

For a deeper insight in this theory we refer the reader to the monograph of Downey and Fellows [DF99].

Chapter 3

Generalizations of Vertex Cover

Three natural generalizations of the Vertex Cover problem, namely Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover, have been intensively studied concerning their approximability. Recently, their fixed-parameter tractability with respect to the vertex cover size as parameter has been analyzed [GNW05]. Using the treewidth as a parameter, we study the fixed-parameter tractability of these problems in Chapter 4. In this chapter we give the definitions of these three generalizations, we provide an overview of known and new results, and we state possible applications.

3.1 Partial Vertex Cover

Definition 3.1.1. (Partial Vertex Cover)

Input: *An undirected graph $G = (V, E)$ and two nonnegative integers k and t.*

Question: *Can we find a subset V' of V of size at most k such that at least t edges are incident to vertices in V'?*

We abbreviate Partial Vertex Cover with PVC.

Approximation: The PVC problem is NP-complete and known to have a factor-2 approximation [BB98, GKS04]. With d denoting the maximum vertex degree of the input graph, an approximation factor of $(2 - \Theta(1/d))$ was developed, e.g., in [GKS04]. The current-best approximation factor is $(2 - \Theta(\frac{\ln \ln d}{\ln d}))$ [HS02].

Fixed-parameter tractability: PVC is fixed-parameter tractable with respect to the number t of edges to be covered [Blä03]. When parameterized by the size k of the cover, Partial Vertex Cover as well as the minimization version of the problem, i.e., to cover at most t edges with at least k vertices, are $W[1]$-complete [GNW05].

New results: We show in Section 4.3 that Partial Vertex Cover is fixed-parameter tractable with the treewidth ω of the input graph as parameter. The running time of our algorithm is $O(2^\omega \cdot k \cdot (\omega^2 + k) \cdot n)$, where n is the number of vertices in the input graph.

Applications: Partial Vertex Cover is closely related to facility location. Suppose a network of cities and streets where we need to build facilities to provide service (e.g., maintenance) to a certain fraction of the streets (due to limited public funds). A facility in a city provides service to all outgoing streets. We can model this as a Partial Vertex Cover problem, where we ask which cities should be chosen to build the facilities [GKS04].

3.2 Connected Vertex Cover

Definition 3.2.1. (Connected Vertex Cover)

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

Question: *Can we find a vertex cover V' of size at most k such that $G[V']$ is a connected subgraph?*

We abbreviate Connected Vertex Cover with ConVC. By introducing a weight function there are two variants of ConVC, namely Tree Cover and Tour Cover. For a given graph G with weight function $w : E \to \mathbb{N}^+$, an integer $k \geq 0$, and a real number $W \in \mathbb{R}^+$, the Tree Cover problem is to determine whether there exists a subgraph $G' = (V', E')$ of G with at most k vertices and $\sum_{e \in E'} w(e) \leq W$ such that V' is a vertex cover on G and G' is a tree. The unweighted version of Tree Cover is equivalent to the Connected Vertex Cover problem. The second variant is Tour Cover, where we require that the edges in G' form a closed walk (where vertices and edges can be used repeatedly).

Approximation: The ConVC problem is NP-complete and polynomial-time approximable within factor 2, Tree Cover is approximable within

factor 3.55 and Tour Cover within factor 5.5 [AHH93]. The approximation factors of Tree Cover and Tour Cover were improved to 3 [KKPS03].

Fixed-parameter tractability: The ConVC problem is fixed-parameter tractable with respect to the vertex cover size k. Guo, Niedermeier, and Wernicke [GNW05] show that Connected Vertex Cover can be solved in $O(6^k n + 4^k n^2 + n^2 \log n + nm)$ time, Tour Cover in $O(2^k k^{2k} m)$ time, and Tree Cover in $O(6^k k^2 n + 4^k k^2 n^2 + kn^3)$ time, where n and m denote the number of vertices and edges of the input graph, respectively.

New results: In Section 4.4 we show that ConVC is fixed-parameter tractable with the treewidth ω of the input graph as parameter. The running time of our algorithm is $O(2^\omega \cdot \omega^{3\omega+2} \cdot n)$, where n is the number of vertices of the input graph.

We remark that the connected version of Dominating Set has been studied by Demaine and Hajiaghayi [DH05]. They show that Connected Dominating Set can be solved for graphs with treewidth ω in $O(\omega^\omega \cdot n)$ time, where n is the number of vertices of the input graph [DH05].

Applications: The Connected Vertex Cover problem occurs for instance in the field of wireless network design. In a wireless network, network nodes (vertices) are connected by transmission links (edges). To operate the network we need to place relay stations on nodes, such that the relay stations form a connected subnetwork and every transmission link is incident to a relay station. The optimization criterion is to minimize the number of relay stations. This is exactly the Connected Vertex Cover problem. Other variants are for instance wireless networks with less failure vulnerability, demanding that the failure of a relay station does not destroy the connectivity of the relay station network [Gro05]. The Tree Cover and Tour Cover problems are motivated by their close relation to Vertex Cover, Watchman Route, and Traveling Purchaser [AHH93, KKPS03].

3.3 Capacitated Vertex Cover

Definition 3.3.1. *A* capacitated graph *consists of a graph $G = (V, E)$ and a* capacity *function $c : V \to \mathbb{N}^+$ which assigns an integer $c(v) \geq 1$ as v's capacity to each vertex v. Given a vertex cover C of a graph $G = (V, E)$, we call C a* capacitated vertex cover *(cvc) if there exists an assignment which assigns each edge in E to one of its endpoints, such that the number of edges assigned to any vertex $v \in V$ does not exceed $c(v)$.*

Definition 3.3.2. (Capacitated Vertex Cover)

Input: *A capacitated graph $G = (V, E)$, a nonnegative integer k, a vertex* weight $w : V \to \mathbb{R}^+$*, and a positive real number W.*

Question: *Can we find a capacitated vertex cover C of size at most k such that $\sum_{v \in C} w(v) \leq W$?*

We abbreviate Capacitated Vertex Cover with CVC. There exist two variants of this problem which allow using multiple copies of a vertex. That is, if we use d copies of a vertex v with capacity $c(v)$, then v can cover up to $d \cdot c(v)$ edges. We differentiate between Soft Capacitated Vertex Cover (Soft CVC), where the number of allowed copies of a vertex is not restricted, and Hard Capacitated Vertex Cover (Hard CVC), where the number of allowed copies of a vertex is restricted for each vertex individually.

Approximation: The CVC problem was introduced by Guha, Hassin, Khuller, and Or [GHKO03]. They show that this problem is NP-complete and approximable within factor 2. They also studied several generalizations, as well as the problem restricted to trees. The unweighted version of Hard CVC is significally harder so solve than CVC. Chuzhoy and Naor [CN02] developed a factor-3 approximation for Hard CVC. They also showed that the weighted version of Hard CVC is as least as hard to approximate as Set Cover. Gandhi et al. [GHK$^+$03] improved the approximation factor for unweighted Hard CVC to 2.

Fixed-parameter tractability: CVC is fixed-parameter tractable with respect to the vertex cover size k and can be solved in $O(1.2^{k^2} + n^2)$ time and has a problem kernel of size $O(4^k \cdot k^2)$ [GNW05].

New results: In Section 4.5 we give an algorithm that solves CVC in time $O(k^{2\omega} \cdot \omega \cdot n)$, where ω and n denote the treewidth and the number of vertices of the input graph, respectively. This algorithm also shows that the problem is fixed-parameter tractable for graphs with bounded vertex degree d when parameterized by the treewidth of the input graph. Then the running time can be expressed as $O(d^{2\omega} \cdot \omega \cdot n)$. However, it remains open whether CVC on general graphs is fixed-parameter tractable with respect to treewidth.

Applications: The CVC problem has an interesting application in bioinformatics. Guha et al. [GHKO03] refer to Glycodata[1], a company researching in the areas of glycobiology and bioinformatics. One of its projects is a chip-based technology called GMID (Glycomolecule ID) which is used to uniquely identify glycomolecules in a given liquid solution. In a simplified view, such glycomolecules consist of building blocks. Methods to identify these building blocks exist. However, in order to identify a glycomolecule it is necessary to determine the connectivity of the building blocks. The GMID chip determines in a single application for a building block A and a set of building blocks S, whether A is connected to B for each building block $B \in S$. The size of S is limited due to technical reasons. To plan an experiment to identify a glycomolecule, the necessary information can be represented as a capacitated graph, where the building blocks are vertices with capacity $|S|$, and an edge exists between two vertices if the information about their connectivity is required. The problem of minimizing the number of chip applications is exactly the Capacitated Vertex Cover problem with uniform capacities.

Interesting Generalizations: Guha et al. [GHKO03] also considered a generalization of Capacitated Vertex Cover, the so-called Minimum Capacitated Vertex Cover with Demand (CVCD) problem, and showed that CVCD is NP-complete on trees. In Section 5 we summarize the results from Guha et al. [GHKO03], and for one NP-complete variant of CVCD we give an algorithm that shows that it is fixed-parameter tractable with the maximum vertex degree as parameter.

3.4 Summary

Interestingly, Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover all behave more or less in the same way from the viewpoint of polynomial-time approximability, as all have factor-2 approximations. However, the picture becomes completely different from a parameterized complexity point of view: Parameterized by the solution size k, Partial Vertex Cover appears to be fixed-parameter intractable, whereas Connected Vertex Cover and Capacitated Vertex Cover are fixed-parameter tractable. In the next chapter, we attack these generalizations from another parameterized complexity point of view, where the parameter is the treewidth of the given input graph.

[1] http://www.glycodata.com

Chapter 4

Dynamic Programming on Tree Decompositions

In Chapter 3 we introduced Partial Vertex Cover (PVC), Connected Vertex Cover (ConVC), and Capacitated Vertex Cover (CVC). When these problems are parameterized by the size k of a minimum solution, then CVC and ConVC are fixed-parameter tractable while PVC is $W[1]$-complete.

In this chapter we study the parameterized complexity of these problems with respect to *treewidth*, a parameter which describes the tree-likeness of a graph. Parameterized by the treewidth ω of a given n-vertex graph, we show that

- Partial Vertex Cover is fixed-parameter tractable and can be solved in $O(2^\omega \cdot k \cdot (\omega^2 + k) \cdot n)$ time (Section 4.3),
- Connected Vertex Cover is fixed-parameter tractable and can be solved in $O(2^\omega \cdot \omega^{3\omega+2} \cdot n)$ time (Section 4.4), and
- Capacitated Vertex Cover is fixed-parameter tractable for graphs with bounded vertex degree d and it can be solved in $O(d^{2\omega} \cdot \omega \cdot n)$ time (Section 4.5).

Moreover, we show that CVC can be solved in $O(k^{2\omega} \cdot \omega \cdot n)$ time (Section 4.5). The algorithms to solve these problems use a technique called *dynamic programming on tree decompositions*.

This chapter is organized as follows. First, we give an introduction to treewidth and the underlying concept of *tree decomposition*. After that, we describe the technique of dynamic programming on tree decompositions. Using this technique, we present the algorithms for PVC and ConVC which

lead to the results stated above. The subsequent studies of CVC are divided into two parts. First, we give an algorithm to solve CVC on graphs with treewidth at most two, then we generalize this approach to obtain an algorithm for CVC for graphs with treewidth ω. With this general approach we can also show that CVC is fixed-parameter tractable with treewidth as a parameter for graphs with bounded vertex degree.

4.1 Tree Decompositions

The concept of tree decompositions for graphs was introduced by Robertson and Seymour [RS86] and plays an important role in algorithmic graph theory. In this section we give an introduction to *tree decomposition*, *treewidth*, and *nice tree decomposition*. These notions are needed for the dynamic programming on tree decompositions.

Tree decompositions are motivated by the observation that many NP-complete problems are easy to solve on trees. We are then interested how such problems can be solved for graphs that are similar to trees. Tree decompositions are a formal way to describe the "tree-likeness" of a given graph.

Definition 4.1.1. (Tree Decomposition)
Given a graph $G = (V, E)$, a pair $(\{X_i : i \in I\}, T)$ is called a tree decomposition*, where each $X_i \subseteq V$ is called a* bag*, and $T = (I, F)$ 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 $e \in E$ there exists a bag X_i with $i \in I$ and $e \subseteq X_i$, and*
3. *for all $i, j, k \in I$, if j lies on the path from i to k in T then $X_i \cap X_k \subseteq X_j$.*

The third property is equivalent to the requirement that, for each $v \in V$, the nodes of all bags containing v induce a subtree of T.

Observe that the size of the bags is influenced by the given graph G: For a tree, there exists a tree decomposition where the size of each bag equals two. Such a tree decomposition consists of a node i for each edge e_i of the tree, each bag only contains both vertices incident to e_i, and two nodes i, j are adjacent in the decomposition tree if $e_i \cap e_j \neq \emptyset$. In contrast, a clique K_n of n vertices has minimum bag size n. The corresponding tree decomposition consists of only one node whose corresponding bag contains all n vertices of K_n (due to property (3) of Definition 4.1.1). A similar observation is that a graph containing a complete size-k subgraph G' has bags of size at least k since there has to exist a bag containing all vertices of G'. Loosely speaking,

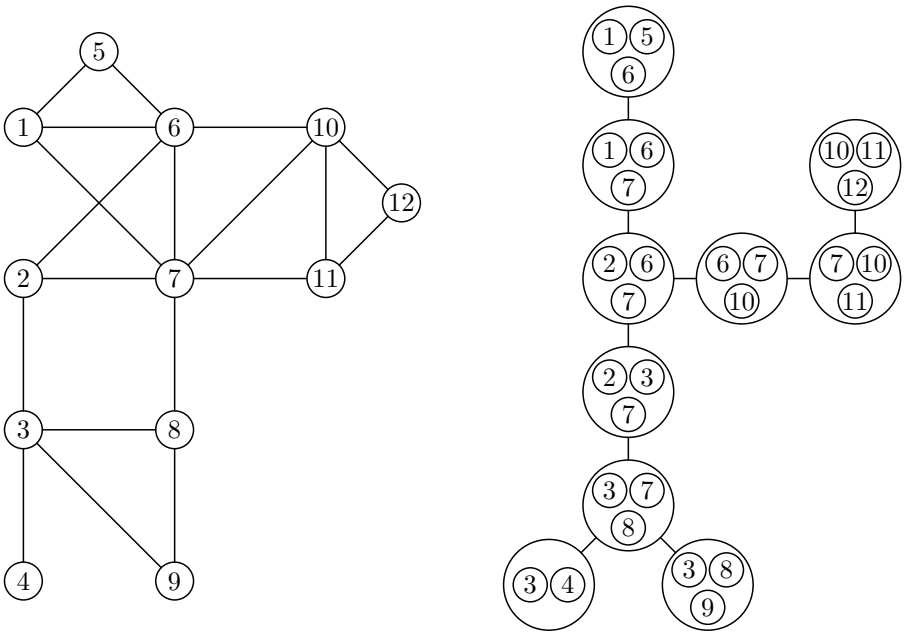


Figure 4.1: Example of a graph G and a corresponding tree decomposition $(\{X_i : i \in I\}, T)$. It is optimal in the sense that there is no tree decomposition for G such that every bag contains fewer than three vertices. Observe that the properties of tree decompositions as stated in Definition 4.1.1 hold.

bags are smaller if the given graph is more "tree-like". This idea leads to the following definition of treewidth.

Definition 4.1.2. (Treewidth)
The width *of a tree decomposition* $(\{X_i : i \in I\}, T)$ *is defined as*

$$\max\{|X_i| : i \in I\} - 1.$$

The treewidth *of a graph* G *is defined as the minimum width over all tree decompositions of* G*.*

Thus, a tree has treewidth 1 and a clique K_n has treewidth $n-1$. In Figure 4.1 we give an example of a graph and a tree decomposition of width 2. The concept of treewidth measures how tree-like a given graph is. However, it is not trivial to compute a tree decomposition for a given graph. A limiting factor of the dynamic programming technique using tree decompositions is the construction of tree decompositions of *small* width. Given an n-vertex graph G and an integer ω, the problem to determine whether the treewidth of G is at most ω is NP-complete [ACP87]. However, if the parameter ω is a fixed constant, then the problem can be decided in time $O(2^{\Theta(\omega^3)} \cdot n)$,

and a corresponding tree decomposition can be constructed within the same running time [Bod96]. The drawback of this result is the constant factor of $2^{\Theta(\omega^3)}$. For this reason several different approaches to compute tree decompositions were developed, for instance heuristic algorithms [KBH02], graph reduction [ACPS93], parallel processing [BH98], and approximation algorithms (see, e.g., [BGHK95, Ree92, DHT02]). Especially for small values of ω ($2, 3$, or 4) there exist efficient linear time algorithms based on graph reduction [AP86, San96].

For this thesis it is important that the problem to determine whether the treewidth of a graph is at most ω is fixed-parameter tractable with respect to treewidth [Bod96], and that a tree decomposition can be constructed in $f(\omega) \cdot n^{O(1)}$ time, where f is a function only depending on ω, and n is the number of vertices of the input graph.

For the description of tree decomposition based algorithms it is convenient to use a *nice tree decomposition*, which has a particularly simple structure. Each node of such a nice tree decomposition has a type with certain properties, which makes the use of such a nice tree decomposition easier. For the following definition of *nice tree decompositions* we define a *binary tree* in the sense that we only allow vertices of degree one, two, and three.

Definition 4.1.3. (Nice Tree Decomposition)
A nice tree decomposition $(\{X_i : i \in I\}, T)$ *for a graph* $G = (V, E)$ *is a tree decomposition for* G *with the following properties.*

- T *is rooted at a designated node* $r \in I$*, called root node.*
- T *is a binary tree.*
- *The nodes of* T *are of one of the following four node types:*
  1. Leaf nodes i *which have no children and the corresponding* leaf bags X_i *have* $|X_i| = 1$.
  2. Introduce nodes i *which have one child* j *with* $X_i = X_j \cup \{v\}$ *for some vertex* $v \in V$.
  3. Forget nodes i *which have one child* j *with* $X_j = X_i \cup \{v\}$ *for some vertex* $v \in V$.
  4. Join nodes i *which have two children* $j, l \in I$ *with* $X_i = X_j = X_l$.

Introduce, forget, and join nodes are called *inner nodes*. In Figure 4.2 we give an example of a nice tree decomposition for the graph shown in Figure 4.1. We can easily construct a nice tree decomposition from a tree decomposition as stated in the following lemma.

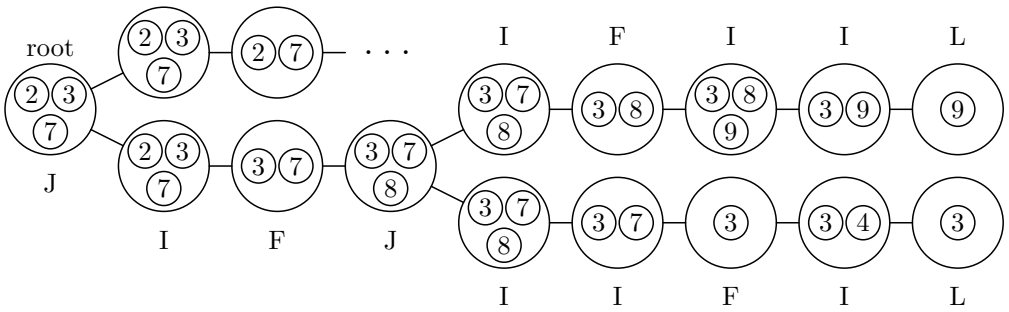


Figure 4.2: A part of a nice tree decomposition of the graph in Figure 4.1. Each node is marked with a letter denoting the node type: Leaf node (L), Introduce node (I), Forget node (F), and Join node (J).

Lemma 4.1.1. [Klo94, Lemma 13.1.3] *Given a tree decomposition for an n-vertex graph G that has $O(n)$ nodes and width ω,*[1] *we can find a nice tree decomposition of G that has $O(n)$ nodes and the same width ω in time $O(n)$.*

Note that nice tree decompositions do not provide more algorithmic possibilities. Rather, they make the description of dynamic programming algorithms easier. Therefore, we use nice tree decompositions to describe the technique of dynamic programming on tree decompositions in the next section.

4.2 Dynamic Programming on Tree Decompositions

We apply the concept of tree decomposition to design algorithms using the tree-like structure of a given graph. The usual approach of tree decomposition based algorithms is dynamic programming. In the following we give an introduction to this technique applied to *nice* tree decompositions. We use this technique in Sections 4.3 and 4.4 to show fixed-parameter tractability with treewidth as parameter for Partial Vertex Cover and Connected Vertex Cover. Using the same technique, we show in Section 4.5 that Capacitated Vertex Cover is fixed-parameter tractable with respect to treewidth for graphs with bounded vertex degree.

First, we introduce some basic notation used to describe the dynamic programming on a nice tree decomposition. Given a graph G and a nice tree decomposition $(\{X_i : i \in I\}, T)$ for G with treewidth ω which is rooted at node $r \in I$, we want to solve a problem on G (as for instance Vertex

[1] Note that for an n-vertex graph G that has treewidth ω there always exists a tree decomposition of width ω that has $O(n)$ nodes [Klo94, Lemma 2.2.5].

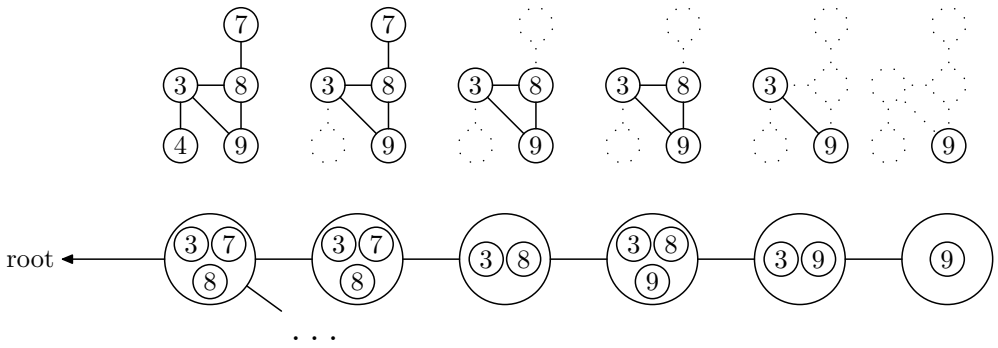


Figure 4.3: A small part of our nice tree decomposition (see Figure 4.2), together with the corresponding graph G_i for each node i.

Cover). Let $T[i]$ denote the subtree rooted at node i. We assign each node $i \in I$ the subgraph

$$G_i = (V_i, E_i) := G[\bigcup_{j \in T[i]} X_j].$$

Note that G_r corresponds to the whole input graph G. In Figure 4.3 we give an example of subgraphs G_i assigned to nodes i of a nice tree decomposition. It is an interesting property of such a subgraph G_i that paths from vertices in G_i to vertices not in G_i *always* contain vertices in X_i. This fact is formally described with the notion of *separator*.

Definition 4.2.1. *Given a connected graph $G = (V, E)$, a* separator *of G is a subset $S \subseteq V$ such that the induced subgraph $G[V \setminus S]$ is not connected.*

The following lemma shows that the property of G_i described above holds.

Lemma 4.2.1. [Die05, Lemma 12.3.1.] *Given a nice tree decomposition for a connected graph $G = (V, E)$, each non-leaf bag X_i is a separator of G.*

As one consequence, while solving some problem on a graph G, we can process $G[V_i \setminus X_i]$ independently from $G[V \setminus V_i]$ if we fix the solution in X_i. Once we computed all solutions on G_i for a node i, we can reuse them to compute solutions on G_j, where j is the parent node of i. The idea of dynamic programming on tree decompositions is to use tables A_i for each bag X_i to represent feasible solutions on G_i. A table A_i for an inner node i is computed using the graph $G[X_i]$ and the information of the tables corresponding to the child(ren) of i. The tables are computed in a bottom-up manner from the leaves to the root. The entries of the root table then represent possible solutions to the problem on $G_r = G$. The following example of dynamic programming on a nice tree decomposition for Vertex Cover will point this out.

Table description: In the table description we define the tables used for the dynamic programming on a nice tree decomposition.

Example. Suppose that we want to solve Vertex Cover on G with a given nice tree decomposition $(\{X_i : i \in I\}, T)$. We use for each tree node i a table A_i that has $2^{|X_i|}$ rows corresponding to all possible *configurations* of whether or not a vertex in X_i is chosen as cover vertex. For each configuration we store the size of a vertex cover on G_i in the corresponding table entry such that the vertex cover has minimum size assuming the given configuration. In other words, each configuration represents a vertex cover on G_i, and the corresponding entry tells us how many cover vertices it needs.

Next, we have to describe how the table entries are computed for each table. In the following we describe the necessary steps of such a dynamic programming.

Initialization step: In this step we compute the tables for the leaf nodes.

Example (continued). Table A_i of each leaf node i is computed as follows: We verify for each possible configuration in A_i whether it represents a vertex cover on $G_i = G[X_i]$. If so, we store the size of the cover in the corresponding table entry. If not, we store a special value denoting that the corresponding configuration is "invalid "(that is, not every edge in G_i is covered).

After computing the tables of the leaf nodes, we have to compute the tables for the inner tree nodes. We refer to this as *updating process*.

Updating process: Recall that we are working on *nice* tree decompositions. We distinguish introduce, forget, and join nodes to compute the tables A_i for the inner nodes i. (Typically, the processing of the join nodes is more cost-expensive than for the other node types since this computation involves two child tables instead of only one.) Here we give an example for computing A_i for a join node, using again the Vertex Cover problem. The computation for the other node types is not exemplified here.

Example (continued). Suppose that i is a join node with children j and l. The entries for each configuration in A_i are computed by looking up the two entries of A_j and A_l of the same configuration, i.e, with the same vertex cover on $G[X_i]$. So for each configuration we possess the size s_j of a vertex cover on G_j and the size s_l of a vertex cover on G_l. To compute the corresponding entry in A_i, we add s_j and s_l and subtract the number of cover vertices in X_i, since this number is already counted in s_j and s_l. If s_j or s_l is invalid, then the corresponding entry of A_i is set to "invalid".

Solution: We observe the entries of the root table and verify whether there is an entry that corresponds to a solution to the problem.

Example (continued). To get a solution to the Vertex Cover problem, we have to look up an entry of the root table with minimum value. This value is the size of a minimum vertex cover on the input graph. If the value does not exceed the maximum vertex cover size k, then the algorithm returns "YES-Instance"; otherwise, "NO-Instance".

This concludes our example of the technique of dynamic programming on a nice tree decomposition. The next section gives an algorithm to solve Partial Vertex Cover using this technique.

4.3 Partial Vertex Cover

We saw how the dynamic programming technique can be used to solve Vertex Cover. Using the same technique, we address Partial Vertex Cover (PVC) in this section. First, we will give a dynamic programming algorithm to solve PVC in Section 4.3.1. After that, we conclude with the analysis of its running time in Section 4.3.1, showing that PVC can be solved in $O(2^\omega \cdot k \cdot (\omega^2 + k) \cdot |I|)$ time, where k denotes the maximal size of the desired partial vertex cover, and ω and $|I|$ denote the treewidth and the number of nodes of the nice tree decomposition, respectively. This means that PVC is fixed-parameter tractable when parameterized by the treewidth.

4.3.1 The Algorithm

Given is an instance of Partial Vertex Cover (PVC), that is, a graph G and integers $k \geq 0$ and $t \geq 0$, where k is the maximal size of the cover, and t is the minimum number of edges to be covered, and a nice tree decomposition $(\{X_i : i \in I\}, T)$ for G of width ω. We give a dynamic programming algorithm which computes the maximum number t' of edges covered by a partial vertex cover of size at most k. If $t' \geq t$, the algorithm returns that the instance is a "YES-instance"; otherwise, it is a "NO-instance". As in the example for Vertex Cover in Section 4.2 we use tables for the dynamic programming on the nice tree decomposition. Remember the definition of a graph $G_i = (V_i, E_i)$ for each node i of the nice tree decomposition (see page 4.2).

The difference of PVC compared to VC is the following. In the case of VC in Section 4.2, we computed in each row of the table the size of an optimal vertex cover on G_i assuming the corresponding configuration. In

the case of PVC, the problem is to decide how many vertices in G_i to use as cover vertices in order to cover a optimal number of edges (assuming the corresponding configuration). However, we cannot know which number of cover vertices in G_i is optimal such that the overall solution covers at least t edges with at most k vertices, assuming a given configuration of whether or not a vertex in X_i is a part of the cover. The main idea for the following algorithm is to let the number of cover vertices in $V_i \setminus X_i$ be a part of each configuration.

Table description: We define for each node i of the nice tree decomposition a table A_i that has $2^{|X_i|} \cdot k$ rows corresponding to all possible configurations of whether or not a vertex X_i is a part of the cover, and of how many vertices in $V_i \setminus X_i$ are a part of the cover. For each configuration we store the maximum number of covered edges in G_i in the corresponding table entry.

Unlike in the example of Vertex Cover, we describe the tables in more detail as we want to state the algorithm more formally. Therefore, we need a concept to represent the configurations corresponding to each row of a table. For this reason we introduce *2-colorings* to the vertices in a bag X_i, where color "1" specifies that a vertex is a part of the cover, and color "0" specifies that it is not a part of the cover.

Definition 4.3.1. *Suppose a bag $X_i = \{v_1, \ldots, v_{n_i}\}$ and assume that the vertices are ordered by their indices. A* 2-coloring *c of X_i is a vector*

$$c = (c_1, c_2, \ldots, c_{n_i}) \in \{0,1\}^{n_i},$$

such that vertex v_j has color c_j for all $1 \leq j \leq n_i$. We write $c(v_j)$ to denote the color of vertex v_j in 2-coloring c.

In the dynamic programming, such 2-colorings for a bag have to be combined with other 2-colorings. We define the corresponding operation formally in the following. Given are two disjoint vertex sets V and W and two corresponding 2-colorings $c_V \in \{0,1\}^{|V|}$ and $c_W \in \{0,1\}^{|W|}$. We assume that the vertices of $V \cup W$ are ordered such that every vertex of V has a smaller index than any vertex of W (which is always possible since the vertices in a bag can be renumbered accordingly). Suppose that $c_V = (c_1, \ldots, c_{|V|})$ and $c_W = (c'_1, \ldots, c'_{|W|})$. The *concatenation* $c_V \times c_W \in \{0,1\}^{|V|+|W|}$ of c_v and c_w is defined a 2-coloring

$$(c_1, \ldots, c_{|V|}, c'_1, \ldots, c'_{|W|}).$$

Given a 2-coloring $c \in \{0,1\}^{|V|}$ of V and a color $d \in \{0,1\}$, we write $\#_d(c)$ to denote the number of vertices in V with color d.

Now, using a 2-coloring for each row, the table A_i for node i of the nice tree decomposition with bag $X_i = \{x_1, \ldots, x_{n_i}\}$ looks as follows.

x_1	x_2	$\ldots$	x_{n_i-1}	x_{n_i}	d	m_i
0	0	$\ldots$	0	0	0	
0	0	$\ldots$	0	0	1	
		$\vdots$			$\vdots$	
0	0	$\ldots$	0	0	k	
0	0	$\ldots$	0	1	0	
0	0	$\ldots$	0	1	1	
		$\vdots$			$\vdots$	
1	1	$\ldots$	1	1	$k-1$	
1	1	$\ldots$	1	1	k	

Each row of this table represents a 2-coloring of the vertices in the first n_i columns. An integer value in column "d" denotes the number of cover vertices in $V_i \setminus X_i$. Column m_i is a mapping

$$m_i : \{0,1\}^{n_i} \times \mathbb{N}_0 \to \mathbb{N}_0 \cup \{-\infty\}$$

and stores a maximum number of covered edges in G_i for each configuration. The value "$-\infty$" can be interpreted as *invalid*, which means that the subgraph $G[V_i \setminus X_i]$ contains less than d vertices and thus, there cannot be any cover with exactly d vertices from $V_i \setminus X_i$.

The first step is to describe how to compute the tables for the leaf nodes (initialization step).

Initialization step: For each leaf node i assume that $X_i = \{x\}$. Table A_i is computed as follows. For each coloring $c \in \{0,1\}$ of vertex x and each number $d \in \{0, \ldots, k\}$ set

$$m_i(c,d) := \begin{cases} 0, & \text{if } d = 0 \\ -\infty, & \text{otherwise} \end{cases}$$

This assignment is correct since leaf nodes have no children, i.e., $X_i = V_i$, and since there are no edges in $G[X_i]$ (X_i only contains one vertex).

After this initialization step we compute the tables for the inner nodes in the updating process.

Updating process: We state for each node type how to compute table A_i for an inner node i.

FORGET NODES: Let i be a forget node with child j. Assume that $X_i = \{x_1, \ldots, x_{n_i}\}$ and $X_j = \{x_1, \ldots, x_{n_i}, x\}$.

Loosely speaking, we have to decide for each configuration in A_i (which represents a partial vertex cover on G_i) whether x should be a cover vertex or not such that the corresponding table entry is maximal. For each of these two cases (x in the cover/not in the cover) we retrieve both corresponding entries of table A_j to compute a maximum value.

Formally, we compute

$$m_i(c, d) := \max\{m_j(c \times \{0\}, d), m_j(c \times \{1\}, d-1)\}$$

for each 2-coloring $c \in \{0,1\}^{n_i}$ and each $d \in \{0, \ldots, k\}$. This computation is correct, since the two partial vertex covers on G_j represented by $(c \times \{0\}, d)$ and $(c \times \{1\}, d-1)$ are the only candidates for the partial vertex cover on G_i represented by row (c, d) due to the following reasons. Clearly, the vertices in X_i have to be colored equally in A_i and A_j, so we have to fix the coloring c. Concerning the value of d and the color of x observe the following: If x is not a cover vertex, then the number of cover vertices in $V_i \setminus X_i$ is the same as the number of cover vertices in $V_j \setminus X_j$, so we retrieve row $(c \times \{0\}, d)$. If the vertex x is a cover vertex, then we have one more cover vertex in $V_i \setminus X_i$ as compared to $V_j \setminus X_j$, so we retrieve row $(c \times \{1\}, d-1)$. We take the maximum over both corresponding entries of table A_j, since we require that the new entry of table A_i in row (c, d) represents a partial vertex cover on G_i with a maximum number of covered edges. See Figure 4.4 for an example of this step showing the computation of a row in A_i. Note that we only set $m_i(c, d) := -\infty$ if both $m_j(c \times \{0\}, d)$ and $m_j(c \times \{1\}, d-1)$ equal "$-\infty$". (We assume that "$-\infty$" is the smallest element.)

INTRODUCE NODES: Let i be an introduce node with child j. Assume that $X_i = \{x_1, \ldots, x_{n_j}, x\}$ and $X_j = \{x_1, \ldots, x_{n_j}\}$.

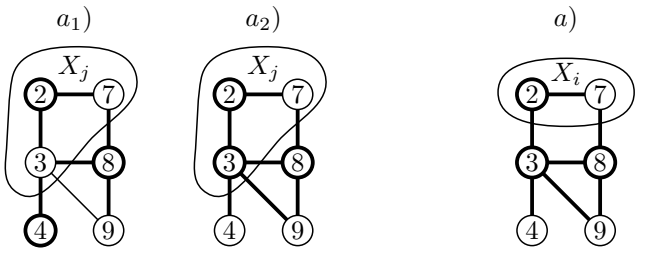
Loosely speaking, for each configuration in A_i, which represents a partial vertex cover C on G_i, we have to look up the maximum number of edges covered by C on G_j using table A_j, adding the edges that are additionally covered in G_i due to vertex x.

Formally, for each 2-coloring $c \in \{0,1\}^{n_j}$ and each 2-coloring $a \in \{0,1\}$ of vertex x, we compute the number n_a of covered edges in $G[X_i]$ with x as one endpoint, and we compute for each $d \in \{0, \ldots, k\}$

$$m_i(c \times a, d) := m_j(c, d) + n_a.$$

The correctness of this computation can be seen considering the following.

- The partial vertex cover on G_j represented by row (c, d) in A_j is the only candidate for the partial vertex cover on G_i represented by



A_j

	2	3	7	d	m_j
	0	0	0	0	0
		⋮			
$a_1)$	1	0	0	2	6
		⋮			
$a_2)$	1	1	0	1	7
		⋮			
	1	1	1	2	7

A_i

	2	7	d	m_i
	0	0	0	0
		⋮		
$a)$	1	0	2	7
		⋮		
	1	1	2	7

Figure 4.4: Example of the dynamic programming step for PVC in the case of a forget node i with child j. The interesting rows are denoted by letters. For each letter we show the corresponding graph at the top of the figure. To compute row a we have to choose among rows a_1 and a_2. The rows a_1 and a_2 both represent solutions with two cover vertices in $V_i \setminus X_i$. Here, row a_2 represents a solution covering more edges, so we choose value "7" of row a_2.

row $(c \times a, d)$ in A_i due to the following reasons: Clearly, the vertices of X_j have to be colored equally in A_i and A_j, so we can fix the coloring c. Since $V_i \setminus X_i = V_j \setminus X_j$, the number d of cover vertices has to be the same in the corresponding rows of table A_i and A_j.

- The value n_a has to be added to $m_j(c, d)$, since the edges with x as one endpoint covered by the partial vertex cover on G_i represented by row $(c \times a, d)$ in A_i are not counted in $m_j(c, d)$.

JOIN NODES: Let i be a join node with children j and l. Assume that $X_i = X_j = X_l = \{x_1, \ldots, x_{n_i}\}$.

Loosely speaking, since $X_i = X_j = X_l$, we only have to select for each coloring $c \in \{0, 1\}^{n_i}$ of X_i the best pair of table entries of table A_j and A_l with the same 2-coloring, satisfying the constraint that exactly d vertices of the subgraph $G[V_i \setminus X_i]$ are in the cover.

Formally, for each 2-coloring $c \in \{0, 1\}^{n_i}$ of X_i and each $d \in \{0, \ldots, k\}$ we compute an interim value, which is the number of edges covered by a partial

vertex cover represented by (c, d), but counting the covered edges in $G[X_i]$ twice. This interim value is computed by

$$m'_i(c, d) := \max_{0 \leq d_j \leq d} (m_j(c, d_j) + m_l(c, d - d_j)).$$

Since covered edges in $G[X_i]$ are counted both in $m_j(c, d_j)$ and $m_l(d - d_j)$, we have to subtract their number in order to avoid counting them twice. For that reason we compute

$$n_c := \text{Number of edges in } G[X_i] \text{ covered, assuming 2-coloring } c,$$

and we get the correct maximum number of covered edges of a partial vertex cover on G_i represented by row (c, d) by computing

$$m_i(c, d) := m'_i(c, d) - n_c.$$

For the correctness of this computation observe the following: Obviously, the vertices of $X_i = X_j = X_l$ have to be colored equally in A_i, A_j, and A_l, so c has to be fixed. Concerning the value d, it is clear that, in order to obtain a partial vertex cover on G_i with exactly d cover vertices in $V_i \setminus X_i$, we have to combine partial vertex covers on G_j and G_l, such that the number of cover vertices in $V_j \setminus X_j$ and $V_l \setminus X_l$ is exactly d in total. By trying all possible combinations for $0 \leq d_j \leq d$ we find a partial vertex cover on G_i covering a maximum number of edges.

Combinations of rows (c, d_j) and $(c, d - d_j)$ are only used if both entries are valid, otherwise the sum $m_j(c, d_j) + m_l(c, d - d_j)$ equals "$-\infty$". If all possible combinations of rows (c, d_j) and $(c, d - d_j)$ are invalid, then there is no possibility to have exactly d vertices of $G[V_i \setminus X_i]$ in the cover, so $m_i(c, d)$ is set to "$-\infty$" (see also the description for forget nodes).

Solution: Let r be the root node of the nice tree decomposition. To obtain the solution, we look at every possible configuration in table A_r and verify if it represents a partial vertex cover that covers at least t edges with at most k cover vertices. If there is no such configuration, then the algorithms returns "NO-Instance", else, it returns "YES-Instance". In other words, we retrieve all rows (c, d) in A_r, and for every row we compute the number of cover vertices of the corresponding cover by counting the number $\#_1(c)$ of cover vertices in X_r and adding the number of cover vertices in $V \setminus X_r$. Additionally we check the constraint that the number of covered edges must be at least t.

Formally, we compute

$$S := \{m_r(c, d) : c \in \{0, 1\}^{n_r} \ \wedge \ \#_1(c) + d \leq k \ \wedge \ m_r(c, d) \geq t\}$$

and return "YES-Instance" if $S \neq \emptyset$ and "NO-Instance" otherwise. This concludes the description of our algorithm.

4.3.2 Analysis

The correctness of the algorithm in Section 4.3.1 follows from the correctness of each step in the dynamic programming on the nice tree decomposition. The running time is stated in the following.

Theorem 4.3.1. *The algorithm in Section 4.3.1 solves* Partial Vertex Cover *on a given graph G with nice tree decomposition $(\{X_i : i \in I\}, T)$ in $O(2^\omega \cdot k \cdot (\omega^2 + k) \cdot |I|)$ time, where $|I|$ is the number of nodes of the nice tree decomposition and ω the treewidth.*

Proof. Each table has $O(2^\omega \cdot k)$ rows and $O(\omega)$ columns. Note that each row can be accessed in constant time, since the position of a row can be easily computed when coloring c and value d are given (for instance using indirect addressing). Thus for each node type we have the following running times.

Leaf node: The table for each leaf node can be computed in $O(2^\omega \cdot k)$ time since for each row (c, d) we just have to set the value for $m_i(c, d)$ depending on d.

Forget node: The table for each forget node can be computed in $O(2^\omega \cdot k)$ time, since for each row we have to look up exactly two rows of the child table.

Introduce node: The table for each introduce node can be computed in time $O(2^\omega \cdot k \cdot \omega)$, since for each row we have to determine the number of covered edges in $G[X_i]$ incident to the new vertex which is bounded from above by ω.

Join node: The table for a join node can be computed in $O(2^\omega \cdot k \cdot (\omega^2 + k))$ time, since for each row of table A_i we have to search the best combination of rows of tables A_j and A_l, which takes $O(k)$ time, and we have to subtract the number of covered edges counted twice which is bounded from above by ω^2.

Thus, the worst-case running time is $O(2^\omega \cdot k \cdot (\omega^2 + k) \cdot |I|)$. This concludes the proof of Theorem 4.3.1. □

Theorem 4.3.1 shows that the Partial Vertex Cover problem is fixed-parameter tractable when the problem is parameterized by the treewidth of the input graph.

In the next section we address the Connected Vertex Cover problem using also dynamic programming on a nice tree decomposition.

4.4 Connected Vertex Cover

In this section we show that Connected Vertex Cover is fixed-parameter tractable with respect to treewidth as parameter. The same parameterization has been considered for Connected Dominating Set by Demaine and Hajiaghayi [DH05]. We first give an introduction to the general idea that leads to our approach. After that, we give an algorithm to solve ConVC using dynamic programming on a nice tree decomposition in Section 4.4.2. In Section 4.4.3 we show that this algorithm runs in $O(2^{\omega} \cdot \omega^{3\omega+2} \cdot |I|)$ time, where ω and $|I|$ denote the treewidth and the number of nodes of the nice tree decomposition, respectively. This means that ConVC is fixed-parameter tractable when parameterized by the treewidth of the input graph.

4.4.1 The Basic Idea

We use the concept of dynamic programming on tree decompositions as introduced in Section 4.2. Recall that in the example for Vertex Cover we used tables A_i for each node i, where each row corresponds to a vertex cover on the subgraph G_i. How do we have to modify the tables such that its entries can represent a connected vertex cover? The idea is to annotate for every cover vertex in a bag X_i to which other cover vertices in the bag X_i it is connected by a path of cover vertices in the subgraph G_i. Loosely speaking, this enables us to know which cover vertices form connected components, and which configurations of the root table represent a connected cover. To denote which vertices are in the same connected component, we introduce *groups* of vertices in a bag. Cover vertices in X_i are in the same group only if they are connected via paths of cover vertices in G_i. Non-cover vertices are in an arbitrary group. Then, after the dynamic programming, we look up all configurations in the root table that represent a connected vertex cover, i.e., that all cover vertices in the root bag are in the same group.

We will use this concept of groups in the following description of the algorithm.

4.4.2 The Algorithm

Given is an instance of Connected Vertex Cover, that is, a graph G and a integer $k \geq 0$, and a nice tree decomposition $(\{X_i : i \in I\}, T)$ for G of

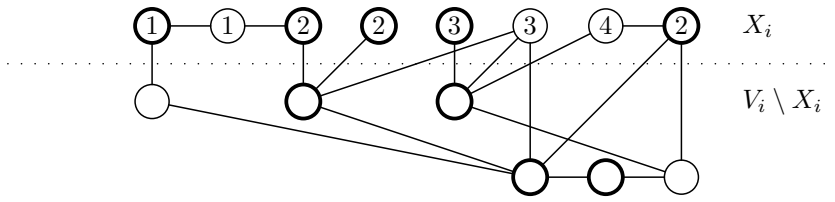


Figure 4.5: This example shows a graph G_i associated with a bag X_i. Cover vertices have a thicker boundary. The number in each vertex $v \in X_i$ is the value of $g(v)$, such that g is a correct group coloring.

width ω. We give a dynamic programming algorithm which computes the size k' of a minimum connected vertex cover. If $k' \leq k$, the algorithm returns that the instance is a "YES-Instance"; otherwise, it returns "NO-instance".

As in Section 4.3 each node i of the *nice* tree decomposition is assigned the graph $G_i = G[\bigcup_{j \in T[i]} X_j]$ where $T[i]$ denotes the subtree rooted at node i. Like in the example for Vertex Cover in Section 4.2 we use tables to describe our algorithm.

Table description: We define for each node i of the nice tree decomposition a table A_i in which rows correspond to all possible configurations of whether or not a vertex X_i is a cover vertex, and group membership.

To describe the tables in more detail we reuse the concept of 2-coloring (Definition 4.3.1). However, we also need to describe the groups to which the vertices of a bag belong. For this reason we introduce *group colorings*.

Definition 4.4.1. *Each vertex of a bag X_i is assigned a* group color*, where a group color is an element of $\{1, \ldots, \omega\}$. Vertices with the same group color belong to the same group. Suppose that $X_i = \{v_1, \ldots, v_{n_i}\}$, and assume that the vertices are ordered by their indices. A* group coloring *g for X_i is a vector*

$$g = (g_1, \ldots, g_{n_i}) \in \{1, \ldots, \omega\}^{n_i},$$

such that each vertex v_j is member of group g_j for all $1 \leq j \leq n_i$. We write $g(v_j)$ to denote the group color of vertex v_j in the group coloring g.

If a group coloring complies with the requirement that cover vertices are in the same group only if there exists a path of cover vertices in G_i connecting them, then we call it *correct*, otherwise we call it *incorrect*. See Figure 4.5 for an example of a correct group coloring. Without defining it explicitly, we also use a *concatenation* of group-colorings, which is defined analogously as for 2-colorings (see Section 4.3).

Now, using a 2-coloring and a group coloring for each row, the table A_i for node i of the nice tree decomposition with bag $X_i = \{x_1, \ldots, x_{n_i}\}$ looks as follows.

x_1	x_2	$\ldots$	$x_{n_{i-1}}$	x_{n_i}	m_i
$(0,1)$	$(0,1)$	$\ldots$	$(0,1)$	$(0,1)$	
$(0,1)$	$(0,1)$	$\ldots$	$(0,1)$	$(0,2)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$(0,1)$	$(0,1)$	$\ldots$	$(0,1)$	$(0,\omega)$	
$(0,1)$	$(0,1)$	$\ldots$	$(0,2)$	$(0,1)$	
$(0,1)$	$(0,1)$	$\ldots$	$(0,2)$	$(0,2)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$(0,\omega)$	$(0,\omega)$	$\ldots$	$(0,\omega)$	$(0,\omega)$	
$(1,1)$	$(1,1)$	$\ldots$	$(1,1)$	$(1,1)$	
$(1,1)$	$(1,1)$	$\ldots$	$(1,1)$	$(1,2)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$(1,\omega)$	$(1,\omega)$	$\ldots$	$(1,\omega)$	$(1,\omega)$	

Each row represents a 2-coloring c and a group coloring g and is denoted by (c, g), where we have a label $(c(x_i), g(x_i))$ in column x_i. The last column m_i is a mapping

$$m_i : \{0,1\}^{n_i} \times \{1, \ldots, \omega\}^{n_i} \to \mathbb{N}_0 \cup \{+\infty\}$$

which returns for each row (c, g) how many cover vertices are needed for a vertex cover on G_i under the restriction. A row (c, d) of A_i can describe a forbidden situation, e.g., that the coloring c leaves uncovered edges, or that the group coloring g defines a group in which two vertices are not connected by a path like described above. In this case the value of $m_i(c, d)$ is set to "$+\infty$".

This concludes the description of the tables, the next step is to describe the initialization of the tables for the leaf nodes of the nice tree decomposition.

Table initialization: Table A_i for each leaf node i of the nice tree decomposition is computed as follows. We assume $X_i = \{x\}$. For each row (c, g) set

$$m_i(c, g) := c(x).$$

All possible combinations of 2-colorings and group colorings are valid since there are no edges in $G[\{x\}]$ and since i has no children, i.e., $X_i = V_i$. So we just have to count x if it is a cover vertex.

After this initialization step we compute the tables for the inner nodes.

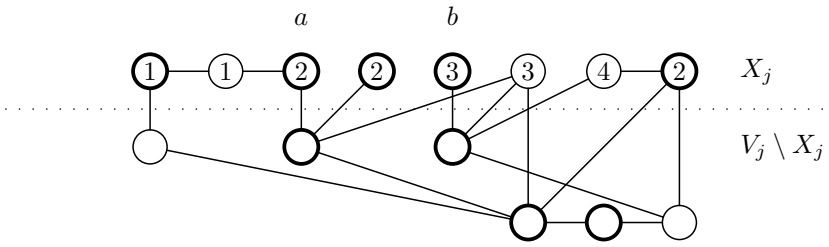


Figure 4.6: Example of a graph G_j and the corresponding bag X_j. The labeling is as in Figure 4.5. Suppose that bag X_i does not contain vertex a. In this case there still is a chance to obtain a connected vertex cover since there remain cover vertices of the same group in X_i. However, suppose that bag X_i does not contain vertex b; then it would be impossible to obtain a connected vertex cover in a later step in any circumstance.

Updating process: The remaining tables are computed in a bottom-up manner from the leaves to the root depending on the underlying node type.

FORGET NODES: Let i denote a forget node with child j. Assume that $X_i = \{x_1, \ldots, x_{n_i}\}$ and $X_j = \{x_1, \ldots, x_{n_i}, x\}$.

Informally speaking, the idea is to verify whether each configuration in A_i represents a vertex cover such that every cover vertex in $V_i \setminus X_i$ is connected to a cover vertex in X_i by a path of cover vertices. If vertex x is a cover vertex, then there must be a cover vertex in X_i with the same group color; otherwise, a connected component of the vertex cover would separate from $G[X_i]$ and thus a connected vertex cover would not be possible anymore. Figure 4.6 gives an example of a vertex which would cause this conflict.

Formally, for each possible 2-coloring $c \in \{0,1\}^{n_i}$ and each possible group coloring $g \in \{1, \ldots, \omega\}$ we compute $m_i(c, g)$ as follows: For each color $c_x \in \{0,1\}$ of vertex x and each group color $g_x \in \{1, \ldots, \omega\}$ of vertex x we verify for row $(c \times c_x, g \times g_x)$ in A_j the condition that if x is a cover vertex, i.e., $c_x(x) = 1$, then there has to exist a cover vertex $v \in X_i$ with $g(v) = g_x(x)$.[2]

Let A be the set of rows $(c \times c_x, g \times g_x)$ complying with this condition. We set

$$m_i(c, g) := \begin{cases} \min_{(c \times c_x, g \times g_x) \in A} \{m_j(c \times c_x, g \times g_x)\}, & \text{if } A \neq \emptyset \\ +\infty, & \text{if } A = \emptyset. \end{cases}$$

[2]Note that we also would have to verify the condition that if x is the only cover vertex in X_i, then every node between i and the root node must be a forget node. However, we can avoid this condition without loss of generality by permitting leaf bags with more than one vertex.

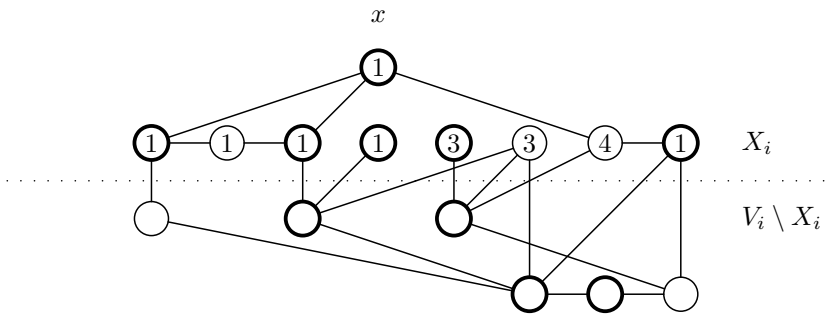


Figure 4.7: Example of a graph G_i. We use Figure 4.6 as a basis. Vertex x causes two connected components of cover vertices in $G[V_i \setminus \{x\}]$ to be a part of one connected component of cover vertices in $G[V_i]$.

This computation is correct, as the vertex cover represented by row (c, g) in A_i only can be one of the vertex covers represented by rows $(c \times c_x, g \times g_x)$ in A_j due to the following reasons. The vertices in X_i have to be 2-colored equally in A_i and A_j and they have to be in the same group. Concerning the 2-color and the group of x, we try all combinations to color x and to assign it to a group, where we verify the condition that x is not the only cover vertex in its group for each combination. We take the minimum over all corresponding entries of table A_j, since we require a vertex cover on G_i with a minimum number of cover vertices.

INTRODUCE NODES: Let i denote an introduce node with child j. Assume that $X_i = \{x_1, \ldots, x_{n_j}, x\}$ and $X_j = \{x_1, \ldots, x_{n_j}\}$.

Informally, the idea is that connected components of a vertex cover are possibly merged by vertex x, if x is a cover vertex. If x is not a cover vertex, then we have to assure that every edge incident to x is covered.

Formally, for each possible coloring $c \in \{0, 1\}^{n_j}$, each coloring $c_x \in \{0, 1\}$ of vertex x, each group coloring $g \in \{1, \ldots, \omega\}^{n_j}$, and each group coloring $g_x \in \{1, \ldots, \omega\}$ of vertex x we compute $m_i(c \times c_x, g \times g_x)$ as follows. We differentiate between the two cases whether x is a cover vertex or not.

1. Vertex x is a cover vertex, i.e., $c_x(x) = 1$. The groups of cover vertices that are adjacent to x are merged. Figure 4.7 gives an example of connected components of cover vertices in $G[X_j]$ that are connected to each other by a path of cover vertices containing x in $G[X_i]$. To compute $m_i(c \times c_x, g \times g_x)$ we search for rows (c, g') in table A_j such that:

   - All groups defined by g' which contain cover vertices that are adjacent to x have the same group color in g like x in g_x. Vertex x

and these groups have a group color that is used by one of these groups in g'.

- All groups defined by g' which do *not* contain cover vertices that are adjacent to x do have the group color in g they have in g'.
- If there do not exist edges between x and vertices in X_i, then x has a group color in g not used by any other cover vertex in X_i in group coloring g.

Let A be the set of rows (c, g') complying with these conditions. We set

$$m_i(c \times c_x, g \times g_x) := \begin{cases} \min_{(c,g') \in A}\{m_j(c, g')\}, & \text{if } A \neq \emptyset \\ +\infty, & \text{if } A = \emptyset. \end{cases}$$

2. Vertex x is not a cover vertex, i.e., $c(x) = 0$. The row $(c \times c_x, g \times g_x)$ has to satisfy that all vertices $v \in X_i$ adjacent to x are cover vertices, i.e., $c(v) = 1$. If the group coloring g complies with this requirement, then we set
$$m_i(c \times c_x, g \times g_x) := m_j(c, g),$$
else, we set $m_i(c, g) := +\infty$.

JOIN NODES: Let i denote a join node with children j and l. Assume that $X_i = X_j = X_l = \{x_1, \ldots, x_{n_i}\}$.

Suppose different connected components of cover vertices of two vertex covers on G_j and G_l represented by rows in tables A_j and A_l, respectively. The important observation is that these different connected components possibly merge by sharing cover vertices in X_i.

Formally, for each row (c, g) in A_i we search pairs of rows (c, g_j),(c, g_l) in A_j and A_l, respectively, such that the corresponding group colorings g_j and g_l comply with the requirement that g equals a group coloring which is returned by the following algorithm.

1. Initialize a group coloring $g'_j = g_j$.

2. While there exist cover vertices in X_i with different group colors in g'_j, but equal group colors in g_l, merge the corresponding groups in g'_j. We merge the groups by associating the same group color in g'_j to all related vertices, using a group color used by any of these vertices in g'_j.

3. Return g'_j.

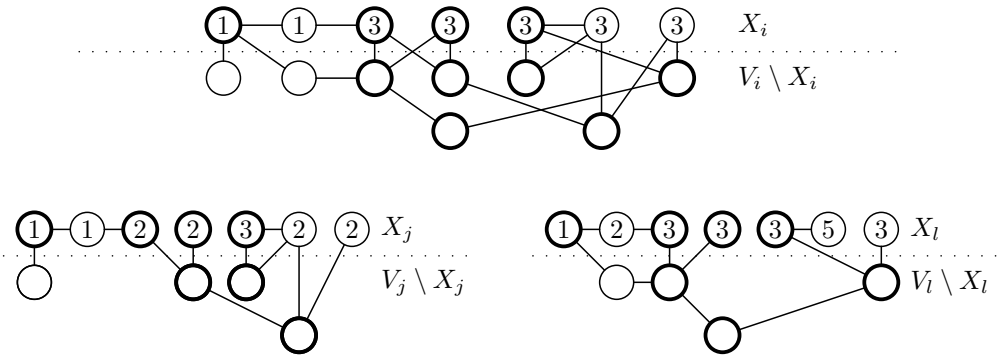


Figure 4.8: Example of a join node i with children j and l. Graph G_j and G_l at the bottom have both a correct group coloring. The tophand graph G_i illustrates how the groups of X_j and X_l merge together to a correct group coloring for X_i.

Figure 4.8 gives an example of how two group colorings merge. This way two cover vertices $u, v \in X_i$ are in the same group in g only if there exists a path of cover vertices in G_i connecting u and v. Among these pairs of rows (c, g_j),(c, g_l) we search a pair such that $m_j(h_j) + m_l(h_l)$ is minimal. If there exists such a pair, then the value of $m_i(c, g)$ can be computed as

$$m_i(c, g) := m_j(h_j) + m_l(h_l) - \#_1(c).$$

(We subtract the number $\#_1(c)$ of vertices in X_i which are part of the cover since they are counted in both $m_j(h_j)$ and $m_l(h_l)$.) If there exist no such pairs (c, g_j),(c, g_l) then $m_i(c, g)$ equals "$+\infty$".

Solution: Let r be the root node of the nice tree decomposition. We have to choose a row of A_r describing a group coloring such that all cover vertices of bag X_r are in the same group.

Rows (c, g) describing group colorings g for X_i which assign to all cover vertices the same group color are candidates to give the size $m_i(c, g)$ of a minimum connected vertex cover. Among them, we choose a row (c, g) with minimum $m_i(c, g)$. If $m_i(c, g) \leq k$, then the algorithm returns "YES-Instance"; otherwise, it returns "NO-Instance".

This concludes the description of our algorithm. We proceed with its analysis to show that Connected Vertex Cover is fixed-parameter tractable when parameterized by the treewidth.

4.4.3 Analysis

The correctness of the algorithm in Section 4.4.2 follows from the correctness of each step in the dynamic programming on the nice tree decomposition. The running time is stated in the following.

Theorem 4.4.1. *The algorithm in Section 4.4.2 solves* ConVC *on a given graph G with nice tree decomposition $(\{X_i\}, T)$ in $O(2^\omega \cdot \omega^{3\omega+2} \cdot |I|)$ time, where $|I|$ is the number of nodes of the nice tree decomposition and ω the treewidth.*

Proof. Each table has $O(\omega)$ columns and $O(2^\omega \cdot \omega^\omega)$ rows, since the maximum size of each bag is ω. A table row can be accessed in constant time using for instance indirect addressing. For each node type we give the running time needed to compute the entire corresponding table.

Leaf node: For each row we can compute m_i in $O(1)$ time, so the running time for a table is $O(2^\omega \cdot \omega^\omega)$.

Forget node: The running time is $O(2^\omega \cdot \omega^\omega \cdot \omega)$, since for each entry of table A_i we have to look at 2ω entries of table A_j.

Introduce node: The running time is $O(2^\omega \cdot \omega^{2\omega+1})$. For each row of table A_i we have to look at $O(\omega^\omega)$ rows of A_j, since the coloring is determined. For each row of table A_j we have to check the given conditions, and this check can be performed in $O(\omega)$ time.

Join node: Let i denote a join node with children j and l. The running time is $O(2^\omega \cdot \omega^{3\omega+2})$. For each row of table A_i we have to look at $O(\omega^\omega)$ rows of A_j, and for each of these rows we observe $O(\omega^\omega)$ rows of A_l (the coloring is determined). For each pair of rows of A_j and A_l we have to check if they fulfill the given conditions, which takes $O(\omega^2)$ time since we have to check for all vertex pairs in X_i if their corresponding groups can be merged.

Thus, the worst-case estimation of the running time is $O(2^\omega \cdot \omega^{3\omega+2} \cdot |I|)$. This concludes the proof of Theorem 4.4.1. □

4.4.4 How to Improve the Running Time

The tables A_i used for the dynamic programming algorithm in Section 4.4.2 use group colorings to describe partitions of the vertex set. Observe that many group colorings describe the same partition. This can be improved by assigning to each vertex the lowest possible group color when processing

from the left to the right in the order given by the table. In all steps of the algorithm we then have to satisfy this property by re-coloring the groups of each partition. This can be done in $O(\omega)$ time. We did not state the algorithm this way since it would complicate its description.

If we apply the requirement that the group colorings have to describe the partitions in a unique way, then each table for the dynamic programming has $O(2^\omega \cdot B_\omega)$ instead of $O(2^\omega \cdot \omega^\omega)$ rows, where B_n is the bell number. The bell number satisfies $B_n = O(n!)$. The algorithm thus can be improved to run in $O(2^\omega \cdot (\omega!)^3 \cdot \omega^2 \cdot |I|)$ time. Note that methods to efficiently generate all partitions of a given set exist (see, e.g. [Knu05]).

In the next section we address the Capacitated Vertex Cover using a similar dynamic programming technique.

4.5 Capacitated Vertex Cover

In this section we address Capacitated Vertex Cover. We first state an algorithm that solves the problem in polynomial time for graphs with treewidth two, namely series-parallel graphs, then we sketch how this approach can be generalized to graphs with treewidth ω. This approach also shows that CVC is fixed-parameter tractable with treewidth as a parameter for graph classes with bounded vertex degree. However, the properties of CVC concerning fixed-parameter tractability with treewidth as a parameter in the general case remain open.

4.5.1 Series-Parallel Graphs

In this subsection we give an introduction to series-parallel graphs and sp-trees. The sp-trees of series-parallel graphs were used to design dynamic programming algorithms which run in polynomial time for combinatorial problems on series-parallel graphs, see e.g., Takamizawa et al. [TNS82]. These results were generalized to graphs with bounded treewidth using a similar dynamic programming technique to achieve efficient algorithms [AP89]. We take on this approach; first we show that CVC is efficiently solvable for series-parallel graphs, then we explain how to generalize this method to graphs with treewidth ω.

Definition 4.5.1. (Series-Parallel Graph)
A two-terminal labeled graph *is a triple* (G, a, b)*, where* $G = (V, E)$ *is a graph and* $a, b \in V$ *are the* terminals*. A* series-parallel graph *is a two-terminal labeled graph defined recursively as follows.*

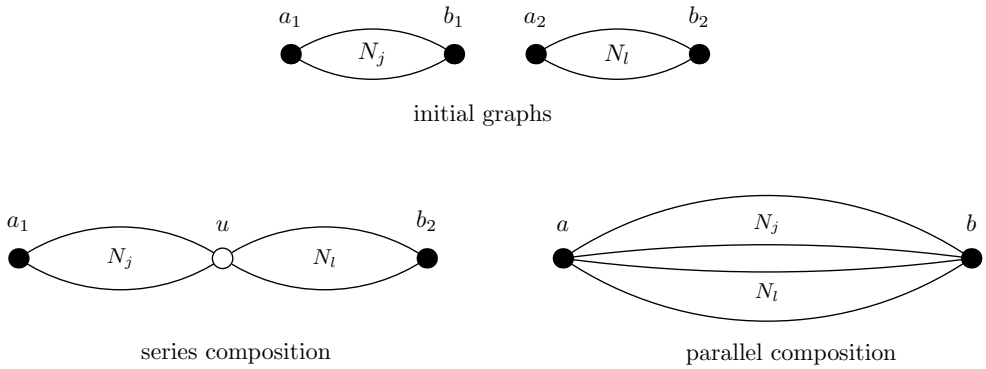


Figure 4.9: A schematic diagram of the series and parallel compositions.

1. *Two vertices u and v connected by an edge constitute a series-parallel graph. We call this the basic series-parallel graph.*

2. *Given two series-parallel graphs $((V_1, E_1), a_1, b_1)$ and $((V_2, E_2), a_2, b_2)$, the following two graphs are series-parallel:*

   (a) *Series composition:*
   The two-terminal labeled graph $((V_1 \cup V_2, E_1 \cup E_2), a_1, b_2)$, where the vertices b_1 and a_2 are replaced by a new vertex u such that u is adjacent to all neighbors of b_1 and a_2, is series-parallel.

   (b) *Parallel composition:*
   The two-terminal labeled graph $((V_1 \cup V_2, E_1 \cup E_2), a, b)$ is series-parallel, where the vertices a_1, a_2 and b_1, b_2 are replaced by two new vertices a and b, respectively, such that a is adjacent to all neighbors of a_1 and a_2, and b is adjacent to all neighbors of b_1 and b_2.

In Figure 4.9 we sketch the series and parallel composition. The structure of a series-parallel graph can be described by an *sp-tree*. The notion of sp-tree is related to tree decomposition, as sp-trees describe the underlying graph structure with a similar decomposition tree. We apply dynamic programming on such an sp-tree to solve CVC on series-parallel graphs in Section 4.5.2.

Definition 4.5.2. (SP-Tree)
An sp-tree $T_{(G,a,b)}$ *for a series-parallel graph (G, a, b) is a rooted tree in which each node N_i has a type and an ordered pair of vertices (u, v) as label. There are three different possible types, namely* leaf-node, p-node, *and* s-node. *Every node N_i corresponds to a series-parallel graph (G_i, a_i, b_i), where G_i is a subgraph of G and (a_i, b_i) is the label of the node, such that:*

- *If N_i is a leaf-node, then (G_i, a_i, b_i) is a basic series-parallel graph. Each edge in G is represented by a leaf-node.*

- *If N_i is an s-node with children N_j and N_l, then (G_i, a_i, b_i) is the series-parallel graph which results by applying the series composition to the two-terminal labeled graphs (G_j, a_j, b_j) and (G_l, a_l, b_l) represented by the nodes N_j and N_l in the sp-tree.*

- *If N_i is a p-node with children N_j and N_l, then (G_i, a_i, b_i) is the two-terminal labeled graph which results by applying the parallel composition to the two-terminal labeled graphs represented by node N_j and N_l.*

- *The root of the tree has label (a, b) and corresponds to the graph (G, a, b).*

The class of series-parallel graphs can be recognized in linear time in the input size [Sch95, VTL82]. Series-parallel graphs are exactly the so-called partial 2-trees [BLS99]. Partial 2-trees in turn are exactly the graphs with treewidth two, so all graphs with treewidth two are series-parallel [BLS99]. This concludes the description of series-parallel graphs and sp-trees. For a more detailed introduction to series-parallel graphs we refer to [BLS99]. In the next section we will apply dynamic programming on sp-trees to solve CVC on series-parallel graphs.

4.5.2 Dynamic Programming on SP-Trees

For an sp-tree $T_{(G,a,b)}$ of a series-parallel graph (G, a, b) with $G = (V, E)$, a capacity function $c : V \to \mathbb{N}^+$, and a weight function $w : V \to \mathbb{R}^+$, we can solve CVC using dynamic programming on $T_{(G,a,b)}$.

The main idea is to observe how much capacity remains unused for each of the terminal vertices of each terminal graph in $T_{(G,a,b)}$ (see Definition 4.5.2). To describe such a distribution of unused capacity we introduce the notion of r-capacity.

An *r-capacity* ("remaining capacity") for a node N_i with two-terminal labeled graph (G_i, a_i, b_i) is a vector $(x_i, y_i) \in {\mathbb{N}_0}^2$. It is used to express the numbers x_i and y_i of edges that can still be covered by each of the vertices a_i and b_i, respectively. This means that $c(a_i) - x_i$ edges in G_i are covered by a_i, and $c(b_i) - y_i$ edges by b_i. The r-capacity determines whether or not the vertices a_i and b_i are a part of the cover: If the r-capacity of a terminal vertex equals the total capacity, i.e., $x_i = c(a_i)$, then the respective vertex a_i is not a part of the cover, whereas it is a part of the cover in all other cases. For each node N_i with terminal graph (G_i, a_i, b_i) there exist $(c(a_i) + 1)(c(b_i) + 1)$ possible r-capacities.

As in Sections 4.3 and 4.4 we use tables to store information about solutions on subgraphs of G, and we apply a similar description scheme. The algorithm works in a bottom-up manner. First we compute the tables for the leaf nodes, then the remaining nodes are processed from the bottom to the top until we reach the root node.

Table description: For each node N_i with terminal graph (G_i, a_i, b_i) we maintain a table A_i:

a_i	b_i	w_i
0	0	
0	1	
⋮	⋮	
0	$c(b_i)$	
1	0	
⋮	⋮	
$c(a_i)$	$c(b_i)$	

This table has $(c(a_i) + 1)(c(b_i) + 1)$ rows. The first two values x_i and y_i of each row correspond to the r-capacity (x_i, y_i). The last column w_i stores the weight of the appropriate cover. It is defined as a function depending on the r-capacity (x_i, y_i) and returns the weight of a minimum cover on G_i such that a_i covers $c(a_i) - x_i$ edges of G_i, and b_i covers $c(b_i) - y_i$ edges. The value $w_i(x_i, y_i)$ equals "$+\infty$" if the r-capacity (x_i, y_i) describes a distribution of the capacities of a_i and b_i such that a capacitated vertex cover for G_i is not possible. After the initialization the values of $w_i(x_i, y_i)$ are computed for each table, beginning at the leaf nodes.

Table initialization: Let (G_i, a_i, b_i) be a two-terminal labeled graph of a leaf node N_i, i.e., G_i consists of one edge $\{a_i, b_i\}$.

For each leaf node N_i we have to cover $\{a_i, b_i\}$. Therefore, we have to assure that at least one vertex a_i or b_i is a part of the cover. The value w_i is computed for each row depending on whether a_i or b_i is chosen. Let (x_i, y_i) denote the r-capacity of each row of table A_i.

Formally, we set for each row

$$w_i(r) := \begin{cases} w(a_i), & \text{if } y_i = c(b_i) \ \wedge \ x_i = c(a_i) - 1 \\ w(b_i), & \text{if } x_i = c(a_i) \ \wedge \ y_i = c(b_i) - 1 \\ +\infty, & \text{otherwise} \end{cases} \tag{4.1}$$

The next step is to compute the tables for the remaining nodes.

Updating process: We state how to compute the tables for non-leaf nodes for each node type (s-nodes and p-nodes).

S-NODES: Let (G_i, a_i, b_i) be a two-terminal labeled graph of an s-node N_i with children N_j and N_l. The corresponding two-terminal labeled graphs of N_j and N_l are (G_j, a_i, u) and (G_l, u, b_i), respectively. Let (x_i, y_i) denote the r-capacity for each row of table A_i.

Informally, the two-terminal labeled graph (G_i, a_i, b_i) is composed by attaching the two-terminal labeled graphs (G_j, a_i, u) and (G_l, u, b_i) at vertex u. For each r-capacity (x_i, y_i) we have to find the best distribution of the capacity $c(u)$ such that the cover of terminal graph (G_i, a_i, b_i) has minimum weight. Therefore, we search in tables A_j and A_l for pairs of r-capacities $(x_i, y_j), (x_l, y_i)$ which satisfy the constraint that u covers at most so many edges as its capacity, i.e. $y_j + x_l \leq c(u)$. We then select the pair which leads to a minimum value for w_i.

Values $w_j(x_i, y_j)$ and $w_l(x_l, y_i)$ both may already include the weight $w(u)$, and we have to subtract it to get the correct value of $w_i(x_i, y_i)$. Only if y_j or x_l equals $c(u)$ we do not have to subtract $w(u)$, since it is only counted in either $w_j(x_i, y_j)$ or $w_l(x_l, y_i)$ then. If both y_i and x_l equal $c(u)$, then $w(u)$ is not counted at all since u is not a part of the cover.

To formally describe these facts we define a function $d : {\mathbb{N}_0}^3 \to \{0, 1\}$.

$$d(x_1, x_2, c) := \begin{cases} 0, & \text{if } x_1 = c \ \vee \ x_2 = c \\ 1, & \text{otherwise} \end{cases}$$

This function is used in the following to subtract extra weight which is counted twice. For each row of table A_i we compute

$$w_i(x_i, y_i) := \min_{\substack{0 \leq x_l \leq c(a_i) \\ 0 \leq y_j \leq c(b_i)}} \{w_j(x_i, y_j) + w_l(x_l, y_i) - d(y_j, x_l, c(u)) \cdot w(u) : \\ c(u) - y_j \leq x_l\}$$

This concludes the description of the computation for the s-nodes.

P-NODES: Let (G_i, a_i, b_i) be a two-terminal labeled graph of a p-node N_i with child nodes N_j and N_l, and with corresponding two-terminal labeled graphs (G_j, a_i, b_i) and (G_l, a_i, b_i). Let (x_i, y_i) denote the r-capacity for each row of table A_i.

Here, we attach the two-terminal labeled graphs (G_j, a_i, b_i) and (G_l, a_i, b_i) at vertices a_i and b_i. For each row of table A_i we search pairs of rows of tables A_j and A_l with r-capacities (x_j, y_j) and (x_l, y_l), respectively, such that attaching (G_j, a_i, b_i) and (G_l, a_i, b_i) at vertices a_i and b_i results in a two-terminal labeled graph (G_i, a_i, b_i) having r-capacity (x_i, y_i). More precisely, we have to demand that $x_i = x_j + x_l - c(a_i)$ and $y_i = y_j + y_l - c(b_i)$. For

instance, vertex a_i covers $c(a_i) - x_j$ edges in G_j, and it covers $c(a_i) - x_l$ edges in G_l. Thus it covers $2c(a_i) - x_j - x_l$ edges in G_i. This means that it still can cover $c(a_i) - (2c(a_i) - x_j - x_l) = x_j + x_j - c(a_i)$ edges.

Again we have to subtract weight which is counted twice. This works in the same way as for s-nodes, but applied on both a_i and b_i. We use the function d like defined above, but introduce the following abbreviations to keep the succeeding equation shorter.

$$\delta_a := d(x_j, x_l, c(a_i)),\ \delta_b := d(y_j, y_l, c(b_i)).$$

We compute for each row of table A_i

$$w_i(x_i, y_i) := \min_{\substack{0 \le x_j, x_l \le c(a_i) \\ 0 \le y_j, y_l \le c(b_i)}} \{w_j(x_j, y_j) + w_l(x_l, y_l) - \delta_a \cdot w(a_i) - \delta_b \cdot w(b_i) : \\ x_i = x_j + x_l - c(a_i) \ \wedge \ y_i = y_j + y_l - c(b_i)\}.$$

This concludes the description of the computation for the p-nodes.

Solution: After processing the root node N_r we get the weight of a minimum capacitated vertex cover by searching the row (x_i, y_i) with minimum value $w_i(x_i, y_i)$. This concludes the description of this algorithm.

4.5.3 Time Complexity

The correctness of the algorithm in Section 4.5.2 follows from the correctness of each step in the dynamic programming on the sp-tree. The running time is stated in the following.

Lemma 4.5.1. *The algorithm presented in Section 4.5.2 solves* CVC *on n-vertex series-parallel graphs in $O(n^5)$ time.*

Proof. The tables used to store information about possible solutions for each two-terminal labeled graph (G_i, a_i, b_i) have $(c(a_i) + 1)(c(b_i) + 1)$ rows. The maximum meaningful capacity of each vertex of the input graph is $n - 1$, since there cannot exist more incident edges. If a capacity exceeds $n - 1$, then it can be set to $n - 1$ without affecting the solution.

With this bound on the vertex capacities, we have at most n^2 rows in each table. We can assume that a table row can be accessed in constant time using for instance indirect addressing. The table initialization can be done in $O(n^2)$ time for each table, since for each row of a table we have to apply one out of three cases, which can be done in $O(1)$ time (see Equation (4.1)). To compute table A_i for s-nodes we need $O(n^4)$ time, since for each row of table A_i we have to search for the best solution by examining $O(n^2)$ combinations of rows

in table A_j and table A_l. The computation of the tables for the p-nodes also needs $O(n^4)$ time, since we also have to examine $O(n^2)$ combinations due to the restrictions $x_i = x_j + x_l - c(a_i)$ and $y_i = x_j + x_l - c(b_i)$.

We get the worst-case time of $O(n^5)$ to compute all tables, and the following determination of the result using the root node only needs $O(n^2)$ time. □

In the next section, we generalize this approach to dynamic programming on tree decompositions.

4.5.4 CVC on Tree Decompositions

Here we sketch how to generalize the dynamic program given in Section 4.5.2 to graph classes with treewidth ω.

We stated the algorithm for series-parallel graphs first, as the detailed description of an algorithm for graph classes with treewidth ω would be far more complicated, although the main idea is very similar. Thus, using the idea of our algorithm for series-parallel graphs, we describe the algorithm for graph classes with treewidth ω in a less detailed way.

Let us recall what we actually did to solve CVC on series-parallel graphs. Each node N_i of the given sp-tree is associated with a two-terminal labeled graph (G_i, a_i, b_i). We store the r-capacity of the vertices a_i and b_i to describe possible solutions on G_i. In other words, we store possible solutions for the subgraph (G_i, a_i, b_i) independent from how the solution looks like in the remaining graph.

This approach works in a similar way for graph classes with treewidth ω. Let $(\{X_i : i \in I\}, T)$ be a nice tree decomposition for a given graph G. Recall that each non-leaf bag X_i is a separator of G. We have to characterize possible solutions on G_i in a way such that they are independent from the remaining graph. We adapt the method we applied for series-parallel graphs. For each vertex $v \in X_i$ we store the information about how many edges it still can cover. Again we use a table A_i to store for all vertices in $X_i = \{x_1, \ldots, x_{n_i}\}$ all combinations of free capacities.

x_1	$\ldots$	x_{n_i}	w_i
0	$\ldots$	0	
	$\vdots$		
$c(x_1)$	$\ldots$	$c(x_{n_i})$	

Such a table has $O(n^\omega)$ rows, because the maximal size of a bag is ω and the capacities of all vertices are upper-bounded by n. For each row of a table A_i we store the weight w_i of the appropriate solution for G_i. For each node

type (leaf, introduce, forget, and join) we compute the values of w_i of the appropriate table A_i by using the child tables of A_i: We check for each row of table A_i if the corresponding distribution of the capacities is possible, and if so we compute the weight of the corresponding capacitated vertex cover. In the following we explain the most important aspects of this method.

Table initialization: Each leaf bag contains only one vertex x, so there is no edge to cover. The row denoting that the free capacity of x is $c(x)$ has value "0" in column w_i and describes the only valid situation. In the remaining rows we set value "$+\infty$" in column w_i.

Updating process: The remaining tables are computed in a bottom-up manner from the leaves to the root depending on the underlying node type.

FORGET NODES: For each row r of the corresponding table we look up all rows of the child table which describe the same amount of free capacity for each vertex in the bag, except for vertex x, which is not included in the bag of the forget node. Value w_i of row r is set to a minimal value w_j of all matching rows of the child table.

INTRODUCE NODES: In this step we have to consider that the new vertex x possibly has adjacent vertices which are a part of the introduce node i. The corresponding new edges have to be covered. Similar to our approach for series-parallel graphs, each row r of table A_i describes the free capacity of each vertex in X_i, and we write $r(x)$ to describe how many edges still can be covered by vertex x. For each row r of table A_i we have to search for rows r_j of the child table A_j, such that

- there are $c(x) - r(x)$ vertices in X_j that are adjacent to x and have the same free capacity in r as in r_j,
- all other vertices adjacent to x have one less free capacity in r as in r_j, and
- vertices in X_i not adjacent to x are have the same free capacity in r as in r_j.

The value $w_i(r)$ is computed by searching the cheapest solution. That is, for all adequate rows r_j we search a row r' with minimum value $w_j(r')$. To compute $w_i(r)$, we use $w_j(r')$ and add the weight of vertex x if x is a part of the cover. If there do not exist adequate rows r_j we set $w_i(r) := +\infty$.

JOIN NODES: For each row r of table A_i of a join node we have to search for pairs of rows (r_j, r_l) of the child tables A_j and A_l, respectively, such that r

is the result of merging the free capacities described by r_j and r_l. Here we are actually doing exactly the same as we did for p-nodes for series-parallel graphs. The only difference is the number of vertices involved.

Solution: The solution to the problem can be obtained by searching a row of the root table with a minimum value of w_i.

Analysis: The running time of this algorithm is stated in the following.

Theorem 4.5.1. *The algorithm solves* Capacitated Vertex Cover *on a given graph G with nice tree decomposition $(\{X_i\}, T)$ in $O(n^{2\omega} \cdot \omega \cdot |I|)$ time, where k is the size of the desired capacitated vertex cover, and $|I|$ and ω are the number of nodes and the width of the nice tree decomposition, respectively.*

Proof. The worst-case running time to compute a table is $(n^\omega)^2 \cdot \omega$: In the case of forget nodes we have to compute n^ω table entries, and for each entry we have to look up $O(n^\omega)$ rows and perform a test which takes $O(\omega)$ time for each row. In the case of introduce nodes we have to look up for each table entry at most n^ω entries of the child table, again performing a test which needs $O(\omega)$ time. In the case of join nodes we have to look up at most $2 \cdot n^\omega$ entries of the child tables due to restrictions of type "$x_i = y_j + x_l - c(a_i)$" for each vertex of the bag (see Section 4.5.3). Thus, the total running time of the algorithm is $O(n^{2\omega} \cdot \omega \cdot |I|)$, where $|I|$ is the number of nodes of the nice tree decomposition. □

However, this running time can be improved as follows. Note that, if the degree $\deg(v)$ of a vertex v exceeds k, then v must cover at least $\deg(v) - k$ edges, i.e., $c(v) \geq \deg(v) - k$. Otherwise, more than k edges would be covered by more than k vertices adjacent to v, so the cover would exceed its maximum size k. On the other hand, the maximum meaningful value of $c(v)$ is $\deg(v)$. This means that the meaningful capacity of each node lies withing a range of k.

With this observation, the size of the tables used in the algorithm presented in this section can be reduced from $O(n^\omega)$ to $O(k^\omega)$. Hence, the time to compute a table is $O(k^{2\omega} \cdot \omega)$. That means that for all tables we need $O(k^{2\omega} \cdot \omega \cdot |I|)$ time, where $|I|$ is the number of nodes of the nice tree decomposition.

Theorem 4.5.2. *The algorithm stated in this section can be modified such that it solves* Capacitated Vertex Cover *on a given graph G with nice tree decomposition $(\{X_i\}, T)$ in $O(k^{2\omega} \cdot \omega \cdot |I|)$ time, where k is the size of the desired capacitated vertex cover, and $|I|$ and ω are the number of nodes and the width of the nice tree decomposition, respectively.*

Another observation can be made when we adapt the tables of the algorithm in Section 4.5.4 for graphs with bounded vertex degree.

Corollary 4.5.1. CVC *is fixed-parameter tractable with treewidth as parameter for graph classes with bounded vertex degree.*

Proof. If the vertex degree is bounded by a constant c, then the maximal meaningful free capacity of a vertex is c. In this case, the table size can be reduced from n^ω (see proof of Theorem 4.5.1) to c^ω. This means that the running time can be improved from $O(n^{2\omega} \cdot \omega \cdot |I|)$ (see Theorem 4.5.1) to $O(c^{2\omega} \cdot \omega \cdot |I|)$, which shows that CVC is fixed-parameter tractable with respect to treewidth for graphs with bounded vertex degree. □

The difficulty in the case of general CVC seems to be that the distribution of the vertex capacities of the vertices in X_i affects the solution for G_i; one more unit of capacity available for subgraph G_i might change its cover by a sort of "chain reaction". That is, if a vertex has one more unit of capacity available, then it can cover one more edge in the subgraph, and the corresponding neighbor then again has one more unit of capacity available. In this way a free capacity unit can be passed to any vertex on a path of cover vertices, possibly affecting the weight of the corresponding solution if there is a vertex which can be omitted from the cover by using the free capacity unit.

So in each bag X_i we *have* to fix the number of edges covered in the corresponding subgraph G_i. We store for each vertex the number of edges it still can cover, and unfortunately this number is not bounded by a function depending only on the treewidth. We have $O(n^\omega)$ possible capacity distributions for a bag and so the table would be too big to lead to fixed-parameter tractability with respect to treewidth in the general case.

4.6 Concluding Remarks

Interestingly, Partial Vertex Cover is fixed-parameter tractable with respect to treewidth as parameter, but it is $W[1]$-hard when parameterized by the maximum size of the desired cover (see Section 3). On the other hand, Connected Vertex Cover is fixed-parameter tractable for both parameterizations. We showed that Capacitated Vertex Cover is fixed-parameter tractable with the treewidth as parameter for graphs with bounded vertex degree. However, we do not know if the problem is fixed-parameter tractable with respect to treewidth in the general case. This is an open problem and appears to be an interesting starting point for future work.

In the next chapter we will consider a generalization of CVC restricted to trees. In this case we also encounter the difficulty of "distributing capacities", so this variant of the problem can be used to study this difficulty of CVC more closely under simplified conditions. It is interesting that the running time of the algorithm that solves this variant also depends on the maximum vertex degree.

Chapter 5

Capacitated Vertex Cover on Trees

In this chapter we study the Minimum Capacitated Vertex Cover with Demand (CVCD) problem on trees. This problem is a generalization of Capacitated Vertex Cover and is introduced by Guha et al. [GHKO03]. The additional requirement is that each edge is assigned a *demand* which denotes the number of capacity units needed to cover the edge. Minimum Capacitated Vertex Cover with Demand is NP-complete even when restricted to trees (recall that Capacitated Vertex Cover is polynomial-time solvable on trees). The NP-completeness can be shown by a reduction from the Knapsack problem [GHKO03].

Guha et al. [GHKO03] present several variants of the problem, some of them remain NP-complete while others can be solved in polynomial time, when restricted to trees. We continue their work and give a fixed-parameter algorithm with the maximum vertex degree as parameter for an NP-complete variant of this problem on trees.

This chapter is organized as follows. In Section 5.1 we first give the definition of CVCD restricted to trees and present the different variants of the problem. Thereafter, we give an overview of the known results presented by Guha et al. [GHKO03], including the new results which are derived in Section 5.2.

5.1 Definitions and Preliminaries

In this section we introduce CVCD restricted to trees, explain its variants, and state known and new results.

For the definition of CVCD we need the notion of *multiset*. A multiset is

a collection in which a element is allowed to occur more than once. In the CVCD problem, vertices can be taken several times into the cover, and we use multisets to describe such a cover. Each occurrence of a vertex in the multiset is called a *copy* of the vertex.

Definition 5.1.1. *Given a tree $T = (V, E)$, vertex weights $w : V \to \mathbb{R}^+$, vertex capacities $c : V \to \mathbb{N}^+$, and edge demands $d : E \to \mathbb{N}^+$, a* cvcd-cover *is defined as a multiset V' of vertex copies with the following properties:*

1. *Taking x copies of a vertex into the cvcd-cover causes the vertex to have x-times its original capacity.*

2. *Each edge e has to be covered by at least $d(e)$ capacity units.*

The weight *of a cvcd-cover V' is the sum of the weights of each vertex copy in V'.*

In other words, we have to "pay" the weight $w(v)$ for each copy of a vertex v. The central problem of this chapter is defined as follows.

Definition 5.1.2. (Minimum Capacitated Vertex Cover with Demand on Trees):

Input: *A tree $T = (V, E)$, vertex weights, vertex capacities, and edge demands.*

Output: *A cvcd-cover with minimum weight (minimum cvcd-cover).*

The abbreviation for this problem is CVCDT. We consider several variants of this problem. Variants consist of combinations of constraints on the general case. We list these constraints in the following, where we distinguish between the general case and the restricted case, stating always the general case in first place.

SPLIT/NOSPLIT: We differentiate between the two cases whether or not the demand $d(e)$ of an edge e can be split. In the general case (SPLIT), it is possible that an edge obtains its assigned capacity by copies of both incident vertices, whereas in the restricted case (NOSPLIT) the demand of an edge has to be satisfied by copies of only one of its endpoints. Thus for an edge $e = \{u, v\}$ the amount of capacity units of either u or v assigned to e has to be equal or greater than $d(e)$ (and the other has to equal 0).

Unbounded vertex degree/Bounded vertex degree: In general there is no constraint on the vertex degree, whereas in the latter case the vertex degree is bounded by a constant.

weight	capacity	demand	splittable	Complexity
$*$	u	$*$	SPLIT/NOSPLIT	NP-complete
u	$*$	$*$	SPLIT/NOSPLIT	P
$*$	$*$	u	NOSPLIT	P
$*$	$*$	u	SPLIT	NP-complete

Table 5.1: Some complexity results of CVCDT. We denote the general case with an asterisk ("*"), and uniform weight, capacity, or demand with a "u". These results are presented in [GHKO03].

Unrestricted weight/Uniform weight: In the general case, there is no constraint on the vertex weight, whereas in the uniform weight case we require that all vertices have the same weight. The following two constraints are defined analogously: "Unrestricted capacity/Uniform capacity" and "Unrestricted demand/Uniform demand".

How do different combinations of constraints affect the computational complexity of CVCDT? For instance, Guha et al.[GHKO03] showed that CVCDT with SPLIT and uniform demand is NP-complete, whereas CVCDT with NOSPLIT and uniform demand is in P. More variants and their computational complexity as shown by Guha et al. can be found in Table 5.1. In the next section we will show that:

- CVCDT with NOSPLIT can be solved in $O(2^k \cdot n)$ time, where k and n are the maximum vertex degree and the number of vertices of the input graph, respectively.

- CVCDT with SPLIT, uniform edge demand, and bounded vertex degree can be solved in polynomial time if the edge demand is polynomial in n.

5.2 Fixed-Parameter Tractability

The main result of this section is an algorithm that shows that CVCDT is fixed-parameter tractable with respect to the vertex degree when we assume NOSPLIT. The principle of the algorithm is similar to that of the algorithms calculating the weight of a minimum cvcd-cover on a tree as stated by Guha et al. [GHKO03] for some restricted variants of CVCDT, but it can solve the problem in a more general case (assuming only NOSPLIT). Our algorithm

works in two phases. In the first phase, it computes the weight of a minimum cvcd-cover on a given tree, and in the second phase it computes the corresponding cvcd-cover. We present our algorithm in the following section.

5.2.1 An Algorithm for CVCDT (NOSPLIT)

Given is an instance of the CVCDT problem, that is, a tree $T = (V, E)$ with vertex weights $w(v)$, vertex capacities $c(v)$, and edge demands $d(e)$. We suppose that splitting is not allowed (NOSPLIT). We root T at an arbitrary vertex and assume that every non-root vertex v in V has a smaller index than its parent. Let T_v denote the subtree of T with root v. The idea of the algorithm is the following: We process the vertices in increasing order of their indices and compute the weight of a minimum cvcd-cover on T_v and the weight of a minimum cvcd-cover on T_v with the constraint that v covers the edge to its parent. To compute these weights, we use the already computed weights of the children of v and try all possibilities to cover the edges between v and its children. After having processed every vertex, the weight of a minimum cvcd-cover on T has been computed. After that, the algorithm collects the vertices of the corresponding cvcd-cover in a second step.

In the following we describe more explicitly the variables used in the algorithm. The value stored in W_v^0 denotes the weight of the minimum cvcd-cover on T_v in the case that v does not cover the edge $\{u, v\}$ to its parent u (thus it is covered by u entirely due to NOSPLIT). Accordingly, W_v^1 gives the weight of the minimum cvcd-cover on T_v, assuming that v covers $\{u, v\}$. In order to provide the minimum cvcd-cover itself, we introduce a *choice variable* $C_v^i(w) \in \{0, 1\}$, $i \in \{0, 1\}$ for each child w of a vertex v. For each child w of v the variable $C_v^0(w)$ indicates whether w covers the edge $\{w, v\}$ if v does not cover $\{u, v\}$. Analogously, $C_v^1(w)$ indicates whether w covers the edge $\{w, v\}$ if v covers $\{u, v\}$. Moreover, we use N_v^i to denote the number of copies of vertex v that are assigned to the cvcd-cover. Note that this value is actually not needed as it could be calculated using $C_v^i(w)$, but it eases the description of the algorithm.

Using this notation, the procedure in Figure 5.1, as the first step of our algorithm, computes the weight of a minimum cvcd-cover on T. Afterwards, the procedure in Figure 5.2 computes a minimum cvcd-cover, accomplishing the task of the second step of our algorithm.

Analysis: The following lemma shows that CVCDT is fixed-parameter tractable when parameterized by the maximum vertex degree.

inputs
Rooted tree $T = (V, E)$, weights $w : V \to \mathbb{R}$,
capacities $c : V \to \mathbb{N}$, edge demands $d : E \to \mathbb{N}_0$
output
The weight of a minimum cvcd-cover of T.

begin
  [Initialization of the leaves]
  forall *leaves* v *of* T **do**
    $u \leftarrow$ parent of v
    [Set the number of copies and the corresponding weights]
    $N_v^0 \leftarrow 0$; $N_v^1 \leftarrow \lceil \frac{d(\{u,v\})}{c(v)} \rceil$; $W_v^0 \leftarrow 0$; $W_v^1 \leftarrow w(v) \cdot N_v^1$
  end
  [Main loop]
  for $v \leftarrow v_1$ **to** v_n **do**
    if $v \neq v_n$ **then** $u \leftarrow$ parent of v
    [The value of i indicates whether or not v covers $\{u, v\}$]
    forall $i \in \{0, 1\}$ **do**
      if $v = v_n \wedge i = 1$ **then return** $W_{v_n}^0$
      $A \leftarrow$ set of children of v; $W_{min} \leftarrow \infty$
      forall $S \subseteq A$ **do**
        [Each vertex $w \in S$ covers the edge $\{w, v\}$]
        [d_S is the demand of the edges that have to be covered by v]
        $d_S \leftarrow \sum_{w \in A \setminus S} d(\{v, w\}) + i \cdot d(\{u, v\})$
        [W_S is the weight of the corresponding cvcd-cover]
        $W_S \leftarrow \sum_{w \in S} W_w^1 + \sum_{w \in A \setminus S} W_w^0 + \lceil d_S / c(v) \rceil \cdot w(v)$
        if $W_{min} > W_S$ **then** $W_{min} \leftarrow W_S$; $S_{min} \leftarrow S$; $d_{min} \leftarrow d_S$
      end
      [Sets W_v^i and N_v^i correspond to a minimum cvcd-cover]
      $W_v^i \leftarrow W_{min}$; $N_v^i \leftarrow \lceil d_{min} / c(v) \rceil$
      [Set variables $C_v^i(w)$ which indicate whether or not vertex w covers $\{w, v\}$ for all children w of v. These variables will be used by the second phase.]
      forall $w \in A \setminus S_{min}$ **do** $C_v^i(w) \leftarrow 0$
      forall $w \in S_{min}$ **do** $C_v^i(w) \leftarrow 1$
    end
  end
  return $W_{v_n}^0$
end

Figure 5.1: Procedure to compute the weight of a minimum cvcd-cover of a tree.

inputs
Rooted tree $T = (V, E)$, weights $w : V \to \mathbb{R}$,
capacities $c : V \to \mathbb{N}$, edge demands $d : E \to \mathbb{N}_0$
output
A minimum cvcd-cover of T.

```
1  begin
2      return getCover(v_n,0)
3  end

4  function getCover(v ∈ V, i ∈ {0,1})
5      M ← {(v, N_v^i)}
6      A ← set of children of v
       [For each child w of v we collect recursively a cvcd-cover
       of T_w, using the variable C_v^i which indicates whether w
       covers {w,v} or not (C_v^i was computed in the first phase of
       the algorithm).]
7      forall w ∈ A do
8          M ← M ∪ getCover(w, C_v^i(w))
9      end
10     return M
11 end function
```

Figure 5.2: Procedure to compute a minimum cvcd-cover of a tree. Function "getCover()" returns the cvcd-cover of T_v depending on whether or not v covers the edge to its parent. Each element of M is a pair of a vertex v and the number of copies of v in the cvcd-cover.

Lemma 5.2.1. *The procedure in Figure 5.1 correctly calculates the weight of a minimum cvcd-cover of a tree T in $O(2^k \cdot n)$ time, where k denotes the maximum vertex degree. Moreover, the values of C_v^i and N_v^i ($i \in \{0,1\}$) needed for the second phase are correctly set for each vertex v.*

Proof. The correctness of this procedure can be shown by induction on the vertex index: if we assume for a vertex v_j that the values $N_{v_l}^i$, $W_{v_l}^i$ ($i \in \{0,1\}$) for all $1 \leq l \leq j-1$ have been correctly computed, then the values $N_{v_j}^i$, $W_{v_j}^i$ ($i \in \{0,1\}$) are computed correctly, since the inner loop in line 11 tries brute force all combinations to cover the edges from v_j to its children and selects a combination with minimum weight. The combination which leads to a minimum weight then is stored in $C_{v_j}^i$ ($i \in \{0,1\}$) in line 17.

It remains to analyze the running time. The initialization in line 2 runs in linear time, and the main loop iterates exactly n times. Each iteration passes the two cases $i = 0$ and $i = 1$, trying for each case at most 2^{k-1}

possible subsets, so the total running time is $O(2^k \cdot n)$. □

The next step is to compute the minimum cvcd-cover itself, which can be done in $O(n)$ time.

Lemma 5.2.2. *The procedure in Figure 5.2 constructs a minimum capacitated vertex cover for a tree T in time $O(n)$.*

Proof. After running the procedure in Figure 5.1 the variables $C_v^i(w)$ and N_v^i are correctly computed (see Lemma 5.2.1). Thus, for each of the two cases $i \in \{0, 1\}$ whether a vertex v covers the edge to its parent u or not we know that there are N_v^i copies of v in a cvcd-cover, and that a cvcd-cover has minimum weight if each child w of v with $C_v^i(w) = 1$ covers $\{v, w\}$. We apply the first case ($i = 0$) to the root v_n (it does not have a parent). This determines the cases for all the remaining vertices $v_1, \ldots, v_{n-1}$ in the tree, so we just have to collect the vertices of the minimum cvcd-cover by using the recursive procedure "getCover". The first call of this procedure in line 2 results in a recursive call for each child, where each call needs constant time. Since there are $n - 1$ predecessors of v_n in total, the running time of the second phase is $O(n)$. □

Using Lemma 5.2.1 and Lemma 5.2.2, we can state the following theorem.

Theorem 5.2.1. *In the case that splitting is not allowed, CVCDT can be solved in $O(2^k \cdot n)$ time, where k and n denote the maximum vertex degree and the number of vertices of the input tree, respectively.*

5.2.2 Splitting Demands

We have shown that the CVCDT problem can be solved in $O(2^k \cdot n)$ time in the NOSPLIT-case, where k denotes the maximum vertex degree of the given tree with n vertices. However, it is also possible to solve CVCDT with uniform demand in the SPLIT-case with a slightly modified algorithm. We describe the idea of this modification briefly in the following.

Recall that the main principle of the algorithm is to compute for each vertex v the weight of a minimum cvcd-cover on subtree T_v for the two cases whether or not v covers the edge to its parent u. The weight of a minimum cvcd-cover on T_v for each of these cases is computed by trying brute force all configurations of whether or not v covers an edge to its children.

The modified algorithm for the SPLIT-case with uniform demand and bounded vertex degree works similar: A vertex v can cover an edge partially. Thus we have to compute for each vertex v the weight of a minimum cvcd-cover on subtree T_v for each possibility to assign capacity units from v to the edge to its parent u. If the uniform demand is denoted by $d \in \mathbb{N}$, then

there are at most $d+1$ ways to assign capacity of v to $\{u, v\}$. The weight of a minimum cvcd-cover on T_v for each of these cases is computed by trying brute force all configurations of distributing capacity units from v to the edges to its child vertices. This gives $(d+1)^k$ possible configurations, and this means that the total running time to compute a minimum cvcd-cover is $O((d+1)^k \cdot n)$.

This means that if d is polynomial in n, then this modificated algorithm computes the weight of a minimum cvcd-cover in polynomial time for trees with bounded vertex degree k. The appropriate cvcd-cover can be computed by a backtracking step similar to the procedure in Figure 5.2. We summarize this result with the following corollary:

Corollary 5.2.1. *Assuming uniform edge demand d, CVCDT can be solved in $O((d+1)^k \cdot n)$ time, where k and n denote the maximum vertex degree and the number of vertices of the input tree, respectively.*

Chapter 6

Conclusion

In this chapter we give a brief summary of this work and its results. Moreover, we give some suggestions of starting points for related future research.

6.1 Summary

In Chapter 2 we defined the basic notation needed for this work and gave a brief introduction to parameterized complexity theory. After that, in Chapter 3, we introduced Partial Vertex Cover, Connected Vertex Cover, and Capacitated Vertex Cover, the three key problems of this work. In Chapter 4 we provided a basic introduction to tree decompositions and treewidth and used the technique of dynamic programming on tree decompositions to show that

- Partial Vertex Cover is fixed-parameter tractable with respect to treewidth,
- Connected Vertex Cover is fixed-parameter tractable with respect to treewidth,
- Capacitated Vertex Cover is fixed-parameter tractable with respect to treewidth for graphs with bounded vertex degree, and
- Capacitated Vertex Cover can be solved in $O(k^{2\omega} \cdot \omega \cdot |I|)$ time, where k denotes the maximum size of the capacitated vertex cover, and ω and $|I|$ denote the treewidth and the number of nodes of the nice tree decomposition, respectively.

In Section 5 we gave an overview of existing work about Minimum Capacitated Vertex Cover with Demand on Trees, and showed the following.

- In the case that the splitting of edge demands is not allowed, Minimum Capacitated Vertex Cover with Demand on Trees is fixed-parameter tractable with respect to the maximum vertex degree as parameter.

- For trees with uniform edge demand d, where the splitting of the edge demand is allowed, Minimum Capacitated Vertex Cover with Demand on Trees can be solved in $O((d+1)^k \cdot n)$ time, where k denotes the maximum solution size, and n the number of vertices of the input graph. The case without restrictions on edge demands remains open.

6.2 Open Problems

For Partial Vertex Cover and Connected Vertex Cover it would be interesting to see how we can improve the running time of the dynamic programming to get better worst-case running time bounds.

Another interesting question is whether or not Capacitated Vertex Cover is fixed-parameter tractable with respect to treewidth as parameter. This could be shown with different approaches. One possibility would be to improve our dynamic programming approach on nice tree decompositions. Another method to show the fixed-parameter tractability of a graph problem is to formulate the problem in monadic second order logic [Bod97, CMR01]. Moreover, it could be interesting to learn more about Minimum Capacitated Vertex Cover with Demand on Trees because its main difficulty of how to "distribute capacities" also occurs in the dynamic programming on tree decompositions to solve Capacitated Vertex Cover. Also, it would be interesting to investigate the practical behavior of our algorithms, so another future task would be to implement the described algorithms.

6.3 Acknowledgments

I would like to thank my advisors Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke for the considerable effort they devoted to this thesis. They introduced me to parameterized complexity theory, gave many helpful advises, and spent hours in discussing my work and proof-reading my drafts. I am especially grateful for what they taught me about scientific writing, which will be certainly beneficial to me in the future. Moreover, I would like to thank Tobias Berg and Matthias Hagen for reading my final draft and giving me some helpful comments.

Bibliography

[ABF+02] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, and Rolf Niedermeier. Fixed parameter algorithms for Dominating Set and related problems on planar graphs. *Algorithmica*, 33(4):461–493, 2002. → 4

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987. → 23

[ACPS93] Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, 1993. → 24

[AFN04] Jochen Alber, Henning Fernau, and Rolf Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. *Journal of Algorithms*, 52(1):26–56, 2004. → 4

[AHH93] Esther M. Arkin, Magnus M. Halldórsson, and Refael Hassin. Approximating the tree and tour covers of a graph. *Information Processing Letters*, 47(6):275–282, 1993. → 17

[AKCF+04] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christof 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, 2004. → 3, 4

[AKLSS05] Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable parallel algorithms for FPT problems. To appear in *Algorithmica*, 2005. → 3

[Alb03] Jochen Alber. *Exact Algorithms for NP-hard Problems on Networks: Design, Analysis, and Implementation.* Ph.D. dissertation, Universität Tübingen, Germany, 2003. → 6

[AP86] Stefan Arnborg and Andrzej Proskurowski. Characterization and recognition of partial 3-trees. *SIAM Journal of Algebraic and Discrete Methods*, 7(2):305–314, 1986. → 24

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989. → 43

[AR02] Vikraman Arvind and Venkatesh Raman. Approximation algorithms for some parameterized counting problems. In *Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC'02)*, volume 2518 of *LNCS*, pages 453–464. Springer, 2002. → 4

[BB98] Nader H. Bshouty and Lynn Burroughs. Massaging a linear programming solution to give a 2-approximation for a generalization of the vertex cover problem. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS'98)*, volume 1373 of *LNCS*, pages 298–308. Springer, 1998. → 5, 15

[BdF96] Hans L. Bodlaender and Babette de Fluiter. Reduction algorithms for constructing solutions in graphs with small treewidth. In *Proceedings of the Second Annual International Conference on Computing and Combinatorics (COCOON'96)*, volume 1090 of *LNCS*, pages 199–208. Springer, 1996. → 6

[BFR98] R. Balasubramanian, Michael R. Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65(3):163–168, 1998. → 4, 5

[BGHK95] Hans L. Bodlaender, John R. Gilbert, Hjálmtyr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2):238–255, 1995. → 24

[BH98] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27(6):1725–1746, 1998. → 24

[Blä03] Markus Bläser. Computing small partial coverings. *Information Processing Letters*, 85(6):327–331, 2003. → 16

[BLS99] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: a survey.* Society for Industrial and Applied Mathematics, 1999. → 45

[Bod88a] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming (ICALP'88)*, volume 317 of *LNCS*, pages 105–118. Springer, 1988. → 6

[Bod88b] Hans L. Bodlaender. Some classes of graphs with bounded treewidth. *Bulletin of the EATCS*, 36:116–125, 1988. → 6

[Bod93] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993. → 6

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996. → 24

[Bod97] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS'97)*, volume 1295 of *LNCS*, pages 19–36. Springer, 1997. → 6, 64

[CDRC+03] James Cheetham, Frank 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. → 4

[CG05] L. Sunil Chandran and Fabrizio Grandoni. Refined memorization for vertex cover. *Information Processing Letters*, 93(3):123–131, 2005. → 4, 5

[CKJ01] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001. → 4

[CMR01] Bruno Courcelle, Janos A. Makowsky, and Udi Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1):23–52, 2001. → 64

[CN02] Julia Chuzhoy and Joseph Naor. Covering problems with hard capacities. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS'02)*, pages 481–489. IEEE Computer Society, 2002. → 5, 18

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999. → 4, 10, 12, 13

[DH05] Erik D. Demaine and Mohammad T. Hajiaghayi. Bidimensionality: New connections between FPT algorithms and PTASs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 590–601, 2005. → 17, 35

[DHT02] Erik D. Demaine, Mohammad T. Hajiaghayi, and Dimitrios M. Thilikos. Approximation for treewidth of graphs excluding a graph with one crossing as a minor. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX'02)*, volume 2462 of *LNCS*, pages 67–80. Springer, 2002. → 24

[Die05] Reinhard Diestel. *Graph Theory*. Springer, 3rd edition, 2005. → 10, 26

[DS02] Irit Dinur and Shmuel Safra. The importance of being biased. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 33–42. ACM Press, 2002. → 4

[Fel03] Michael R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized. In *Proceedings of the 8th Workshop on Algorithms and Data Structures (WADS'03)*, volume 2748 of *LNCS*, pages 505–520. Springer, 2003. → 4

[GHK+03] Rajiv Gandhi, Eran Halperin, Samir Khuller, Guy Kortsarz, and Aravind Srinivasan. An improved approximation algorithm for vertex cover with hard capacities. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *LNCS*, pages 164–175. Springer, 2003. → 5, 18

[GHKO03] Sudipto Guha, Refael Hassin, Samir Khuller, and Einat Or. Capacitated vertex covering. *Journal of Algorithms*, 48(1):257–270, 2003. → 5, 6, 18, 19, 55, 57

[GKS04] Rajiv Gandhi, Samir Khuller, and Aravind Srinivasan. Approximation algorithms for partial covering problems. *Journal of Algorithms*, 53(1):55–84, 2004. → 5, 15, 16

[GNW05] Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke. Parameterized complexity of generalized vertex cover problems. In *Proceedings of the 9th Workshop on Algorithms and Data Structures (WADS'05)*, volume 3608 of *LNCS*, pages 36–48. Springer, 2005. → 5, 15, 16, 17, 18

[Gro05] Vic Grout. Principles of cost minimisation in wireless networks. *Journal of Heuristics*, 11(2):115–133, 2005. → 17

[HS02] Eran Halperin and Aravind Srinivasan. Improved approximation algorithms for the partial vertex cover problem. In *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX'02)*, volume 2462 of *LNCS*, pages 161–174. Springer, 2002. → 5, 15

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. → 3

[KBH02] Arie M.C.A. Koster, Hans L. Bodlaender, and Stan P.M. Hoesel. Treewidth: Computational experiments. Technical report, Maastricht Research School of Economics of Technology and Organization, Maastricht, Netherlands, 2002. → 24

[KKPS03] Jochen Könemann, Goran Konjevod, Ojas Parekh, and Amitabh Sinha. Improved approximations for tour and tree covers. *Algorithmica*, 38(3):441–449, 2003. → 17

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994. → 25

[Knu05] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005. → 43

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, forthcoming, 2006. → 6

[NR99] Rolf Niedermeier and Peter Rossmanith. Upper bounds for vertex cover further improved. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 561–570. Springer, 1999. → 4

[NR03] Rolf Niedermeier and Peter Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, 47(2):63–77, 2003. → 4, 5

[PS03] Elena Prieto and Christian Sloper. Either/or: Using vertex cover structure in designing FPT-algorithms—the case of k-internal spanning tree. In *Proceedings of the 8th Workshop on Algorithms and Data Structures (WADS'03)*, volume 2748 of *LNCS*, pages 474–483. Springer, 2003. → 4

[Ree92] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC'92)*, pages 221–228. ACM Press, 1992. → 24

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. → 6, 22

[San96] Daniel P. Sanders. On linear recognition of tree-width at most four. *SIAM Journal on Discrete Mathematics*, 9(1):101–117, 1996. → 24

[Sch95] Berry Schoenmakers. A new algorithm for the recognition of series parallel graphs. Technical report, CWI Amsterdam, The Netherlands, 1995. → 45

[SLM+05] Yinglei Song, Chunmei Liu, Russell Malmberg, Fangfang Pan, and Liming Cai. Tree decomposition based fast search of RNA structures including pseudoknots in genomes. In *Proceedings of the 4th Computational Systems Bioinformatics Conference (CSB'05)*, pages 223–234. IEEE Computer Society, 2005. → 6

[TNS82] Kazuhiko Takamizawa, Takao Nishizeki, and Nobuji Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM*, 29(3):623–641, 1982. → 43

[VTL82] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982. → 45

[XJB05] Jinbo Xu, Feng Jiao, and Bonnie Berger. A tree-decomposition approach to protein structure prediction. In *Proceedings of the 4th Computational Systems Bioinformatics Conference (CSB'05)*, pages 247–256. IEEE Computer Society, 2005. → 6

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, den 09.11.2005 (Hannes Moser)