Karlsruhe Institute of Technology

On Parameterized Complexity of Clique-Partitioned Treewidth and Related Parameters

Master's Thesis of

Sven Geißler

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

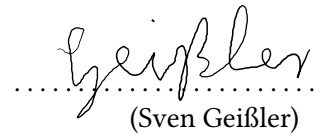
Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	Prof. Dr. Marvin Künnemann
Advisors:	Marcus Wilhelm
	Mirza Redzic

22 November 2025 – 22 May 2026

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 22 May 2026



(Sven Geißler)

Abstract

Treewidth is one of the most important and powerful graph parameters. However, many real-world instances do not admit tree-decompositions of small width, even though their bags exhibit a simple internal structure. To address this limitation, numerous refinements of treewidth have been proposed. At present, though, only few efficient algorithms for computing such parameters are known. One such refinement, for which many $\mathcal{NP}$-complete problems can be solved in FPT-time, while no efficient algorithms for computing the parameter have been established so far, is *clique-partitioned treewidth*. This parameter refines treewidth by capturing the internal structure of bags through partitions of the bags into logarithmically weighted cliques.

We investigate the computational complexity of clique-partitioned treewidth and present several algorithmic results. First, we develop an exact algorithm with running time $2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}$. Second, we give a $4k + 3$-approximation algorithm running in time $2^{2^{\mathcal{O}(k)}} \cdot n^3$. Furthermore, we design an FPT-algorithm with running time $2^{2^{\mathcal{O}(k)}} \cdot n^2$ for computing globally-partitioned treewidth, a variant of clique-partitioned treewidth that employs a global partition of the input graph instead of local partitions of the bags. Finally, we prove that computing either parameter exactly, as well as computing any additive approximation of either parameter, is $\mathcal{NP}$-hard.

Zusammenfassung

Baumweite ist einer der wichtigsten und mächtigsten Graphparameter. Allerdings besitzen viele reale Instanzen keine Baumzerlegungen mit kleiner Weite, obwohl ihre Bags eine einfache interne Struktur aufweisen. Um diese Einschränkung zu überwinden, wurden zahlreiche Verfeinerungen der Baumweite vorgeschlagen. Gegenwärtig sind jedoch nur wenige effiziente Algorithmen zur Berechnung solcher Parameter bekannt. Eine solche Verfeinerung, mit deren Hilfe viele $\mathcal{NP}$-vollständige Probleme in FPT-Zeit gelöst werden können, während bislang keine effizienten Algorithmen zur Berechnung des Parameters bekannt sind, ist die *cliquen-partitionierte Baumweite*. Dieser Parameter verfeinert die Baumweite, indem er die interne Struktur der Bags durch Partitionen in logarithmisch gewichtete Cliquen erfasst.

Wir untersuchen die Berechnungskomplexität der cliquen-partitionierten Baumweite und präsentieren mehrere algorithmische Ergebnisse. Zunächst entwickeln wir einen exakten Algorithmus mit Laufzeit $2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}$. Außerdem geben wir einen $4k + 3$-Approximationsalgorithmus mit Laufzeit $2^{2^{\mathcal{O}(k)}} \cdot n^3$ an. Darüber hinaus entwerfen wir einen FPT-Algorithmus mit Laufzeit $2^{2^{\mathcal{O}(k)}} \cdot n^2$ zur Berechnung der global-partitionierten Baumweite, einer Variante der cliquen-partitionierten Baumweite, die eine globale Partition des Eingabegraphen anstelle der lokalen Partitionen der Bags nutzt. Schließlich zeigen wir, dass sowohl die exakte Berechnung beider Parameter als auch jede additive Approximation dieser Parameter $\mathcal{NP}$-schwer ist.

Contents

1 Introduction

Many fundamental graph problems, such as TSP, HamiltonCycle, and IndependentSet, have been shown to be $\mathcal{NP}$-complete. Although $\mathcal{NP}$-complete problems are generally considered algorithmically intractable, many practical applications, such as route planning, scheduling, and integrated circuit design [LK75], rely on solving them efficiently. Fortunately, many real-world instances exhibit additional structural properties that can be exploited algorithmically (see for example [Lam+19]). Parameterized complexity provides a framework for partially explaining this phenomenon by formalizing structural features of instances that enable efficient algorithms. Formally, an integer parameter k is associated with the input, and the goal is to design *fixed-parameter tractable* (FPT) algorithms with running time $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a computable function and n denotes the input size.

One structural property frequently exhibited by real-world instances is the presence of hierarchical structures that resemble trees. This observation is particularly promising because many $\mathcal{NP}$-hard problems can be solved efficiently on trees. The parameter *treewidth* formalizes this idea by measuring how tree-like a graph is. It is defined via *tree-decompositions*, which organize the graph into a recursive hierarchy of balanced separators, that are called bags. While at first glance this concept may appear somewhat artificial, perhaps the strongest evidence for its naturalness lies in the fact that it was discovered independently on three separate occasions: by Bertele and Brioschi [BB73], by Rudolf Halin [Hal76], and by Robertson and Seymour [RS86]. Since its introduction, treewidth has become a powerful tool in algorithm design and has been used to obtain FPT-algorithms based on dynamic programming over tree-decompositions for numerous problems, including Independent Set, Dominating Set, and Hamilton Cycle (see [Cyg+15]). Despite its usefulness for algorithm design, for a long time a major open question was whether optimal tree-decompositions could be computed efficiently. Towards answering this question, both an exact XP-algorithm [ACP87] and an FPT-time approximation algorithm [BGKH91 | RS95] were developed. Finally, in 1996, Bodlaender and Kloks [BK96] presented an FPT-algorithm for computing optimal tree-decompositions. These results laid the foundation for the widespread success of treewidth, which has since become one of the most extensively studied graph parameters.

However, many real-world graphs admit only tree-decompositions with large bags while still exhibiting hierarchical decompositions into highly-structured balanced separators. Chordal graphs, for example, admit tree-decompositions in which every bag forms a clique. Consequently, they may have large treewidth, even though the structure of bags can be exploited algorithmically [Gol04]. More generally, large cliques are a major contributor to high treewidth [Bod98], despite their simple and well-understood structure. To address this limitation, several refinements of treewidth have been proposed that incorporate additional structural information, such as the presence of large cliques, into the parameter. Typically, these parameters assign each bag a weight depending on the subgraph induced by the bag and define the parameter in terms of these weights rather than the bag sizes alone.

One of the earliest such parameters is *$\mathcal{P}$-flattened treewidth*, introduced by De Berg et al. [Ber+18]. This parameter captures the structure of bags using a global partition of the input graph into cliques together with the contraction of unions of constantly many such cliques.

The weight of a bag is then defined using the logarithms of the sizes of these unions. As a consequence, large cliques contribute only logarithmically to the parameter value. This parameter has been used to establish algorithms and lower bounds for several $\mathcal{NP}$-hard problems on specific classes of intersection graphs [Ber+18]. However, the only known algorithm for computing the parameter is an exponential-time approximation algorithm restricted to certain graph classes [Ber+18].

Extending this idea, Bläsius et al. [BKW23] introduced *clique-partitioned treewidth*, which replaces the global clique-partition by local partitions of the individual bags. Again, each clique contributes only logarithmically to the weight of a bag. The algorithmic potential of clique-partitioned treewidth has already been investigated. In particular, it has been shown that clique-partitioned treewidth never performs worse than treewidth for algorithm design [Gei23]. Although improvements in the parameter value may in the worst case be offset by a stronger running time dependence on the parameter, several problems, including Independent Set, Maximum Induced Forest, and Feedback Vertex Set, admit improved algorithms when clique-partitioned treewidth is significantly smaller than treewidth [Gei23]. For the computation of the parameter, however, only experimental results are currently available. Bläsius et al. [BKW23] proposed an approach that first computes a low-width tree-decomposition and then uses heuristics and other approaches to determine clique-partitions for each bag. Nevertheless, no efficient algorithms for computing the parameter either exactly or with a guaranteed approximation ratio are known so far. This thesis aims to close this gap by designing several approaches for computing clique-partitioned treewidth.

Besides assigning logarithmic weights to cliques, other approaches for capturing the structural simplicity introduced by large cliques have also been studied. These approaches are based on graph-theoretic concepts closely related to clique-partitions. One such concept is the size of a minimum edge clique-cover, which, in contrast to weighted clique-partitions, assigns a constant weight to each clique while additionally requiring all edges to be covered. Using this quantity to define the weight of a bag led to the introduction of *tree-clique width* by Aronis [Aro21]. Although no algorithmic applications of this parameter are currently known, several complexity results regarding its computation haven been established. In particular, computing tree-clique width is $\mathcal{NP}$-complete and no polynomial constant-factor approximation algorithm exists unless $\mathcal{P} = \mathcal{NP}$ [Aro21]. On the algorithmic side, single-exponential algorithms for general graphs and polynomial-algorithms for special graph classes such as cographs and permutation graphs are known [Aro21].

Another concept closely related to cliques is that of independent sets. In particular, defining the weight of a bag as the size of a maximum independent set ensures that each clique contributes at most one to the parameter value. Using this approach, Dallard et al. [DMS24] introduced the *tree-independence number*. This parameter can be used to solve several $\mathcal{NP}$-hard-problems, including Maximum Independent Set, Graph Homomorphism, and Maximum Induced Matching, in time $n^{\mathcal{O}(k)}$ [Yol18 | DMS24]. For its computation, an XP-time approximation algorithm is known [Dal+26]. However, for every fixed constant k, deciding whether the tree-independence number of a graph is at most k is known to be $\mathcal{NP}$-hard [Dal+26]. While we do not improve upon these algorithmic or hardness results, we use the tree-independence number as a baseline for placing our results on clique-partitioned treewidth into context.

Overall, parameters refining treewidth appear to be in a position similar to that of treewidth in its early days. Many promising parameters are known, but efficient algorithms for computing optimal tree-decompositions for them are still lacking. In particular, one of the central open questions in this context is for which of these refinements of treewidth, most notably

the tree-independence number, FPT-algorithms with respect to either the parameter itself or treewidth can be obtained. This thesis contributes towards answering this problem by studying the computation of two parameters that are based on logarithmically weighted clique-partitions. Besides clique-partitioned treewidth mentioned above, we also introduce and study *globally-partitioned treewidth*, a new parameter based on a global clique-partition of the graph.

1.1 Results

In our study of clique-partitioned treewidth and globally-partitioned treewidth, we present three main contributions. As our first contribution, we adapt two well-known approaches for computing treewidth to the setting of clique-partitioned treewidth. In particular, we show that the parameter can be computed exactly in XP-time, more precisely in time $2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}$, and that a $4k+3$-approximation can be obtained in FPT-time, more precisely in time $2^{2^{\mathcal{O}(k)}} \cdot n^3$. We further show that a direct adaptation of these approaches fails for the closely related parameter of tree-clique-cover number, further demonstrating that not all refinements of treewidth allow this adaptation.

Our second contribution establishes lower bounds for the computational complexity of determining clique-partitioned treewidth. More precisely, we present a reduction from Pathwidth to clique-partitioned Treewidth. As part of this reduction, we introduce a novel intermediate problem and develop a new technique based on grid constructions to enforce specific configurations in path-decompositions. Using this reduction, we derive that computing clique-partitioned treewidth is $\mathcal{NP}$-hard and that computing an approximation within an additive constant is also $\mathcal{NP}$-hard.

While these algorithmic and hardness results provide an initial understanding of the complexity of computing clique-partitioned treewidth, a gap remains between the upper and lower bounds. To address this gap, our third contribution introduces a variant called globally-partitioned treewidth. This parameter is larger than clique-partitioned treewidth but has the additional property that all local clique-partitions are compatible and together form a global partition of the input graph. Exploiting this additional structure, we adapt the FPT-algorithm of Bodlaender and Kloks [BK96] for computing treewidth to this new setting. We thus obtain a $2^{2^{\mathcal{O}(k)}} \cdot n^2$-time algorithm for computing globally-partitioned treewidth. In the course of developing this algorithm, we further improve upon the simplified formulation of the original algorithm due to Althaus and Ziegler [AZ21]. Combined with an extension of the earlier hardness result to globally-partitioned treewidth, this yields a complete complexity classification for the parameter. Overall, our results demonstrate that parameters based on weighted clique-partitions can be computed more efficiently than the tree-independence number.

We extend this comparison by considering both the relative sizes of the different parameters and the graph problems that admit efficient algorithms when parameterized by them. In particular, we show that clique-partitioned treewidth improves on treewidth with respect to parameter size while still admitting many parameterized algorithms. In contrast, all considered parameters that are strictly smaller than clique-partitioned treewidth admit strictly fewer algorithms than treewidth.

Taken together, these contributions lay the groundwork for the systematic study and computation of clique-partitioned treewidth. In particular, they provide efficient approximation algorithms for clique-partitioned treewidth and efficient exact algorithms for the larger pa-

rameter of globally-partitioned treewidth. To the best of our knowledge, these are the first exact or approximation algorithms running in FPT-time with respect to either the parameter itself or even treewidth for any of the aforementioned refinements of treewidth.

1.2 Outline

We structure this thesis as follows. In Chapter 2, we introduce the notation used throughout the thesis and define the different parameters under consideration. In Chapter 3, we present several well-known results on weighted clique-partitions and treewidth and extend them to fit our setting and applications. In Chapter 4, we compare clique-partitioned treewidth and globally-partitioned treewidth with several related parameters from literature with respect to both parameter size and algorithmic usefulness. We then turn to algorithmic aspects and our first contribution in Chapter 5, where we develop an XP-algorithm for the exact computation of clique-partitioned treewidth, as well as an FPT-time approximation algorithm. Before presenting our second contribution, namely hardness-results for our two parameters, in Chapter 7, we establish an auxiliary result in Chapter 6. There, we show that certain bags in every path-decomposition must satisfy a minimum-size condition. This structural insight is then used in the reduction presented in Chapter 7. Finally, in Chapter 8, we present our third contribution. We reformulate the algorithm of Bodlaender and Kloks [BK96 | AZ21] for computing treewidth and adapt it to the setting of globally-partitioned treewidth. We conclude in Chapter 9 with a summary of our results and a discussion of open problems.

2 Preliminaries

In this chapter, we outline the concepts used in this work. We begin with general graph-theoretic notation and selected fundamental notions. We then introduce several graph families that will be used throughout the thesis. Finally, we give a brief introduction to the tree-decomposition-based parameters considered in this work.

2.1 On graphs

This section provides a brief overview our graph-theoretic notation and terminology.

2.1.1 Notation

We write $[n] = \{1, \dots, n\}$ for the set of the first n natural numbers. Throughout this work, logarithms are taken to base 2 unless stated otherwise. For a set X, we denote its power set by 2^X.

Unless stated otherwise, we consider only simple, undirected graphs and abbreviate an edge $\{u, v\}$ by uv. For a graph G, we use $V(G)$ and $E(G)$ to denote its vertex and edge sets, respectively. We write $n = |V(G)|$ and $m = |E(G)|$ whenever the graph is clear from the context. For a vertex set $X \subseteq V(G)$, we denote the *subgraph induced* by X as $G[X]$. For a vertex v, we denote by $N(v)$ the *neighbourhood* of v, that is, the set of all vertices adjacent to v. We use $+$ and $-$ to denote the insertion and removal of elements from a set. This notation naturally extends to graphs: We write $G + v$, $G + X$, $G - v$, and $G - X$ for the graph obtained by inserting or removing a vertex v (or vertex set X) together with all incident edges. Similarly, $G - uv$ denotes the removal of the edge uv while keeping its endpoints. Given a weight function $w : V(G) \to \mathbb{R}_0$ on vertices, we extend it to vertex sets by defining $w(X) = \sum_{x \in X} w(x)$.

2.1.2 Selected graph concepts

We now introduce several concepts that will be used throughout this work. A vertex set $S \subseteq V(G)$ is called a *separator* if $G - S$ has strictly more connected components that G. In particular, if G is connected, then $G - S$ must be disconnected. We say that a separator S *separates* a graph $G = (V, E)$ *into two vertex sets* V_1 and V_2, if $V = S \cup V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and there are no edges between V_1 and V_2. For a weight function $w : V(G) \to \mathbb{R}_0$, such a separator S is called *α-balanced*, if $w(V_1) \leq \alpha \cdot w(V(G))$ and $w(V_2) \leq \alpha \cdot w(V(G))$.

A vertex v is called *universal* with respect to a vertex set X if it is adjacent to all vertices in X. For a graph $G = (V, E)$, its *complement graph* is defined as $\overline{G} = (V, \binom{V}{2} - E)$. A vertex set X is a *clique* if every pair of vertices in X is adjacent. The size of a maximum clique in G is called the *clique number* $\omega(G)$ of G. A *clique-cover* of a graph G is a sequence of sets $V_1 \cup \dots \cup V_r = V(G)$ such that each V_i is an clique. The minimum number of such sets in any clique-cover is called the *clique-cover number* $\kappa(G)$ of G. If the sets $V_1, \dots, V_r$ are pairwise disjoint, we call this a *clique-partition*. For a set X and a clique-partition $\mathcal{P}$, we denote by $\mathcal{P}[X]$ the restriction of $\mathcal{P}$ to X. A vertex set X is *independent* if no pair of vertices in X is adjacent.

The size of a maximum independent set in G is called the *independence number* $\alpha(G)$ of G. A *vertex colouring* of a graph G is a partition of $V(G)$ into disjoint set $V_1 \cup \cdots \cup V_r = V(G)$ such that each V_i is an independent set. The minimum number of such sets in any vertex colouring is called the *chromatic number* $\chi(G)$ of G. Observe that in the complement graph, cliques correspond to independent sets and vice versa. Similarly, vertex colourings correspond to clique-covers in the complement graph.

In this thesis, we consider parameterized problems. A *problem* Π *parameterized by a parameter* τ is a formal language $\Pi_\tau \subseteq \Sigma^* \times \mathbb{R}_0$ over an alphabet Σ. Since tree-decomposition-based methods can only be applied to problems containing a graph, we restrict our attention to graph problems. A problem Π_τ is called a *graph problem parameterized by a parameter* τ if, for each instance $(I, k) \in \Pi_\tau$, the instance I contains a graph G such that $\tau(G) = k$.

2.2 Graph families

We now introduce several graph families used throughout this work. The family of *edgeless graphs* consists of all graphs with vertex set $[n]$, $n \in \mathbb{N}_+$, and empty edge set. In contrast, the family of *complete graphs* contains all graphs with vertex set $[n]$ and edge set $\binom{[n]}{2}$. We denote these graphs by K_n. In particular, K_3 is called a *triangle*. The family of *paths* consists of graphs with vertex set $[n]$, $n \in \mathbb{N}_+$, and edge set $\{\{i, i+1\} \mid i \in [n-1]\}$. The family of *cycles* is obtained by adding the edge $\{1, n\}$ to such paths. The family of *trees* consists of connected graphs with n vertices and $n-1$ edges, $n \in \mathbb{N}_+$. The family of *grids* contains all graphs with vertex set $[\alpha] \times [\beta]$, where $\alpha \in \mathbb{N}_0$ is the number of rows and $\beta \in \mathbb{N}_0$ is the number of columns. The edge set consists of all pairs $\{(a, b), (a+1, b)\}$ for $a \in [\alpha - 1]$, $b \in [\beta]$, and $\{(a, b), (a, b+1)\}$ for $a \in [\alpha]$, $b \in [\beta - 1]$. An induced subgraph of a grid is called a *grid graph*. Finally, we consider *chordal graphs*. A graph G is chordal if it does not contain an induced cycle of length greater than three. Equivalently, every cycle of length at least four has a *chord*, that is, an edge connecting two non-consecutive vertices of the cycle. For a graph G, we define the family of *G-free graphs* as the family of graphs that do not contain G as an induced subgraph.

2.3 A brief introduction to treewidth, clique-partitioned treewidth, and related parameters

We now introduce the different parameters studied in this thesis. We first define the parameters and their associated decompositions, together with additional notions specific to individual parameters. We then present several general definitions that apply across all parameters.

2.3.1 Defining the parameters

We begin by introducing pathwidth and treewidth, followed by the parameters clique-partitioned treewidth and globally-partitioned treewidth, as well as several related measures.

Pathwidth. A *path-decomposition* of a graph G is a pair (P, B), where P is a path and $B : P \to 2^{V(G)}$ is a function that assigns to each node $p \in V(P)$ a subset of vertices, called *bag*, such that the following conditions hold: (1) *Vertex coverage:* every vertex of G is contained in at least one bag, (2) *Edge coverage:* for every edge $uv \in E(G)$, there exists a bag containing

both u and v, and (3) *Connectivity:* for each vertex $v \in V(G)$ the set of nodes whose bags contain v induces a connected subpath of P. The *width* of a path-decomposition is the size of its largest bag minus one. The *pathwidth* $\mathrm{pw}(G)$ of G is the minimum width over all path-decompositions of G.

Throughout this work, we use the following notation for (path-)decompositions. We refer to the vertices of the path P as *nodes* and reserve the term *vertices* for the elements of $V(G)$. Thus, nodes correspond to vertices of the decomposition structure, while bags are sets of vertices of G. Properties defined for nodes or bags are extended to the other in the natural way. Given a path-decomposition $(P = (p_1, \ldots, p_q), B)$ we say that a node $p_i \in V(P)$ is *strictly left* of a node $p_j \in V(P)$, if $i < j$. Similarly, p_i is *strictly right* of p_j, if $i > j$.

Treewidth. Tree-decompositions generalize path-decompositions by replacing the path P with a tree T. A *tree-decomposition* is defined analogously to a path-decomposition, with the connectivity condition requiring that the bags containing any fixed vertex induce a connected subtree of T. Analogously, the *width* of a tree-decomposition is the size of its largest bag minus one. The *treewidth* $\mathrm{tw}(G)$ of G is the minimum width over all tree-decompositions of G.

We use the same notation for tree-decompositions as for path-decompositions, with the following additions. The *restriction* of a tree-decomposition (T, B) of a graph G to an induced subgraph G' of G is obtained from (T, B) by removing all vertices of $V(G) - V(G')$ from every bag. A tree-decomposition is called *redundant*, if some bag of this tree-decomposition is a subset of the bag of a neighbour. All other tree-decompositions are *non-redundant*. By fixing a root $r \in V(T)$, we obtain a *rooted tree-decomposition*. For a rooted tree-decomposition, we use G_t to refer to the subgraph of G induced by the set of vertices appearing in bags of the subtree rooted at some node t. This includes the vertices of $B(t)$. We write $\mathrm{children}(t)$ for the set of children of a node t in a rooted tree-decomposition.

On the weight of cliques. We introduce a way to measure the weight of sets based on clique-partitions that will be used in defining the following parameters. Let G be a graph and let $X \subseteq V(G)$ be a vertex set. For a clique-partition $\mathcal{P}_X$ of $G[X]$, the *weight* of a clique $C \in \mathcal{P}_X$ is defined as $\log(|C| + 1)$. The *weight* of the clique-partition $\mathcal{P}_X$ is the sum of the weights of its cliques. The *weight* of $G[X]$, and as a shorthand of X, is the minimum weight over all clique-partitions of $G[X]$. Given a fixed clique-partition $\mathcal{P}_X$ of $G[X]$, the *weight of* $G[X]$ *under* $\mathcal{P}_X$ is simply the weight of $\mathcal{P}_X$. Again, we use X as a shorthand instead of $G[X]$. If a clique-partition of a supergraph is given, it naturally restricts to a clique-partition of $G[X]$.

When only the relative quality of partitions matters, we may equivalently use the *multiplicative formulation* $\prod_{C \in \mathcal{P}_X}(|C| + 1)$ for the weight of a clique-partition $\mathcal{P}_X$. This formulation is equivalent for relative comparisons, since $\log(\Pi_{C \in \mathcal{P}_X}(|C| + 1)) = \sum_{C \in \mathcal{P}_X} \log(|C| + 1)$ and the logarithm is monotone.

Clique-partitioned treewidth. We now turn to parameters that refine treewidth by taking into account the internal structure of bags. We begin by formally introducing clique-partitioned treewidth as originally defined by Bläsius et al. [BKW23], which is the parameter this thesis focuses on. A *clique-partitioned tree-decomposition*, abbreviated as cp-tree-decomposition, of a graph G is a triplet (T, B, P), where (T, B) is a tree-decomposition and P is a function that assigns to each node $t \in V(T)$ a clique-partition of $G[B(t)]$. We say that P *partitions each bag into cliques*. The *clique-partitioned weight* of a bag $B(t)$ is weight of

(a) (b)

Figure 2.1: An example graph for which $\mathcal{P}$-flattened treewidth is strictly larger than gp-treewidth. Part (a) shows the graph together with a partition into two cliques. Part (b) presents a gp-tree-decomposition of this graph with gp-weight $2 \cdot 2 = 4$, using the multiplicative formulation of weight. For $\mathcal{P}$-flattened treewidth, any partition into two cliques yields adjacent cliques, and thus every bag in the resulting decomposition must contain all vertices, leading to a strictly larger weight. A case analysis shows that partitions into more than two cliques do not improve the situation.

$B(t)$ under $P(t)$. The *clique-partitioned weight* of a cp-tree-decomposition is the maximum clique-partitioned weight of a bag of this decomposition. We use the abbreviation cp-weight for the clique-partitioned weight. The *clique-partitioned treewidth* $\mathrm{cptw}(G)$ of a graph G, abbreviated as cp-treewidth, is the minimum cp-weight over all cp-tree-decompositions of G.

We define *clique-partitioned path-decompositions* (short cp-path-decompositions) and the *clique-partitioned pathwidth* $\mathrm{cppw}(G)$ of a graph G, abbreviated as cp-pathwidth, analogously by using an underlying path-decomposition instead of a tree-decomposition.

Globally-partitioned treewidth. As the next parameter, we introduce globally-partitioned treewidth, which differs from cp-treewidth in that the local partitions of the bags must be compatible and together form a global partition of the entire graph. A *globally-partitioned tree-decomposition*, abbreviated as gp-tree-decomposition, of a graph G is a triplet $(T, B, \mathcal{P})$, where (T, B) is a tree-decomposition and $\mathcal{P}$ is a clique-partition of the entire graph G. The *globally-partitioned weight* of a bag $B(t)$ is weight of $B(t)$ under $\mathcal{P}$. The *globally-partitioned weight* of a gp-tree-decomposition is the maximum globally-partitioned weight of a bag of this decomposition. We use the abbreviation gp-weight for the globally-partitioned weight. The *globally-partitioned treewidth* $\mathrm{gptw}(G)$ of a graph G, abbreviated as gp-treewidth, is the minimum gp-weight over all gp-tree-decompositions of G.

We may also *fix a clique-partition*. In this variant of gp-treewidth, only gp-tree-decompositions that respect the given clique-partition are considered.

It can be seen, that globally-partitioned treewidth is closely related to the parameter $\mathcal{P}$-flattened treewidth introduced by De Berg et al. [Ber+18] for $\kappa = 1$. In particular, both parameters are based on global partitions of the entire graph. The $\mathcal{P}$-flattened treewidth additionally contracts each clique into a single vertex before computing tree-decompositions, and can thus be viewed as form of weighted treewidth. This contraction ensures that, for every clique, either all or none of its vertices are contained in any given bag. Consequently, $\mathrm{gptw}(G)$ is always at most the $\mathcal{P}$-flattened treewidth of G, and the inequality is strict for some graphs (see Figure 2.1).

Tree-treewidth. Similarly, we define tree-treewidth. A *tree-treewidth tree-decomposition*, for short tt-tree-decomposition, is a triplet (T, B, D), where (T, B) is a tree-decomposition and D is a function that assigns to each node $t \in V(T)$ a tree-decomposition of $G[B(t)]$. The *tree-tree weight* of a bag $B(t)$ is the width of $D(t)$ and the *tree-tree weight* of a tt-tree-decomposition

is the maximum tree-tree-weight of a bag of this decomposition. Similar to before, we use the abbreviation tt-weight. The *tree-treewidth* $\mathrm{ttw}(G)$ of a graph G is the minimum tt-weight over all tt-tree-decompositions of G.

Tree-independence number. Next, we define the tree-independence number, a parameter that has been studied extensively in the literature [Yol18 | DMS24 | Dal+26]. Recall, that we define the *independence number* of a graph G to be the maximum size of an independent set in G. Then, the *independence number* of a bag $B(t)$ is the independence number of the subgraph $G[B(t)]$ induced by $B(t)$. The *independence number* of a tree-decomposition is the maximum independence number of a bag of this decomposition. The *tree-independence number* tree-$\alpha(G)$ of a graph G is the minimum independence number over all tree-decompositions of G.

Tree-clique-cover number. Analogously, we extend the clique-cover number to tree-decompositions. This parameter has also been used by Hilaire et al. [HMV26]. As defined earlier, the *clique-cover number* of a graph G is the minimum size of a clique-cover of G. Then, the *clique-cover number* of a bag $B(t)$ is the clique-cover number of the subgraph $G[B(t)]$ induced by $B(t)$. Then, the *clique-cover number* of a tree-decomposition is the maximum clique-cover number of a bag of this decomposition. The *tree-clique-cover number* tree-$\kappa(G)$ of a graph G is the minimum clique-cover number over all tree-decompositions of G.

2.3.2 An addendum to the definitions of parameters

We conclude this part with several definitions related to the considered parameters. First note, that all definitions introduced for tree-decompositions extend naturally to cp-, gp- and tt-tree-decompositions, as these are tree-decompositions augmented with additional structure. Whenever the type of weight is clear from the context or is irrelevant, we omit the prefixes cp, gp, or tt.

For algorithmic purposes like dynamic programming approaches, it is common to work with *nice tree-decompositions*. We define these for tree-decompositions and note that the concept extends directly to *nice cp-, gp- and tt-tree-decompositions*. A *nice tree-decomposition* (T, B) is a rooted tree-decomposition with root r such that $B(r) = \emptyset$, and every non-root node is one of the following types: A *leaf-node* is a node t with no children and $B(t) = \emptyset$. An *introduce-node* is a node t with exactly one child t' such that $B(t) = B(t') + v$, for some *introduced vertex* $v \notin B(t')$. A *forget-node* is a node t with exactly one child t' such that $B(t) = B(t') - v$, for some *forgotten vertex* $v \in B(t')$. A *join-node* is a node t with exactly two children t_1 and t_2 such that $B(t) = B(t_1) = B(t_2)$.

Finally, since this work is primarily concerned with the computation of these parameters, we formalize the corresponding decision problems. For instance, in the Treewidth problem, we are given a graph G and an integer k, and the task is to decide whether G admits a tree-decomposition of width at most k. The problems cp-Treewidth, gp-Treewidth, tt-Treewidth, Tree-Independence Number, and Tree-clique-cover number are defined analogously using the respective parameters and decomposition variants. Note, that cp-Treewidth and gp-Treewidth are defined with real-valued parameters.

3 Fundamental properties of tree-decompositions and weighted clique-partitions

In this chapter, we present several fundamental properties of clique-partitions and tree-decompositions that are used throughout this thesis. These results are well-known and have appeared in previous work. We state them for completeness and, where helpful, provide brief proof sketches.

3.1 Chosen fundamental properties of weighted clique-partitions

We begin by studying basic properties of clique-partitions. Most of the following results are either straightforward or have been established by Bläsius et al. [BKW23]. We first consider the effect of removing vertices.

Observation 3.1: *Let X be a vertex set and let $S \subseteq X$ be a subset of X. Then, the weight of S is at most the weight of X.*

Proof. Consider an optimal clique-partition $\mathcal{P}$ of X and use $w_{\mathcal{P}}$ for the weight with respect to $\mathcal{P}$. For every clique C^* of $\mathcal{P}[S]$, there exists a clique of $\mathcal{P}$ that is a superset of C^*. Since the logarithm is monotone, it follows that

$$\begin{aligned} w_{\mathcal{P}}(S) &= \sum_{C \in \mathcal{P}[S]} w_{\mathcal{P}}(C) \\ &\leq \sum_{C' \in \mathcal{P}} w_{\mathcal{P}}(C') \\ &= w_{\mathcal{P}}(X). \end{aligned}$$

■

This observation immediately extends to the corresponding parameters.

Corollary 3.2: *The cp-treewidth and gp-treewidth are monotone under taking subsets.*

Next, we consider the behaviour under unions of disjoint sets.

Observation 3.3: *Let X and Y be two disjoint vertex sets. Then, the weight of $X \cup Y$ equals the sum of the weights of X and Y.*

Proof. Combining an optimal clique-partition of X with an optimal clique-partition of Y yields a clique-partition of $X \cup Y$. Since the sets are disjoint, the partitions do not interact, and the weights add. ■

To gain further intuition on weights, we consider how they depend on the balance of clique sizes. Informally, less balanced partitions have smaller weight. As we are only interested in relative comparisons, we use the multiplicative formulation of weight.

Lemma 3.4 (Lemma 3 of [BKW23])**:** *Let $v, x, y, z \in \mathbb{N}_0$ such that $v + x = y + z$ and $v \geq x, y \geq z, z > x$. Then, $(v+1) \cdot (x+1) < (y+1) \cdot (z+1)$.*

This notion of balancedness extends to sequences.

Lemma 3.5 (Lemma 4 of [BKW23])**:** *Let $\langle s_1, \ldots, s_k \rangle$ and $\langle r_1, \ldots, r_\ell \rangle$ be different non-increasing sequences of natural numbers such that $2 \leq k \leq \ell$, $\sum_{i \in [k]} s_i = \sum_{i \in [\ell]} r_i$ and $s_i \geq r_i$ for all $i \in [k-1]$. Then, $\prod_{i \in [k]} (s_i + 1) < \prod_{i \in [\ell]} (r_i + 1)$.*

3.2 Chosen fundamental properties of tree-decompositions

We continue with several well-known properties of tree-decompositions. We begin by considering how cliques are represented by tree-decompositions.

Lemma 3.6 (Lemma 4 of [Bod98])**:** *Let $G = (V, E)$ be a graph, let (T, B) be a tree-decomposition of G, and let $C \subseteq V$ be a clique. Then, there exists a node $t \in V(T)$ such that $C \subseteq B(t)$.*

Next, we consider non-redundant decompositions and show that optimal tree-decompositions of bounded size exist. We state the result for a general notion of weight, assuming only monotonicity under taking subsets.

Lemma 3.7: *Let $G = (V, E)$ be a graph and let (T, B) be a tree-decomposition of G. Then, there exists a non-redundant tree-decomposition of G with the same weight as (T, B) and at most $|V|$ nodes.*

Proof Sketch. We obtain an non-redundant tree-decomposition by repeatedly contracting any node whose bag is a subset of the bag of a neighbour into this neighbour. This preserves the weight due to monotonicity under taking subsets. Using induction on $|V|$ and the fact that every leaf of a non-redundant tree-decomposition must contain a vertex not appearing elsewhere, one obtains the claimed bound on the number of nodes. A full proof can be found in [AZ21]. ∎

Since every path-decomposition is a tree-decomposition, the same result holds in that setting.

Corollary 3.8: *Let $G = (V, E)$ be a graph and let (P, B) be a path-decomposition of G. Then, there exists a non-redundant path-decomposition of G with the same weight as (P, B) and at most $|V|$ nodes.*

We now show that every tree-decomposition can be converted into a nice tree-decomposition without increasing the weight. Again, we state the result for a general notion of weight, assuming only monotonicity under taking subsets.

Lemma 3.9: *Let $G = (V, E)$ be a graph and let (T, B) be a tree-decomposition of G with width* tw*. Then, a nice tree-decomposition of G with the same weight as (T, B), width* tw*, and $\mathcal{O}(\mathrm{tw} \cdot |V|)$ nodes can be computed in time $\mathcal{O}(\mathrm{tw}^2 \cdot (|V| + |E|))$.*

Proof Sketch. To obtain a nice tree-decomposition, we address three issues. First, we ensure that all leaves have empty bags by adding appropriate child nodes. Second, we enforce that adjacent bags differ by at most one vertex by inserting paths of intermediate bags of length $\mathcal{O}(\text{tw})$. Third, we replace nodes of degree greater than two by binary trees. The resulting decomposition satisfies the desired properties. A full proof for cp-tree-decompositions and a short discussion on the origins of this proof can be found in [Gei23]. ∎

We now show that bags act as separators.

Lemma 3.10: *Let $G = (V, E)$ be a connected graph, let (T, B) be a non-redundant tree-decomposition of G and let $t \in V(T)$ be a non-leaf. Then, $B(t)$ is a separator of G.*

Proof. Since t is a non-leaf, it has at least two neighbours t' and t^*. By non-redundancy $B(t')$ contains some vertex a not in $B(t)$. Similarly, $B(t^*)$ contains some vertex b not in $B(t)$. Let T_1 and T_2 be a partition of $T - t$ into two components such that a is contained in some bag of T_1 and b is contained in some bag of T_2. By the connectivity-property of tree-decompositions, no vertex appears in bags of both T_1 and T_2. Hence, by edge coverage, no edge between vertices of different components exists and a and b are separated by $B(t)$. ∎

An analogous result holds for edges of the decomposition.

Lemma 3.11: *Let $G = (V, E)$ be a connected graph, let (T, B) be a non-redundant tree-decomposition of G and let tt' be an edge in T. Then, $B(t) \cap B(t')$ is a separator of G.*

Proof. First, note that by non-redundancy $B(t)$ contains some vertex a not in $B(t')$ and $B(t')$ contains some vertex b not in $B(t)$. We then proceed analogous to Lemma 3.10 but consider components of T with the edge tt', but not its endpoints, removed. ∎

We now turn to chordal graphs, which are of particular interest as they admit well-structured tree-decompositions. The following well-known lemma formalizes this property. A proof can be found in [Dvo15].

Lemma 3.12: *Let G be a chordal graph. Then, there exists an optimal tree-decomposition (T, B) of G such that, for each $t \in V(T)$, the bag $B(t)$ induces a clique.*

We now turn to a graph construction that enforces strong structural properties on the bags of any tree-decomposition. In particular, we show that this construction guarantees that one of two copies of a given graph is fully contained in every bag. This allows us to relate the tree-independence number to the classical independence number, and similarly for clique-cover variants. Our approach extends Observation 5.1 and Lemma 5.1 of [Dal+26] to all bags and to our setting.

Lemma 3.13: *Let $G = (V, E)$ be a graph and let H be constructed from two disjoint copies G_1 and G_2 of G by connecting each pair of vertices $x \in V(G_1), y \in V(G_2)$ with an edge. Let (T, B) be a non-redundant tree-decomposition of H, and let t be a node of T. Then, at least one of G_1 or G_2 is fully contained in $B(t)$, i. e. $V(G_1) \subseteq B(t)$ or $V(G_2) \subseteq B(t)$.*

Proof. We distinguish cases based on the structure of $B(t)$ and the position of t in T.

If T consist of a single node, then $B(t) = V(H)$ and the claim is immediate.

Assume now that T has at least two nodes. First, consider the case where t is a leaf with unique neighbour t'. If $B(t)$ already contains one of the copies G_1 or G_2, we are done. Otherwise, since (T, B) is non-redundant, there exists a vertex $v \in B(t) - B(t')$. W.l.o.g. let v be a vertex of G_1. Furthermore, let $u \in G_2$ be a vertex not in $B(t)$. Such a vertex exists, as

$B(t)$ does not contain all of G_2 by assumption. Then, the edge uv is not covered by any bag of the decomposition, since the vertex v appears only in $B(t)$, while u does not appear in $B(t)$. This contradicts the edge coverage property of tree-decompositions. Hence, the case of a leaf containing neither of G_1 or G_2 cannot occur.

It remains to consider the case where t is a non-leaf. Then, removing t from T yields at least two connected components. Let T_x be one such component and let T_y be the union of the remaining components. Let X and Y denote the sets of vertices that appear exclusively in bags of T_x and T_y, respectively, but not in $B(t)$. We note, that X and Y are disjoint, non-empty vertex sets that are not connected by an edge in H (see Lemma 3.10). Additionally, the union of X, Y and $B(t)$ is exactly the vertex set of H.

We distinguish cases based on how the set X intersects the vertex sets of G_1 and G_2.

If X contains vertices only from G_1, then no vertex of G_2 can lie in Y. This is, since every vertex of G_2 is adjacent to every vertex of G_1 and since X and Y are not connected by an edge. Hence, $B(t)$ must contain all vertices of G_2, as neither X nor Y contains any vertices from G_2. The case where X contains only vertices of G_2 is symmetric.

If X contains vertices from both G_1 and G_2, then Y cannot contain any vertex from either copy, by similar arguments. However, this implies $Y = \emptyset$, a contradiction. This completes the proof. ∎

We now use this result to relate parameters of H to those of G.

Lemma 3.14: *Let $G = (V, E)$ be a graph and let H be constructed from two disjoint copies G_1 and G_2 of G by connecting each pair of vertices $x \in V(G_1), y \in V(G_2)$ with an edge. Let (T, B) be a non-redundant tree-decomposition of H and let t be a node of T. Then, the independence number of $B(t)$ equals the independence number of G. Moreover, the clique-cover number of $B(t)$ equals the clique-cover number of G.*

Proof. By Lemma 3.13, the bag $B(t)$ contains all vertices of one copy, say G_1. Each vertex in $V(G_2) \cap B(t)$ is adjacent to all vertices of G_1, and thus $V(G_2) \cap B(t)$ forms a set of universal vertices with respect to G_1. Therefore, any independent set of $B(t)$ is either contained entirely in G_1 or entirely in $V(G_2) \cap B(t)$. Hence, the independence-number of $B(t)$ is exactly the independence number of G.

For clique-covers, observe that any clique-cover of G_1 can be extended to a clique-cover of $B(t)$ by assigning each vertex in $V(G_2) \cap B(t)$ to the corresponding clique in G_1. This does not increase the number of cliques in the cover, and thus the clique-cover number of $B(t)$ equals that of G. ∎

4 Comparing tree-decomposition based parameters

In this section, we compare cp-treewidth and gp-treewidth to several other tree-decomposition-based parameters such as treewidth and tree-independence number. When evaluating such parameters, three aspects are of interest: (1) *size:* the parameter value should be as small as possible, (2) *usefulness:* the parameter should enable the design of FPT-algorithms for a wide range of problems, (3) *computability:* the parameter should be efficiently computable. In this chapter, we focus on the first two aspects, as algorithms and hardness results for computing the other parameters are already well-documented in the literature (see for example [BK96 | Bon25 | Dal+26]). Algorithms and hardness results concerning the computation of the parameters cp-treewidth and gp-treewidth will be presented in later chapters.

The goal of this chapter is to establish two hierarchies. The first hierarchy orders the parameters according to their size. More precisely, we show that for a given tree-decomposition, the weight as measured by one parameter is smaller than the weight as measured by another. The second hierarchy orders the parameters according to their algorithmic potential. To this end, we analyse for which problems these parameters admit FPT-algorithms. In this chapter, we extend the comparisons established by Geißler [Gei23]. Accordingly, we largely follow their notation and several of their proof techniques, and we also reuse some of their results.

4.1 A hierarchy of parameter-sizes

When comparing the sizes of parameters, we consider two aspects. First, we establish a sequence of guaranteed upper bounds such that, for every tree-decomposition, the parameter value decreases at each step of the sequence. Second, we study the maximum possible gap between two parameters. To this end, we bound the smaller parameter as a function of the larger one.

4.1.1 Getting smaller in each step

In this subsection, we show that the considered parameters form a hierarchy in which the parameter value decreases in every step. Rather than restricting the comparison to optimal tree-decompositions, we prove a stronger statement that applies to all tree-decompositions. To this end, we use decomposition-wise comparisons. More precisely, we consider an arbitrary graph together with an arbitrary tree-decomposition and the additional structures required for the larger parameter (such as clique-partitions). We then show that the additional structures for the smaller parameter can be chosen so that its weight is at most the weight of the larger parameter. Since our parameters are defined as the minimum over all tree-decompositions and all corresponding additional structures, it follows directly that, for every graph, the smaller parameter is indeed at most as large as the larger one. In particular, given an optimal decomposition together with suitable structures for the larger parameter, we can construct structures for the smaller parameter whose weight is no larger.

Therefore, we show the following hierarchy.

Theorem 4.1: *Let G be a graph and let (T, B) be a tree-decomposition of G. Then, there exists a clique-partition $\mathcal{P}$ of G such that, for every such $\mathcal{P}$, there exists a function P mapping each node t of T to some clique-partition of $G[B(t)]$ with the following property.*
Let tree-α *denote the independence number of (T, B), let* tree-κ *denote the clique-cover number of (T, B), let* cptw *denote the cp-weight of (T, B, P), let* gptw *denote the gp-weight of $(T, B, \mathcal{P})$, and let* tw *denote the width of (T, B). Then,* tree-$\alpha \leq$ tree-$\kappa \leq$ cptw $\leq$ gptw $\leq$ tw $+1$.

To prove this theorem we consider the individual inequalities. As the first parameter, we analyse the classical notion of treewidth. Our goal is to show that it is the largest among the parameters under consideration. We therefore begin by comparing it to gp-treewidth.

Lemma 4.2: *Let G be a graph and let (T, B) be a tree-decomposition of G with width* tw*. Then, there exists a clique-partition $\mathcal{P}$ of G such that the gp-tree-decomposition of $(T, B, \mathcal{P})$ has gp-weight at most than* tw $+1$.

Proof. We construct such a clique-partition explicitly. Here, we use the trivial partition $\mathcal{P}$ of G into single-vertex cliques. Let t be a node of the resulting gp-tree-decomposition $(T, B, \mathcal{P})$, and let $\mathcal{P}[t]$ denote the restriction of $\mathcal{P}$ to the bag $G[B(t)]$. Then, the gp-weight of t is

$$\begin{aligned}
&\sum_{C \in \mathcal{P}[t]} \log(|C| + 1) \\
&= \sum_{C \in \mathcal{P}[t]} \log(1 + 1) \\
&= \sum_{C \in \mathcal{P}[t]} 1 \\
&= \sum_{v \in B(t)} 1 \\
&\leq \text{tw} + 1.
\end{aligned}$$

Here, we used that, for the partition into single-vertex cliques, summing over all cliques is equivalent to summing over all vertices in the bag. Since this bound also holds for the bag with maximum gp-weight, the gp-weight of $(T, B, \mathcal{P})$ is at most tw $+1$. Note, that choosing a more refined clique-partitions can only decrease the gp-weight. ∎

It is straightforward to verify that this bound is tight. For example, consider edgeless graphs, for which the above clique-partition is optimal.

We now extend the hierarchy to the next smallest parameter. In particular, we consider cp-treewidth.

Lemma 4.3: *Let G be a graph and let $(T, B, \mathcal{P})$ be a gp-tree-decomposition of G with gp-weight* gptw*. Then, there exists a function P that maps each node t to a clique-partition of $G[B(t)]$ such that the cp-tree-decomposition (T, B, P) has cp-weight at most* gptw*.*

Proof. We construct a clique-partition for each bag by restricting the global partition to the vertices of the bag. Formally, for every node $t \in V(T)$, we define $P(t)$ as the restriction of $\mathcal{P}$ to the vertex set $B(t)$. With this choice, the cp-weight of each bag equals its gp-weight. Consequently, the cp-weight of (T, B, P) is exactly the gp-weight of $(T, B, \mathcal{P})$. As before, selecting more suitable clique-partitions for the bags can only decrease the cp-weight. ∎

Again, it is straightforward to verify that this bound is tight. In particular, for complete graphs all vertices are contained in a single bag (see Lemma 3.6), and therefore the local and global partitions coincide.

In the next step, we consider the tree-clique-cover number.

Lemma 4.4: *Let G be a graph and let (T, B, P) be a cp-tree-decomposition of G with cp-weight* cptw. *Then, the clique-cover number of each bag in (T, B, P) is at most* cptw.

Proof. We construct a clique-cover of the required size for each bag by reusing the given clique-partition. Since every clique contributes at least 1 to the cp-weight, it follows that the clique-cover number of each bag is at most cptw. As before, selecting more suitable clique-covers for the bags can only decrease the clique-cover number. ∎

It is straightforward to verify that this bound is tight. For example, consider edgeless graphs, for which the above clique-cover is optimal.

As the final parameter, we consider the tree-independence number. This lemma concludes the proof of Theorem 4.1.

Lemma 4.5: *Let G be a graph and let (T, B) be a tree-decomposition of G with tree-clique-cover number* tree-κ. *Then, the independence number of each bag in (T, B, P) is at most* tree-κ.

Proof. Let $B(t)$ be a bag and let $I \subseteq B(t)$ be an independent set in $G[B(t)]$. By definition, each vertex of I must lie in a different clique of a minimum clique-cover of $G[B(t)]$. Choosing I to be a maximum independent set in $G[B(t)]$ yields the desired statement. ∎

For chordal graphs, the two parameters clearly coincide, since each bag induces a clique (see Lemma 3.12), and a clique-cover consisting of a single clique is optimal.

To illustrate that not all parameters fit into such a clear hierarchy, we now consider *tree-treewidth*. Here, we will see that this parameter is bounded by treewidth, but that we are unable to compare it to the other parameters in the hierarchy.

Lemma 4.6: *Let G be a graph and let (T, B) be a tree-decomposition of G with width* tw. *Then, there exists a function D that maps each node t to a tree-decomposition of $G[B(t)]$ such that the tt-tree-decomposition (T, B, D) has tt-weight at most* tw.

Proof. We construct such a tree-decomposition for each bag. For this purpose, we use the trivial tree-decomposition consisting of a single bag. With this choice, the tt-weight of each bag equals its width. Consequently, the tt-weight of (T, B, D) equals the width of (T, B). As before, choosing more suitable tree-decompositions for the bags can only decrease the tt-weight. ∎

However, tree-treewidth is incomparable to the other parameters. We show that there exists a graph family in which tree-treewidth is smaller, and another in which it is larger.

Lemma 4.7: *There exists an infinite family of graphs $\mathcal{G}$ such that a graph $G \in \mathcal{G}$ with n vertices, has tree-treewidth in $\mathcal{O}(1)$, gp-treewidth in $\Omega(\sqrt{n})$, cp-treewidth in $\Omega(\sqrt{n})$, tree-clique-cover number in $\Omega(\sqrt{n})$, and tree-independence number in $\Omega(\sqrt{n})$.*

Proof. We consider the family of grids with equal number of rows and columns. Let G be such a grid with $\sqrt{n}$ rows and $\sqrt{n}$ columns. We verify that the parameters have the claimed values. Remember, that the treewidth of G is $\sqrt{n}$ [Bod98]. A tree-decomposition that processes the grid column by column achieves this width. In this decomposition, each bag induces a tree. Hence, this decomposition can be extended to a tt-tree-decomposition with tt-weight 1.

Now let (T, B) be a tree-decomposition of G. We show that the weight of (T, B) is at least $\Omega(\sqrt{n})$ for the other three parameters. Since G has treewidth $\sqrt{n}$, there must be a node $t \in V(T)$ whose bag has size at least $\sqrt{n} + 1$. The graph G contains no cliques of size at least three, and therefore the same holds for $G[B(t)]$. Consequently, every clique-cover of $G[B(t)]$ contains at least $\frac{\sqrt{n}+1}{2}$ cliques. Thus, the gp-weight, the clique-cover number, and the cp-weight of this bag $B(t)$ are in $\Omega(\sqrt{n})$. Finally, consider the tree-independence number. In any tree-decomposition of G there exists a bag that contains a vertex from each row or from each column (Claim 88A of [Bod98]). Taking a vertex from every second row (or every second column) yields an independent set, since edges occur only between adjacent rows (or columns). Thus, this bag has independence number at least $\frac{\sqrt{n}}{2}$.

Since this holds for every tree-decomposition, the three parameters all are in $\Omega(\sqrt{n})$. ∎

The last lemma shows that there is an infinite family for which tree-treewidth is smaller than the remaining parameters. We now show the opposite direction by constructing an infinite family for which tree-treewidth is larger than the remaining parameters.

Lemma 4.8: *There exists an infinite family of graphs $\mathcal{G}$ such that a graph $G \in \mathcal{G}$ with n vertices, has tree-treewidth in $\Theta(n)$, gp-treewidth in $\mathcal{O}(\log(n))$, cp-treewidth in $\mathcal{O}(\log(n))$, tree-clique-cover number in $\mathcal{O}(\log(n))$, and tree-independence number in $\mathcal{O}(\log(n))$.*

Proof. We consider the family of complete graphs. Let G be such a graph with n vertices. By Lemma 3.6, every tree-decomposition of G must contain one bag that includes the all vertices of the graph. Therefore, the tree-treewidth of G is $n - 1$.

For the remaining three parameters, note that by Lemma 4.3, Lemma 4.4, and Lemma 4.5, it suffices to bound the gp-treewidth. Consider the gp-tree-decomposition consisting of a single bag together with a clique-partition that contains the entire vertex set as one clique. The gp-weight of this decomposition is in $\mathcal{O}(\log(n))$. ∎

4.1.2 Bounding the gaps between parameters

We now investigate by how much the parameters in our hierarchy may differ. More precisely, we study which function of the smaller parameter is sufficient to bound the larger one. We formalize this using the following notion. We say that a parameter τ is *f-bounded in* a parameter ρ, if f is a function such that τ is smaller than $f(\rho)$ in a decomposition-wise comparison for all graphs.

As a first step, we again consider the two clique-partition-based parameters and treewidth. To relate them, we use the following folklore auxiliary lemma, which compares sums and products of sequences.

Lemma 4.9 (Folklore, for a proof see Lemma 3.1 of Geißler [Gei23])**:** *Let $x_1, \ldots, x_k$ be a sequence of length $k \in \mathbb{N}$ with $x_i \geq 2$ for each $i \in [k]$. Then, $\sum_{i=1}^{k} x_i \leq \prod_{i=1}^{k} x_i$.*

Using this lemma, we obtain the following bound.

Lemma 4.10: *Let $G = (V, E)$ be a graph and let $\mathcal{P}$ be a clique-partition of G with $w = \sum_{C \in \mathcal{P}} \log(|C| + 1)$. Then, $|V| \leq 2^w$.*

Proof. First, note that the number of vertices equals the sum of the clique sizes. Hence,

$$\begin{aligned} |V| &= \sum_{C \in \mathcal{P}} |C| \\ &\leq \sum_{C \in \mathcal{P}} (|C| + 1) \\ &\leq \prod_{C \in \mathcal{P}} (|C| + 1) \\ &= 2^{\sum_{C \in \mathcal{P}} \log(|C|+1)} \\ &= 2^w \end{aligned}$$

The second inequality follows from Lemma 4.9, since each clique has size at least one and thus $|C| + 1 \geq 2$. ∎

This immediately implies analogous statements for gp-treewidth and cp-treewidth. Hence, treewidth is 2^x-bounded in both gp-treewidth and cp-treewidth.

Corollary 4.11: *Let G be a graph and let $(T, B, \mathcal{P})$ be a gp-tree-decomposition of G with gp-weight* gptw. *Then, (T, B) has width at most 2^{gptw}.*

Corollary 4.12: *Let G be a graph and let (T, B, P) be a cp-tree-decomposition of G with cp-weight* cptw. *Then, (T, B) has width at most 2^{cptw}.*

Both bounds are asymptotically tight, as witnessed by complete graphs.
Next, we show that gp-treewidth is 2^x-bounded in cp-treewidth.

Lemma 4.13: *Let G be a graph and let (T, B, P) be a cp-tree-decomposition of G with cp-weight* cptw. *Then, there exists a clique-partition $\mathcal{P}$ of G such that the gp-tree-decomposition of $(T, B, \mathcal{P})$ has gp-weight at most $2^{\text{cptw}} + 1$.*

Proof. By Lemma 4.2, the tree-decomposition (T, B) can be extended to a gp-tree-decomposition whose gp-weight is at most one more than the width of (T, B). By Corollary 4.12, this width is at most 2^{cptw}. ∎

To see that this bound is asymptotically tight, consider the following graph family.

Lemma 4.14: *There exists an infinite family of graphs $\mathcal{G}$ such that a graph $G \in \mathcal{G}$ with n vertices, has cp-treewidth in $\mathcal{O}(\log(\log(n)))$ and gp-treewidth in $\Omega(\log(n))$.*

Proof. We adapt the proof of Lemma 1 of Bläsius et al. [BKW23] to the setting of gp-treewidth. We choose $\mathcal{G}$ as follows. For every $h \in \mathbb{N}_+$, we add the following graph G_h to $\mathcal{G}$. G_h is a complete binary tree of height h, where all paths from a leaf to the root are connected to form a clique. Since the graph contains $\Theta(2^h)$ vertices, we have $h \in \Theta(\log(n))$.

We first construct a cp-tree-decomposition (T, B, P) of G_h with cp-weight $\mathcal{O}(\log(\log(n))$. We choose T to be a path and create one bag for each path from a leaf to the root. Since each bag consists of a single clique, this cp-tree-decomposition has cp-weight $\log(h + 1) \in \mathcal{O}(\log(\log(n)))$.

Let $(T, B, \mathcal{P})$ be a gp-tree-decomposition of G_h. By simple induction argument, we can see that there exists a path from a leaf to the root in G_h whose vertices are assigned to pairwise distinct cliques by $\mathcal{P}$. However, such a path forms a clique and thus is entirely contained in at least one bag by Lemma 3.6. This bag therefore contains at least h cliques and hence has gp-weight at least $h \cdot \log(1 + 1) \in \Omega(\log(n))$. ∎

Finally, we consider the tree-independence number and the tree-clique-cover number. In contrast to the previous parameters, there exists no function f such that cp-treewidth is f-bounded in either of those parameters. We say that cp-treewidth is *unbounded* in the tree-independence number and the tree-clique-cover number. This can be seen by considering complete graphs.

Lemma 4.15 (Extension of Lemma 3.12 of Geißler [Gei23])**:** *There exists an infinite family of graphs $\mathcal{G}$ such that a graph $G \in \mathcal{G}$ with n vertices, has tree-independence number in $\mathcal{O}(1)$, tree-clique-cover number in $\mathcal{O}(1)$, and cp-treewidth in $\Omega(\log(n))$.*

It remains to analyse how the tree-clique-cover number and the tree-independence number are related. Similar to before, there exist graphs where the tree-independence number is constant while the tree-clique-cover number is unbounded.

Lemma 4.16: *There exists an infinite family of graphs $\mathcal{G}$ such that, for each $r \in \mathbb{N}_+$, there exists a graph $G \in \mathcal{G}$ with tree-independence number in $\mathcal{O}(1)$ and tree-clique-cover number exactly r.*

Proof. We define the family $\mathcal{G}$ as follows. Let G be a triangle-free graph with chromatic number r. Such a graph exists for every r [Erd59]. In particular, its complement $\overline{G}$ contains no $3K_1$ as an induced subgraph. We construct a graph H by taking two disjoint copies of $\overline{G}$, denoted by $\overline{G_1}$ and $\overline{G_2}$, and connecting each pair of vertices $x \in V(\overline{G_1}), y \in V(\overline{G_2})$ with an edge. We let $\mathcal{G}$ be the family of all such graphs H.

We first analyse the independence number and the clique-cover number of $\overline{G}$. Since $\overline{G}$ contains no $3K_1$, every independent set has size at most 2. Moreover, since G has chromatic number r, the minimum clique-cover of the complement $\overline{G}$ has size exactly r. By Lemma 3.14, each bag in a tree-decomposition of H inherits these properties from $\overline{G}$. Consequently, the tree-independence number is bounded by a constant, while the tree-clique-cover number is exactly r. ∎

In summary, we have shown that cp-treewidth and gp-treewidth can be at most logarithmically smaller than treewidth, and that this logarithmic improvement can be distributed arbitrarily between the two steps of the hierarchy. Moreover, the tree-independence number and the tree-clique-cover number may remain constant while the other parameters grow with the size of the graph.

4.2 Comparing the algorithmic potential of parameters

Besides the size of parameters, their algorithmic potential is also of interest. In particular, we ask for which problems fixed-parameter tractable algorithms exist with respect to a given parameter. To study this question, we consider FPT-reductions from graph problems parameterized by one parameter to the same problem parameterized by another. Such reductions allow us to translate between parameters and thereby apply existing algorithms.

4.2.1 Treewidth, clique-partitioned treewidth, and globally-partitioned treewidth have the same algorithmic potential

We begin by establishing pairwise reductions between the variants of a general graph problem parameterized by treewidth, gp-treewidth or cp-treewidth. Here, we may use the size-bounds established in the previous subsection because the parameters influence only the running time of the algorithm, but not the correctness or quality of the solution.

These reductions allow us to transfer the W[t]-hardness results from one variant to another.

Theorem 4.17 (Extension of Theorem 4.3 of Geißler [Gei23])**:** *Let $t > 0$ be an integer. Let Π_{tw} be a graph problem parameterized by treewidth, let Π_{cptw} be the same graph problem parameterized by cp-treewidth and let Π_{gptw} be the same graph problem parameterized by gp-treewidth. Then, the following statements are equivalent.*

- Π_{tw} *is W[t]-hard.*
- Π_{cptw} *is W[t]-hard.*
- Π_{gptw} *is W[t]-hard.*

Proof. To prove this theorem, we provide the following three FPT-reductions which closely mirror the reductions established in Corollary 4.1 and 4.2 of Geißler [Gei23]. (1) a reduction from Π_{gptw} to Π_{cptw}, (2) a reduction from Π_{cptw} to Π_{tw}, and (3) a reduction from Π_{tw} to Π_{gptw}.

We begin with some general observations that apply to all three reductions. First, note that instances of Π_{tw} Π_{cptw} and Π_{gptw} differ only in the parameter and the given decomposition. In particular, the underlying graph and the problem statement remain unchanged. Therefore, each reduction only needs to transform the type of tree-decomposition associated with the first parameter into the type associated with the second, while ensuring that the second parameter is f-bounded, for some function f, in the first one. It remains to verify that each such reduction can be carried out in FPT-time with respect to the relevant parameter. Correctness then follows immediately, since neither the graph nor the problem statement is altered.

We now describe the individual reductions. For (1), the reduction from Π_{gptw} to Π_{cptw}, we use Lemma 4.3, which bounds the cp-weight of a decomposition by the gp-weight of this decomposition. This is done by interpreting the global partition as a local one. For (2), the reduction from Π_{cptw} to Π_{tw}, we use Corollary 4.12, which ensures that the width of a decomposition is 2^x-bounded in the cp-weight of this decomposition. This is done by ignoring the partition of the graph. Finally, for (3), the reduction from Π_{tw} to Π_{gptw}, we use Lemma 4.2, which bounds the gp-weight of a decomposition by one more than the width of this decomposition. This is done by considering the partition into single-vertex-cliques. All three reductions can be applied in FPT-time for the relevant parameter, since we only adjust the parameter and apply a simple modification to the decomposition. This establishes the required reductions and thus proves the theorem. ∎

Similarly, these reductions transfer results on the containment in complexity classes from one variant to another.

Theorem 4.18 (Extension of Corollary 4.4 of Geißler [Gei23])**:** *Let $t > 0$ be an integer. Let Π_{tw} be a graph problem parameterized by treewidth, let Π_{cptw} be the same graph problem parameterized by cp-treewidth and let Π_{gptw} be the same graph problem parameterized by gp-treewidth. Then, the following statements are equivalent.*

- Π_{tw} *contained in the class W[t].*
- Π_{cptw} *contained in the class W[t].*
- Π_{gptw} *contained in the class W[t].*

Furthermore, the following statements are equivalent.

- Π_{tw} *contained in the class FPT.*
- Π_{cptw} *contained in the class FPT.*
- Π_{gptw} *contained in the class FPT.*

Informally, these results show that the three parameters are equivalent with respect to which graph problems admit an FPT-algorithm. However, the concrete running times of these algorithms may still differ. A simple approach is to apply one of the reductions above and then run an existing algorithm for the target parameter. Note that this may increase the dependence on the parameter in the running time bound because the parameter value changes according to the corresponding bound. Since many algorithms for treewidth are already known, the most relevant reductions are those from gp-treewidth or cp-treewidth to treewidth.

Corollary 4.19 (Corollary 4.5 of Geißler [Gei23])**:** *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of cp-weight* cptw *and width* tw*, and let* (I, cptw) *be an instance of a graph problem* Π_{cptw}*. Assume there exists an traditional FPT-algorithm for* Π_{tw} *with running time* $f(\text{tw}) \cdot q(|V|)$*, where* f *is a computable function and* q *a polynomial. Then, an FPT-algorithm deciding* (I, cptw) *in* $f(2^{\text{cptw}}) \cdot q(|V|)$ *exists. If a gp-tree-decomposition of gp-weight* gptw *and width* tw *is given with the corresponding instance* (I, gptw)*, then an FPT-algorithm deciding* (I, gptw) *in* $f(2^{\text{gptw}}) \cdot q(|V|)$ *exists.*

4.2.2 The tree-independence number, the tree-clique-cover number, and tree-treewidth admit fewer algorithms and stronger hardness results

A natural question is whether the previous equivalence of algorithmic potential also extends to the tree-independence number, the tree-clique-cover number and the tree-treewidth. A first indication that this is not the case comes from the graph classes identified earlier in which these parameters remain constant while the treewidth grows. Consequently, the treewidth is unbounded in these parameters. Thus, the relationship is inherently asymmetric: In one direction, algorithms transfer between parameters, while in the other direction this transfer fails for certain problems.

We first consider the tree-independence number.

Lemma 4.20: *Let* Π_{tw} *be a graph problem with a given tree-decomposition that is parameterized by the width* tw *of this tree-decomposition. Let* $\Pi_{\text{tree-}\alpha}$ *be a graph problem with a given tree-decomposition that is parameterized by the independence number* tree-α *of this tree-decomposition. Then, there exists a FPT-reduction for the parameter treewidth from* Π_{tw} *to* $\Pi_{\text{tree-}\alpha}$ *such that the given tree-decomposition of width* tw *is transformed into a tree-decomposition of independence number* tree-$\alpha \leq \text{tw} + 1$.

Proof. We use a reduction similar to those in the proof of Theorem 4.17. As seen in this proof, it suffices to bound the independence number by the treewidth. We reduce Π_{tw} to $\Pi_{\text{tree-}\alpha}$ by using the same tree-decomposition as before. Since every bag has size at most tw +1, the independence number of each bag is also at most tw +1. Hence, the independence number of this decomposition is at most tw +1. ∎

We then continue with the tree-clique-cover number.

Lemma 4.21: *Let* Π_{tw} *be a graph problem with a given tree-decomposition that is parameterized by the width* tw *of this tree-decomposition. Let* $\Pi_{\text{tree-}\kappa}$ *be a graph problem with a given tree-decomposition that is parameterized by the tree-clique-cover number* tree-κ *of this tree-decomposition. Then, there exists a FPT-reduction for the parameter treewidth from* Π_{tw} *to* $\Pi_{\text{tree-}\kappa}$ *such that the given tree-decomposition of width* tw *is transformed into a tree-decomposition of clique-cover number* tree-$\kappa \leq$ tw $+1$.

Proof. We use a reduction similar to those in the proof of Theorem 4.17. As seen in this proof, it suffices to bound the tree-clique-cover number by the treewidth. We reduce Π_{tw} to $\Pi_{\text{tree-}\kappa}$ by using the trivial clique-cover of single-vertex cliques. Since every bag has size at most tw $+1$, the clique-cover number of each bag is also at most tw $+1$. Hence, the clique-cover number of this decomposition is at most tw $+1$. ∎

We now consider tree-treewidth and obtain a similar reduction.

Lemma 4.22: *Let* Π_{tw} *be a graph problem with a given tree-decomposition that is parameterized by the width* tw *of this tree-decomposition. Let* Π_{ttw} *be a graph problem with a given tt-tree-decomposition that is parameterized by the tt-weight* ttw *of this tt-tree-decomposition. Then, there exists a FPT-reduction for the parameter treewidth from* Π_{tw} *to* Π_{ttw} *such that the given tree-decomposition of width* tw *is transformed into a tt-tree-decomposition of tt-weight* ttw $\leq$ tw $+1$.

Proof. Again, we use a reduction similar to those in the proof of Theorem 4.17. As seen in this proof, it suffices to bound the tree-treewidth by the treewidth. We reduce Π_{tw} to Π_{ttw} by assigning, to each bag of the given tree-decomposition, the trivial tree-decomposition consisting of a single bag. Then, the resulting tt-tree-decomposition has tt-weight at most tw $+1$. ∎

Informally, these reductions ensure that any algorithm using the tree-independence number, tree-clique-cover number, or tree-treewidth can be transformed into an algorithm for treewidth. Furthermore, hardness results for treewidth translate to hardness results for the tree-independence number, the tree-clique-cover number and the tree-treewidth. In particular, the W[t]-hardness of Π_{tw} implies the W[t]-hardness of $\Pi_{\text{tree-}\alpha}$, $\Pi_{\text{tree-}\kappa}$, and Π_{ttw}. Analogously, containment in W[t] for one of $\Pi_{\text{tree-}\alpha}$, $\Pi_{\text{tree-}\kappa}$, and Π_{ttw} implies containment in W[t] for Π_{tw} as well as containment in FPT for one of $\Pi_{\text{tree-}\alpha}$, $\Pi_{\text{tree-}\kappa}$, and Π_{ttw} implies containment in FPT for Π_{tw}.

Next, we show that the reverse directions of the above reductions and results do not hold, assuming $\mathcal{P} \neq \mathcal{NP}$. We begin with the tree-independence number as well as the tree-clique-cover number and identify several problems that are in FPT for treewidth (and therefore also for cp-treewidth and gp-treewidth), but not for the tree-independence number and the tree-clique-cover number, unless $\mathcal{P} = \mathcal{NP}$.

Lemma 4.23: *If* $\mathcal{P} \neq \mathcal{NP}$*, then there exists no FPT-algorithm solving* DominatingSet, HamiltonCycle, HamiltonPath *or* MaxCut *on chordal graphs when parameterizing by either the tree-independence number or the tree-clique-cover number.*

Proof. Both the tree-independence number and the tree-clique-cover number of chordal graphs is 1, since chordal graphs admit a tree-decomposition in which every bag forms a clique (see Lemma 3.12). Additionally, all of the listed problems are known to be $\mathcal{NP}$-complete on chordal graphs [BJ82 | BB86 | BJ00]. Hence, an FPT-algorithm parameterized by either parameter would imply an polynomial-algorithm for these problems on chordal graphs, contradicting $\mathcal{P} \neq \mathcal{NP}$. ∎

For comparison, we note that these problems are fixed-parameter tractable when parameterized by treewidth for general graphs and thus also for chordal graphs.

Lemma 4.24 ([Cyg+15 | BCKN15 | BJ00])**:** *There exists an FPT-algorithm solving* DominatingSet, HamiltonCycle, HamiltonPath *and* MaxCut *when parameterizing by treewidth.*

Similarly for tree-treewidth, there also are problems that admit FPT-algorithms with respect to treewidth but not with respect to tree-treewidth.

Lemma 4.25: *If $\mathcal{P} \neq \mathcal{NP}$, then there exists no FPT-algorithm solving* DominatingSet, HamiltonCycle *or* HamiltonPath *on grid graphs when parameterizing by tree-treewidth.*

Proof. As observed earlier, grid graphs have tree-treewidth 1 because they admit a tree-decomposition in which every bag induces a tree. Additionally, all of the listed problems are known to be $\mathcal{NP}$-complete on grid graphs [IPS82 | CCJ90]. Hence, an FPT-algorithm parameterized by tree-treewidth would imply an polynomial-algorithm for these problems on grid graphs, contradicting $\mathcal{P} \neq \mathcal{NP}$. ∎

Lemma 4.26 ([Cyg+15 | BCKN15 | BJ00])**:** *There exists an FPT-algorithm solving* DominatingSet, HamiltonCycle *and* HamiltonPath *when parameterizing by treewidth.*

In summary, treewidth, cp-treewidth, and gp-treewidth have the same algorithmic potential, whereas the tree-independence number, the tree-clique-cover number, and the tree-treewidth admit strictly fewer FPT-algorithms and strictly stronger hardness results.

5 Algorithms for clique-partitioned Treewidth

It has been shown that, given a cp-tree-decomposition of small weight, a wide range of FPT-algorithms parameterized by cp-treewidth exist [Gei23]. However, no method for computing cp-tree-decompositions of small weight is known so far. In this chapter, we address this gap by presenting two algorithms for computing cp-tree-decompositions of small weight. We first introduce an XP-algorithm that computes the cp-treewidth of a graph exactly by enumerating all separators of small weight. As the running time of this approach is insufficient for many use cases, we then present an FPT-algorithm that computes a $4 \cdot k + 3$-approximation of cp-treewidth by considering balanced separators. Throughout this chapter, we use $n = |V|$ for the input graph $G = (V, E)$.

5.1 An XP-algorithm for clique-partitioned Treewidth

In this section, we present an exact XP-algorithm for verifying whether the cp-treewidth of a graph is at most a given real number k. Our approach adapts the algorithm introduced by Arnborg et al. [ACP87] to the setting of cp-treewidth.

This algorithm is inspired by the observation that the bags of a cp-tree-decomposition form separators of the graph with weight at most k. Our approach inverts this perspective. Instead of starting from a tree-decomposition, we enumerate all separators of weight at most k and use them as candidates for bags of a cp-tree-decomposition. Such separators can be enumerated in XP-time by scanning all vertex sets of size at most 2^k, which is the maximum possible size for sets of weight k by Corollary 4.12, and testing whether they are separators and whether their weight is at most k. Given a separator S of weight at most k, the algorithm must then verify whether S can serve as a bag of a cp-tree-decomposition. That is, we must check whether the connected components induced by S admit cp-tree-decompositions of weight at most k that can be combined with the bag defined by S. We formalize this requirement in our first lemma. Based on this structural characterization, one could design a recursive algorithm that checks this condition independently for each connected component induced by a separator. However, such a recursive approach would require enumerating all separators of each connected component, leading to an exponential running time. To overcome this issue, we introduce a second lemma that shows how, for a connected component, the existence of the needed decomposition can be determined using only separators of the original graph. More specifically, the lemma checks for the existence of a family of closely related separators such that the connected components induced by these separators, each admitting a compatible cp-tree-decomposition of weight at most k, collectively cover the current connected component. Finally, we combine these two structural insights to obtain our algorithm. The algorithm is formulated as a dynamic program that records, for each connected component induced by a separator of weight at most k, whether the condition of the second lemma is satisfied. This allows the algorithm to verify the existence of a cp-tree-decomposition of weight at most k without explicitly recursing on all separators of subgraphs.

We begin by presenting the first lemma, which characterizes when a separator can be used as a bag. This is done by specifying the structural requirements on the connected components it induces. To describe how a bag can be combined with a cp-tree-decomposition, we introduce the following notation. We say that a cp-tree-decomposition (T, B, P) has an *interface* I if there exists a node $t \in V(T)$ such that $B(t) = I$.

Lemma 5.1: *Let $G = (V, E)$ be a graph and let k be a positive real number. Then the following statements are equivalent:*

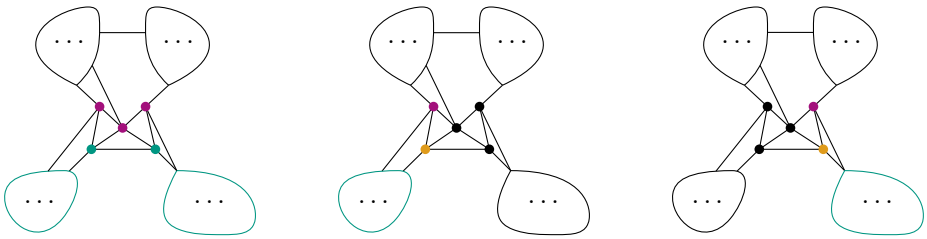
1. *G has cp-treewidth at most k.*
2. *There exists a separator $S \subseteq V$ of G with weight at most k such that, for every connected component C of $G - S$, the graph $G[C \cup S]$ admits a cp-tree-decomposition of weight at most k with interface S.*

Proof. We first show the implication from (1) to (2). Since the cp-treewidth of G is at most k, there exists a non-redundant cp-tree-decomposition (T, B, P) of G with weight k. We now select a bag to serve as the separator and then construct the required partial tree-decompositions using a modified variant of this decomposition. Let $t \in V(T)$ be a non-leaf node. For each neighbour t' of t in (T, B, P), we insert a copy of t between t and t'. This yields a cp-tree-decomposition (T', B', P') of the same weight. Now let $S = B'(t)$. Then, S is a separator of G (see Lemma 3.10) and, for each connected component of $G - S$, the corresponding connected component of $T' - t$ forms a cp-tree-decomposition with interface S and weight at most k.

Next, we show the implication from (2) to (1). To this end, we construct a cp-tree-decomposition of G with weight k by combining the given partial decompositions. We start with a root bag whose content is exactly the separator S. For each connected component C of $G-S$, we attach a cp-tree-decomposition of $G[C \cup S]$ with weight at most k and interface S as a descendant of the root. This is done by connecting some bag whose content is S to the root bag. The vertex coverage, edge coverage and connectivity conditions of tree-decompositions follow directly from the defining properties of separators and the correctness of the partial decompositions. Finally, the weight of the resulting cp-tree-decomposition is at most k, since both the separator S and all partial decompositions have weight at most k. ∎

At this stage, an algorithm could verify property (2) of this lemma by recursively searching for cp-tree-decompositions of each connected component. We now introduce the following condition, visualized in Figure 5.1, that enables a different and more efficient verification of whether the connected components admit the necessary tree-decompositions. Informally, to verify for a connected component C induced by a separator S whether the graph $G[C \cup S]$ admits a cp-tree-decomposition with the required structure, we must check for the existence of a family of separators with the following properties. Each separator in this family consist of a subset of S together with one additional vertex. Moreover, the connected components induced by this family that admit suitable well-structured cp-tree-decompositions together cover the all vertices of C. Clearly, this condition is easier to verify, since few separators are related to S in this manner. Consequently, all such separators and all well-formed connected components induced by them can be enumerated in XP-time.

Lemma 5.2: *Let $G = (V, E)$ be a graph and let k be a positive real number. Let $S \subseteq V$ be a separator of G with weight at most k and let C be a connected component of $G - S$ containing at least two vertices. Then, the following statements are equivalent.*



(a) (b) (c)

Figure 5.1: An example of the structures described in Lemma 5.2 is shown. In part (a), we depict a graph G together with the initial separator S (coloured purple) and the selected connected component C (coloured green). Parts (b) and (c) illustrate the same graph with the two separators (coloured purple in the intersection with S and orange for the added vertex) S_1 (in b) and S_2 (in c) that form the family F. Additionally, the corresponding connected components $C_{S_1} \subsetneq C$ and $C_{S_2} \subsetneq C$ (coloured green) are shown. Taken together, these connected components cover all vertices of $C - \bigcup F$.

1. *$G[C \cup S]$ admits a cp-tree-decomposition (T, B, P) of weight at most k with interface S.*
2. *There exists a family $F = \{S_1, \ldots, S_p\}$ of separators of G such that:*
   - I *each $S_i \in F$ has weight at most k,*
   - II *each S_i is the disjoint union of some subset $S'_i \subseteq S$ and some vertex $v \in C$,*
   - III *for every vertex $w \in C - \bigcup F$ not contained in some separator, there exists a separator $S_i \in F$ and a connected component $C_{S_i} \subsetneq C$ of $G - S_i$ such that $w \in C_{S_i}$ and $G[C_{S_i} \cup S_i]$ admits a cp-tree-decomposition of weight at most k with interface S_i.*

Proof. We first show the implication from (1) to (2). For this, we construct a family of separators with the required properties. To this end, we assume that (T, B, P) is non-redundant, except that the bags with content S may be a strict subset of another bag. Let $t \in V(T)$ be such a node with $B(t) = S$. We construct a cp-tree-decomposition (T', B', P') of weight at most k from (T, B, P) by inserting an additional node between t and each of its neighbours. Let t' be a neighbour of t. Since (T, B, P) is non-redundant, there exists a vertex $w \in B(t') - B(t)$. We now introduce a new node t^* with $B'(t^*) = (B(t) \cap B(t')) + w$ between t and t'. We then define the family F as the collection of bags corresponding to all newly introduced neighbours t^* of t in (T', B', P'). It remains to verify that the family F satisfies the desired properties. Every $S_i \in F$ is a separator, since the connected component C contains at least two vertices and we add only a single vertex to the bag. Moreover, each separator has weight at most k, as it is a subset of some bag $B(t')$, for a neighbour t' of t. Property (II) holds by construction. Finally, observe that for each separator $S_i \in F$, the connected components of $G - S_i$ correspond exactly to the vertex sets appearing in the connected components of $T' - t^*$. Thus, for each vertex in $C - \bigcup F$, we can identify a separator and a corresponding component satisfying the required conditions.

Next, we show the implication from (2) to (1). To this end, we construct a cp-tree-decomposition of $G[C \cup S]$ with weight at most k and interface S by combining the given partial decompositions. We start with a root bag whose content is exactly the separator S. We then iteratively extend the current decompositions by adding a partial decomposition of a connected component induced by a separator of F. Let $S_i \in F$ be a separator of the family

and let $C_{S_i} \subsetneq C$ be a connected component of $G - S_i$ such that $G[C_{S_i} \cup S_i]$ admits a cp-tree-decomposition (T_i, B_i, P_i) of weight at most k with interface S_i. If C_{S_i} is disjoint from the previously added connected components, we attach (T_i, B_i, P_i) to the root by connecting some bag whose content is S_i to the root bag. The vertex coverage and edge coverage conditions of tree-decompositions follow directly from fact that the connected components induced by F cover all vertices of C, together with the defining properties of the separators and the correctness of the partial decompositions. It remains to verify the connectivity condition, namely that for every vertex w, the set of bags containing w induces a connected subtree. First, observe that each vertex $w \in C$ appears in exactly one of the added connected components or in one of the separators S_i, since we only attach pairwise disjoint connected components. Hence, the bags containing such a vertex are connected within the corresponding partial decomposition. For vertices $v \in S$, all occurrences in different partial decompositions are connected via the root bag. Thus, the connectivity condition of cp-tree-decompositions is satisfied. Finally, the weight of the resulting cp-tree-decomposition is at most k, since both the separator S and all partial decompositions have weight at most k. ∎

Together, these lemmas yield the following recursive algorithm for deciding whether the cp-treewidth of a graph is at most a given real number k. Informally, the algorithm employs a dynamic programming approach and stores, for each connected component induced by a separator of weight at most k, whether it admits a cp-tree-decomposition of weight at most k with the required interface. For this, the algorithm enumerates all separators of weight at most k together with the connected components they induce. Using the condition established in the previous lemma, it then determines how the stored value should be set for each such connected component. Processing components in order of increasing size ensures that all values required for recursive checks are already available since the condition only considers strictly smaller connected components. Finally, the algorithm verifies whether there exists a separator that satisfies the condition of the first lemma, that is, whether the partial decompositions can be combined into a cp-tree-decomposition of the whole graph. We now formalize this idea in the following theorem.

Theorem 5.3: *Let $G = (V, E)$ be a graph and let k be a positive real number. Then, there exists an algorithm that decides whether G has cp-treewidth at most k in time $2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}$.*

Proof. We prove this statement by describing a dynamic programming algorithm that checks the conditions described by the two lemmas. This algorithm uses the following partial solutions. For each pair (C, S), where S is a separator of G with weight at most k and C is a connected component of $G - S$, we store a flag indicating whether $G[C \cup S]$ has a cp-tree-decomposition of weight at most k with interface S or not. To obtain all pairs (C, S) for which the flag must be computed, the algorithm enumerates all separators of G with weight at most k. This can be done by iterating over all sets of size at most 2^k and checking for each set whether a given set is a separator and whether its weight is at most k. For each such separator S, we then compute the connected components of $G - S$. In the next step, the pairs (C, S) are processed in increasing order of $|C|$ and for each component the value of the flag is determined using the following two rules.

If $|C \cup S| \leq k$, we construct a cp-tree-decomposition consisting of two bags connected by an edge. One bag contains all vertices of $C \cup S$, the other contains exactly the vertices of S. Both bags use clique-partitions in which every vertex forms its own clique. Hence, in this case the flag can be set to true.

Now consider a pair (C, S) with $|C \cup S| > k$. We first enumerate all separators of weight at most k that can be obtained from S by taking a subset and adding a single vertex. Then, let the family F of separators be given by the set containing all separators enumerated in this step. The algorithm verifies the condition introduced in Lemma 5.2 and thus decides the value of the flag for (C, S) by testing whether F forms a valid family in the sense of the lemma. For this, we consider all connected components $C_{S_i} \subsetneq C$ induced by a separator $S_i \in F$ such that $G[C_{S_i} \cup S_i]$ admits a cp-tree-decomposition of weight at most k with interface S_i. Observe, that these are exactly the connected components $C_{S_i} \subsetneq C$ of $G - S_i$ where the flag for (C_{S_i}, S_i) is set to true. If these connected components together cover all vertices of C that are not contained in any of the separators, then the flag for (C, S) is set to true. Also note that for every such connected component C_{S_i}, the value of the flag for (C_{S_i}, S_i) has already been computed, since C_{S_i} is strictly smaller than C.

Finally, we iterate over all separators of weight at most k and check whether there exists a separator S for which the dynamic program has set the flag to true for all connected components induced by S. If such a separator exists, then condition (2) of Lemma 5.1 is satisfied and G has cp-treewidth at most k by this lemma.

It remains to analyse the running time of the algorithm. We first consider the time needed to enumerate all separators. First, observe that a vertex set of weight k has size at most 2^k by Corollary 4.12 and that there are at most n^{2^k} vertex sets of size at most 2^k. For each such set, we can check in $\mathcal{O}(n^2)$ time whether its forms a separator. To compute the weight of the separator, we brute-force all possible clique-partitions. Here, we use that a set of x vertices has at most x^x clique-partitions, since each clique-partition is a function mapping x vertices to x cliques. Thus, for a set of size at most 2^k, the number of clique-partitions is bounded by $(2^k)^{2^k} = 2^{k \cdot 2^k}$. Computing the weight of a fixed clique-partition takes $\mathcal{O}(2^k)$ time. So the time needed to enumerate all separators of weight at most k by, for each set, testing whether it forms a separator and brute-forcing its weight is

$$\begin{aligned} & n^{2^k} \cdot \left(\mathcal{O}(n^2) + \mathcal{O}(2^{k \cdot 2^k} \cdot 2^k) \right) \\ & \subseteq 2^{2^{\mathcal{O}(k)}} \cdot n^{2^k + 2}. \end{aligned}$$

Next, we continue with the set-up of the connected components. For a fixed separator, its connected components can be computed in time linear in n. To process the pairs (C, S) in increasing order of $|C|$, we use bucket-sorting, which runs in time linear in the number of connected components. Therefore, it remains to bound the total number of connected components considered. Each separator induces at most n connected components. Since there are at most n^{2^k} separators of size at most 2^k, the total number of connected components considered over all separators is at most n^{2^k+1}. Hence, the total time needed for computing and sorting all connected components is

$$\begin{aligned} & \mathcal{O}(n^{2^k+1}) + \mathcal{O}(n^{2^k+1}) \\ & = \mathcal{O}(n^{2^k+1}). \end{aligned}$$

After this, we analyse the time needed for the recursive computation. Fix a pair (C, S). Then, there are at most $2^{|S|} \cdot n \leq 2^{2^k} \cdot n$ candidate separators, since each such separator consist of a subset of S together with a single additional vertex. For each candidate separator, its at most n connected components can be computed in time linear in n. Hence, the $2^{2^k} \cdot n^2$ connected components induced by the candidate separators can be enumerated in time linear

in their total number. Filtering the relevant connected components also requires time linear in their number, since the algorithm merely looks up the previously computed flags for the corresponding pairs. Therefore, it can be decided in time $\mathcal{O}(2^{2^k} \cdot n^2)$ whether these components cover all of C and thus whether the flag for (C, S) should be set to true. Since there are at most n^{2^k+1} pairs of connected components and separators in total, the overall time required for the recursive computation is

$$\begin{aligned} & n^{2^k+1} \cdot \mathcal{O}(2^{2^k} \cdot n^2) \\ & \subseteq 2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}. \end{aligned}$$

Finally, we iterate over all separators and check the stored flag values for their at most n connected components. This final step requires $\mathcal{O}(n^{2^k} \cdot n)$ time. So the total running time is

$$\underbrace{2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+2}}_{\text{enumerating separators}} + \underbrace{\mathcal{O}(n^{2^k+1})}_{\text{enumerating and sorting components}} + \underbrace{2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}}_{\text{recursive computations}} + \underbrace{\mathcal{O}(n^{2^k} \cdot n)}_{\text{search for valid separator}}$$
$$\subseteq 2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}.$$

■

5.2 Approximating CLIQUE-PARTITIONED TREEWIDTH in FPT-time

In this section, we present an FPT-algorithm that computes a $4k + 3$-approximation of cp-treewidth. More precisely, given an input graph G, the algorithm either constructs a cp-tree-decomposition of G with weight at most $4k + 3$, or correctly reports that no cp-tree-decomposition of G with weight at most k exists.

Overview. Our algorithm for approximating CP-TREEWIDTH employs the well-known technique, first introduced by Robertson and Seymour [RS95], of iteratively computing balanced separators of small weight. More concretely, the algorithm repeatedly identifies separators of small weight in the graph. Throughout the recursive steps, we maintain an *interface* to keep track of the parts of previous separators that remain relevant in the current step. The various states of this interface, together with the chosen separators, constitute the bags of the resulting cp-tree-decomposition. Consequently, the algorithm requires separators of small weight that balance the current interface with respect to its weight. Using iterative compression, we may assume that a cp-tree-decomposition of linear weight is available for the subsequent step. To compute suitable separators, we enumerate all partitions of the interface into two subsets and, for each partition, apply a dynamic programming algorithm on the given cp-tree-decomposition to find separators of small weight that separate the interface according to the chosen partition.

For improved readability and to reflect the logical dependencies between the components, we present the algorithm in a bottom-up fashion. We begin by showing that every graph of small cp-treewidth admits a separator of the required form. Next, we describe an algorithm for computing such separators efficiently. Afterwards, we combine these ingredients to construct a cp-tree-decomposition. Finally, we address the precondition of having a cp-tree-decomposition of linear width by using iterative compression.

All graphs of small cp-treewidth have balanced separators of small weight. In the first step, we show that any graph of cp-treewidth at most k admits a balanced separator of weight at most k. To prove the existence of such a separator, we adapt the proof of the well-known result that trees have $\frac{2}{3}$-balanced separators of size 1. We proceed in two steps. First, we show that a such graphs admit a separator of weight at most k that only induces connected components that contain at most half of the interface, measured with respect to weight. In this proof, we closely follow the proof of Lemma 4.6 by Dallard et al. [Dal+26]. Since our algorithm adds at most two child-nodes to the tree-decomposition in each step, we require a separator such that the components induced by this separator can be grouped into two sets that are balanced with respect to the weight of the interface. We extent the initial result to this setting by grouping component together and carefully choosing the parameters of the initial lemma.

Lemma 5.4: *Let $G = (V, E)$ be a graph, let w and k be positive real numbers and let $I \subseteq V$ be an interface of weight at most w. If the cp-treewidth of G is at most k, then there exists a separator $S \subseteq V$ of G with weight at most k such that, for each connected component G_i of $G - S$, the set $I_i = I \cap G_i$ has weight at most $\frac{w}{2}$.*

Proof. Let (T, B, P) be a non-redundant cp-tree-decomposition of G with weight at most k. We show that the desired separator exists by introducing an orientation on the edges of T, based on how each edge separates the interface I.

To this end, we first introduce some notation. Let xy be an edge of T and consider the intersection $S_{xy} = B(x) \cap B(y)$ of its two incident bags. Using this, we define X_{xy} as the component of $G - S_{xy}$ that contains all vertices whose bags lie in the connected component of $T - xy$ that includes x. Similarly, we define Y_{xy} as the component of $G - S_{xy}$ that contains all remaining vertices including those whose bags lie in the connected component of $T - xy$ that includes y. By Lemma 3.11, S_{xy} is a separator of G. In particular, it separates X_{xy} from Y_{xy}.

Using these definitions, we introduce an orientation on the edges of T. We first show the claim that for each edge xy, at least one of $I_x = I \cap X_{xy}$ and $I_y = I \cap Y_{xy}$ has weight at most $\frac{w}{2}$. Here, we use that no clique of any clique-partition $\mathcal{P}$ of I intersects both I_x and I_y. Hence, the set of cliques of $\mathcal{P}$ containing vertices of I_x and the set of cliques of $\mathcal{P}$ containing vertices of I_y are disjoint. Thus, the weight of I is at least the sum of the weights of I_x and I_y and the claim holds. We then orient the edge xy from x to y if I_y has weight greater than $\frac{w}{2}$. If both sides have weight at most than $\frac{w}{2}$, the orientation is chosen arbitrarily.

As T is a tree, there exists a sink $t \in V(T)$. Consequently, all connected components of $G - B(t)$ have weight at most $\frac{w}{2}$ when restricting to I. Note, that removing $B(t)$, instead of $B(t) \cap B(t')$ for some edge tt', only removes more vertices from each component and thus only decreases the weight of the components. Finally, observe that $B(t)$ is a node of T and thus has weight at most k. Therefore, we may choose the bag $B(t)$ as the desired separator. ∎

We now restrict ourselves to the setting with two components. While at least three components are present, we may group together the two components with the smallest weight with respect to the interface. In this way, we obtain a $\frac{2}{3}$-balanced separator. Furthermore, we use an interface of weight $w = 3k + 3$.

Corollary 5.5: *Let $G = (V, E)$ be a graph, let k be a positive real number and let $I \subseteq V$ be an interface of weight at most $3k + 3$. If the cp-treewidth of G is at most k, then there exists a separator S of G that separates I into I_1 and I_2 such that S has weight at most k and each I_i has weight at most $2k + 2$.*

Finding separators with small clique-partitions. In the previous paragraph, we have shown that every graph of cp-treewidth at most k admits a balanced separator of weight at most k. It therefore remains to explain how such a separator can be computed, or how it can be determined that no such separator exists. In the second case, we may conclude by contraposition of Corollary 5.5 that the graph has cp-treewidth greater than k. We present this algorithm in two steps. First, we consider a simplified problem variant in which the separator is only required to have small weight and to separate two given vertex sets, without any balance condition. We then extend this approach to the balanced case by enumerating all partitions of the given interface into two sets and, for each partition, invoking the algorithm for the simplified setting.

In the first step, we employ a dynamic programming algorithm on a given cp-tree-decomposition of weight $\mathcal{O}(k)$. The dynamic program considers, for each vertex in the current bag, whether this vertex is included in the separator. In addition, the algorithm keeps track of several auxiliary properties including the clique-partition of the separator, forgotten connections, and paths from vertices to the sets V_1 and V_2.

Lemma 5.6: *Let $G = (V, E)$ be a graph, let k be a positive real number and let $V_1, V_2 \subseteq V$ be two vertex sets. Let (T, B, P) be a given cp-tree-decomposition of G with weight $\mathcal{O}(k)$. Then, there exists an algorithm that computes a separator S of G together with a clique-partition of S with weight at most k, such that and V_1 and V_2 lie in different connected components of $G - S$, or correctly reports that no such separator exists. This algorithm runs in time $2^{2^{\mathcal{O}(k)}} \cdot n$.*

Proof. As a proof, we present a dynamic programming algorithm with the required running time. We begin by defining the partial solutions for each node, followed by the recursive formulas required to compute them. Finally, we analyse the algorithm's running time by bounding the number of partial solutions, thereby establishing the claimed FPT bound.

The partial solutions for a node $t \in V(T)$ are structured as follows. For the bag $B(t)$, each subset $X \subseteq B(t)$ forms its own partial solution. Here, X represents the portion contained in the current bag of the separator S that separates the graph G_t processed so far. For each subset X, we additionally store the following auxiliary properties. We record the restriction of a clique-partition $\mathcal{P}$ of S to X. We also maintain a mapping from the vertices of $B(t) - X$ to their corresponding connected components in $G_t - S$. Furthermore, we maintain a mapping from those connected components to V_1 or V_2 to track how the components intersect V_1 and V_2. Finally, for each clique of the clique-partition of X, we include a flag indicating whether any vertex of this clique has been forgotten.

We first transform the given cp-tree-decomposition into a nice cp-tree-decomposition of equal weight. Then, we use the following recursive formulas to compute the partial solutions for each node of the nice cp-tree-decomposition.

For a **leaf-node** t, we set $X = \emptyset$. Consequently, the clique-partition of X, the two assignments and the flags are trivial.

For a **forget-node** t with child t' and forgotten vertex v, we construct one partial solution for t from each partial solution for t'. Consider a partial solution of t'. We remove v from X and from the restricted clique-partition. The connected components and the assignment of connected components to the sets V_1 and V_2 no longer store v but remain unchanged otherwise. Finally, we mark the clique that contained v as having a forgotten vertex.

For an **introduce-node** t with child t' and introduced vertex v, we construct multiple partial solutions for t from each partial solution for t'. Consider a partial solution of t'. The subset X for this partial solution can be extended to t by either including v or leaving it unchanged. If v is added to X, one partial solution is created for each clique in the current clique-partition

that has no forgotten vertices, by adding v to that clique. Additionally, one partial solution is created where v forms its own clique. The remaining properties of the new partial solutions are set as follows. If introduced vertex v is not added to X, it is assigned to the connected components of its neighbours. If v is assigned to multiple connected components, these connected components are merged into one connected component. Next, the mapping of components to the sets V_1 and V_2 is updated. Here, only the connected component containing v is affected: if $v \in V_i$ for $i \in [2]$, we map this component to V_i; otherwise, the mapping remains unchanged. If this new assignment leads to a contradiction, that is we assign a component that currently is mapped to V_1 to V_2 or vice versa, we discard this partial solution. This is correct as such an double assignment implies the existence of a path from V_1 to V_2 in $G_t - S$. Due to this such an invalid partial solution represents no valid separator. Finally, all flags remain unchanged. If v is added to a new clique, this clique has not forgotten vertices yet.

For a **join-node** t with children t_1 and t_2, we construct at most one partial solution for t from each pair of partial solutions for t_1 and t_2. Consider a partial solution p_1 of t_1 and a partial solution p_2 of t_2. We require that the subsets of the bags and their restricted clique-partitions for p_1 and p_2 coincide. Then, the partial solution for t inherits the subset and its restricted clique-partition from its children. Next, connected components of the two branches that share vertices are merged. Similarly, the assignment of components to the sets V_1 and V_2 is merged. That is, if a component is assigned to V_i, $i \in [2]$, for one child, it is also assigned to V_i in the current node t. As before, any assignment that would assign a component to both V_1 and V_2 results in an invalid partial solution. Finally, if a clique has forgotten vertices in one branch, it is marked as having forgotten vertices in the join-node as well.

We apply these rules in a bottom-up manner, from the leaves to the root, to compute the set of partial solutions for each node. If there exists a partial solution at the root bag whose unrestricted clique-partition has weight at most k, then the desired separator exists. To ensure that the full separator S separates the sets V_1 and V_2, we have discarded all partial solutions in which a connection between V_1 and V_2 was introduced. The separator S can be reconstructed by tracing the chosen partial solution back through the cp-tree-decomposition. Similarly, the clique-partition of S and its weight is obtained from the restricted clique-partitions stored at the bags. The two (not necessarily connected) components of $G - S$ are determined as follows: all connected components that are mapped to V_1 form one side and all other connected components, meaning those mapped to V_2 or neither part, form the other side.

It remains to analyse the running time of the algorithm. We begin by bounding the number of possible partial solutions. Remember, that a partial solution consists of a subset of the current bag, a clique-partition of this subset, a mapping of vertices to connected components, a mapping of connected components to the sets V_1 and V_2 and a flag for each clique. We consider each part in turn. First, note that the number of vertices of each bag is at most $2^{\mathcal{O}(k)}$, since the weight of each bag of the given cp-tree-decomposition is in $\mathcal{O}(k)$ (see Corollary 4.12). Therefore, there are at most $2^{2^{\mathcal{O}(k)}}$ subset of the current bag. Next, we observe that a clique-partition can be viewed as a function mapping each vertex to a clique. Hence, a set of x vertices has at most x^x clique-partitions, which bounds the number of clique-partitions for one of the considered subsets to $(2^{\mathcal{O}(k)})^{2^{\mathcal{O}(k)}} = 2^{2^{\mathcal{O}(k)}}$. Since the current bag contains at most $2^{\mathcal{O}(k)}$ vertices, there are at most $2^{\mathcal{O}(k)}$ connected components containing vertices of the current bag. Thus, the number of assignments of vertices to connected components is limited by $(2^{\mathcal{O}(k)})^{2^{\mathcal{O}(k)}} = 2^{2^{\mathcal{O}(k)}}$. Similarly, with at most $2^{\mathcal{O}(k)}$ connected components, the number of mappings from components to V_1 or V_2 is at most $2^{2^{\mathcal{O}(k)}}$. Finally, a clique-partition of $2^{\mathcal{O}(k)}$

vertices contains at most $2^{\mathcal{O}(k)}$ cliques, so the number of possible flag assignments is also bounded by $2^{2^{\mathcal{O}(k)}}$. Multiplying all bounds together, the total number of partial solutions per bag is

$$\begin{aligned}&2^{2^{\mathcal{O}(k)}} \cdot 2^{2^{\mathcal{O}(k)}} \cdot 2^{2^{\mathcal{O}(k)}} \cdot 2^{2^{\mathcal{O}(k)}} \cdot 2^{2^{\mathcal{O}(k)}}\\ &= 2^{2^{\mathcal{O}(k)}}\end{aligned}$$

Furthermore, all computations for a partial solution can be carried out in $2^{\mathcal{O}(k)}$ by iterating over all vertices of the current bag. Hence, the dynamic program requires at most $2^{2^{\mathcal{O}(k)}} \cdot 2^{\mathcal{O}(k)} = 2^{2^{\mathcal{O}(k)}}$ time per bag. Remember, that a nice cp-tree-decomposition of at most $2^{\mathcal{O}(k)} \cdot n$ nodes and the same weight can be computed in time $2^{\mathcal{O}(k)} \cdot n$ from the given cp-tree-decomposition (see Lemma 3.9 and Lemma 4.10). Thus, computing all partial solutions requires time $2^{2^{\mathcal{O}(k)}} \cdot 2^{\mathcal{O}(k)} \cdot n = 2^{2^{\mathcal{O}(k)}} \cdot n$. At the root, there are at most $2^{2^{\mathcal{O}(k)}}$ partial solutions. For each such partial solution, the final computation of the total weight can be performed in $2^{\mathcal{O}(k)} \cdot n$ time by tracing this partial solution through the $2^{\mathcal{O}(k)} \cdot n$ nodes and computing the weight in each bag of $2^{\mathcal{O}(k)}$ vertices. Therefore, the total running time of the algorithm is $2^{2^{\mathcal{O}(k)}} \cdot n$. ∎

We now extend this algorithm for the simplified case to the general setting. To this end, we must ensure that the computed separator splits the given interface in a balanced way. This can be achieved by requiring that the two sets V_1 and V_2, which the separator has to separate, form a balanced partition of the interface. Such balanced partitions can be obtained by brute-force enumeration of all partitions of the interface.

Lemma 5.7: *Let $G = (V, E)$ be a graph, let k be a positive real number and let $I \subseteq V$ be an interface of weight at most $3k + 3$. Let (T, B, P) be a given cp-tree-decomposition of G with weight $\mathcal{O}(k)$. Then, there exists an algorithm that computes a separator S of G together with a clique-partition of S with weight at most k that separates I into I_1 and I_2 such that each I_i, $i \in [2]$, has weight at most $2k + 2$ or correctly reports that no such separator exists. This algorithm has running time $2^{2^{\mathcal{O}(k)}} \cdot n$.*

Proof. We first observe, that if the sets I_1 and I_2, into which the interface is separated, are known, we can directly apply the algorithm of Lemma 5.6 for unbalanced separators to solve the problem. Thus, it remains to identify these sets and verify that they satisfy the required weight constraints. We use a brute-force approach to find these two sets. Specifically, we iterate over all subsets I_1 of I, compute the weights of I_1 and $I_2 = I - I_1$ and then invoke the algorithm for unbalanced separators, if the weight constraints are satisfied. As both the size and the weight of I are bounded by a function of k, the computation of the weight can be carried out by comparing all possible clique-partitions of the set.

We now analyse the running time. Since I has weight at most $\mathcal{O}(k)$, it contains at most $2^{\mathcal{O}(k)}$ vertices (see Corollary 4.12). Thus, there are at most $2^{2^{\mathcal{O}(k)}}$ subset to check. To compute the weight of I_1 and I_2 by brute-force, we note that a set of x vertices has at most x^x clique-partitions, which bounds the number of clique-partitions for each of the two subsets to $(2^{\mathcal{O}(k)})^{2^{\mathcal{O}(k)}} = 2^{2^{\mathcal{O}(k)}}$. Computing the weight of a clique-partition of at most $2^{\mathcal{O}(k)}$ vertices can be done in $2^{\mathcal{O}(k)}$ time, so brute-forcing the weight requires at most $2^{\mathcal{O}(k)} \cdot 2^{2^{\mathcal{O}(k)}} = 2^{2^{\mathcal{O}(k)}}$ time. Then, the $2^{2^{\mathcal{O}(k)}} \cdot n$-time algorithm for unbalanced separators is called for each subset I_1 of the interface I that satisfies the weight constraint. Consequently, the total running time to find a balanced separator is $2^{2^{\mathcal{O}(k)}} \cdot \left(2^{2^{\mathcal{O}(k)}} + 2^{2^{\mathcal{O}(k)}} \cdot n\right) = 2^{2^{\mathcal{O}(k)}} \cdot n$. ∎

Computing cp-tree-decompositions using balanced separators. The next step is to use these balanced separators to compute cp-tree-decompositions of low weight. To this end, we iteratively separate the current component of the graph into smaller components that balance the interface with respect to its weight. In each step, the new interface is formed from the relevant part of the old interface and the current separator. These states of the interface combined with the different separators then form the bags of our cp-tree-decomposition.

Lemma 5.8: *Let $G = (V, E)$ be a graph, let k be a positive real number and let (T, B, P) be a given cp-tree-decomposition of G with weight $\mathcal{O}(k)$. Then, there exists an algorithm that either finds a cp-tree-decomposition of G with weight at most $4k + 3$ or correctly reports that G has cp-treewidth more than k. This algorithm has running time $2^{2^{\mathcal{O}(k)}} \cdot n^2$.*

Proof. As a proof, we present an algorithm for constructing such cp-tree-decompositions. If G has less that $4k + 3$ vertices, we use the trivial decomposition and the trivial clique-partition. Otherwise, we recursively construct cp-tree-decompositions that contain a given interface I in the root bag. We maintain the invariants that I has weight at most $3k + 2$ and that a clique-partition of I with this weight is known. We begin by adding one vertex to I and assigning it to its own clique. After this, I has weight at most $3k + 3$. Next, we apply the algorithm of Lemma 5.7 to compute a balanced separators S of weight at most k that separates I into I_1 and I_2. If computing the separator fails, we return that the cp-treewidth of G is greater than k. This is correct, by contra-position of Corollary 5.5 which guarantees the existence of such a separator for graphs of cp-treewidth at most k. Otherwise, we create a root bag containing only S and I. The weight of this bag is at most $3k + 3 + k = 4k + 3$ and the clique-partition of this bag is given by the disjoint union of the partitions of S and I. We then recursively construct the two child bags and their descendants. In the recursive calls, we use the interfaces $I_1 \cup S$ and $I_2 \cup S$, respectively. These have weight at most $2k + 2 + k = 3k + 2$. Here, the known clique-partitions of I_1 and I_2 as well as the clique-partition of S found during its computation are used. Thus, our invariants are maintained.

To establish correctness, it remains to verify that the result indeed is a valid cp-tree-decomposition. Clearly, every vertex of the graph is eventually added to some bag, and thus the vertex coverage property is satisfied. Moreover, any two adjacent vertices remain in the same component for all separators and therefore appear together in at least one leaf. This verifies the edge coverage property. Finally, observe that vertices are never removed from bags and only appear in two different branches if they belong to the interface at the node where the branches split. Hence, the connectivity condition is satisfied.

This algorithm makes at most $\mathcal{O}(n)$ recursive calls, since in each step at least one vertex is added to the interface. Furthermore, each vertex is added at most once. Therefore, the algorithm for balanced separators, which has running time $2^{2^{\mathcal{O}(k)}} \cdot n$, is called at most $\mathcal{O}(n)$ times. ∎

Using iterative compression to obtain a cp-tree-decomposition of linear weight. In the final step, we address the precondition of having a cp-tree-decomposition of weight $\mathcal{O}(k)$ available. Such a decomposition can be obtained using iterative compression.

We thus show the following theorem:

Theorem 5.9: *Let $G = (V, E)$ be a graph and let k be a positive real number. Then, there exists an algorithm that either finds a cp-tree-decomposition of G with weight at most $4k + 3$ or correctly reports that G has cp-treewidth more than k. This algorithm has running time $2^{2^{\mathcal{O}(k)}} \cdot n^3$.*

Proof. To apply iterative compression we order the vertices arbitrarily as $v_1, \dots v_n$. We then iteratively compute a cp-tree-decomposition for each $G_i = G[v_1, \dots, v_i]$. In the base case, we use the trivial cp-tree-decomposition. Given a cp-tree-decomposition of G_{i-1}, we obtain a cp-tree-decomposition of G_i by adding v_i as an isolated vertex to each bag. This increases the weight of the decomposition by one. Since the original cp-tree-decomposition has weight at most $4k + 3$, the new cp-tree-decomposition has weight at most $4k + 4$. Thus, we obtain a cp-tree-decomposition of weight linear in k. We then apply the algorithm of Lemma 5.8 to obtain a cp-tree-decomposition of weight $4k + 3$ for G_i. If this algorithm fails to find such a cp-tree-decomposition, no cp-tree-decomposition of weight at most k can exist for G. This is due to the fact that cp-treewidth is monotone under the taking of induced subgraphs.

Finally, note that we call the algorithm of Lemma 5.8 in total n times and each call requires $2^{2^{\mathcal{O}(k)}} \cdot n^2$ time. ∎

5.3 Adapting these approaches to TREE-CLIQUE-COVER NUMBER fails

While these two algorithms can be adapted naturally to cp-treewidth, this approach does not extend as easily to other related parameters. As an example, we consider the tree-clique-cover number, which is smaller than cp-treewidth as shown in the previous section. However, this advantage in size comes at the cost of increased computational difficulty. In particular, we prove that no XP-algorithm for computing the tree-clique-cover number exists, unless $\mathcal{P} = \mathcal{NP}$. To this end, we adapt the approach used for the tree-independence number by Dallard [Dal+26] and use the following well-known result. Here, the k-CLIQUE COVER problem asks whether a given graph admits a clique-cover consisting of at most k cliques.

Lemma 5.10 ([Kar72 | GJ79])**:** *For every constant $k \geq 3$, the problem k-CLIQUE COVER is $\mathcal{NP}$-hard.*

As a direct consequence, we obtain the following result for CLIQUE COVER. In this problem the integer k is part of the input.

Lemma 5.11: *There exists no algorithm that solves CLIQUE COVER parameterized by the solution size in time $n^{f(k)}$ for any computable function f, unless $\mathcal{P} = \mathcal{NP}$.*

Proof. Assume for sake of contradiction that such an algorithm exists. Then, for every fixed k, the $\mathcal{NP}$-hard problem k-CLIQUE COVER can be solved in polynomial time, since k is a constant. This contradicts the assumption that $\mathcal{P} \neq \mathcal{NP}$. ∎

Using the following reduction, this result extends to the tree-clique-cover number. We construct an auxiliary graph H from two disjoint copies G_1 and G_2 of the input graph G by connecting each pair of vertices $x \in V(G_1), y \in V(G_2)$ with an edge. By Lemma 3.14, the clique-cover number of any bag in a tree-decomposition of H equals the clique-cover number of G. This reduction yields the following results.

Lemma 5.12: *The problem TREE-CLIQUE-COVER NUMBER is $\mathcal{NP}$-hard.*

Theorem 5.13: *There exists no algorithm that solves TREE-CLIQUE-COVER NUMBER parameterized by the tree-clique-cover number in time $n^{f(k)}$ for any computable function f, unless $\mathcal{P} = \mathcal{NP}$.*

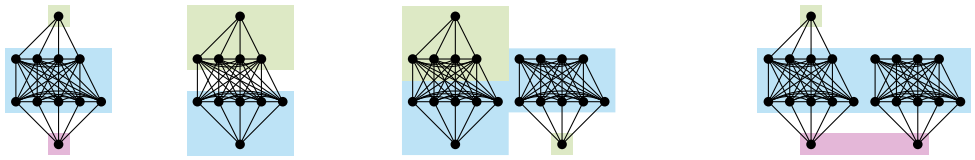


Figure 5.2: An example showing that restricting an optimal clique-partition of a subgraph of the graph H constructed above to the graph G does not necessarily yield an optimal clique-partition of G. The edges between G_1 and G_2 are not shown to preserve readability. We use the multiplicative formulation of weight. Part (a) depicts the graph G with the optimal clique-partition of G (colouring) with weight $10 \cdot 2 \cdot 2 = 40$. Part (b) depicts the graph G with an alternative clique-partition of G (colouring) with weight $7 \cdot 6 = 42$. Part (c) depicts a subgraph of the auxiliary graph H with the optimal clique-partition of this subgraph (colouring) with weight $16 \cdot 7 = 112$. Part (d) depicts a subgraph of the auxiliary graph H with an alternative clique-partition of this subgraph (colouring) with weight $19 \cdot 3 \cdot 2 = 114$. Note that the optimal clique-partition of (a) is a restriction of the suboptimal clique-partition of (d), while the suboptimal clique-partition of (b) is a restriction of the optimal clique-partition of (c).

We conclude with some notes. First, note that the above reduction allows hardness results for approximating k-Clique Cover to be transferred to the approximation of Tree-clique-cover number. An introduction to the hardness of approximating the chromatic number of a graph, which is strongly related to the clique-cover number, can be found in the work of Zuckerman [Zuc07]. Next, note that similar results have been obtained for Tree-independence number (see Theorem 1.2 of [Dal+26]), which concerns the computation of another parameter that is smaller than cp-treewidth. Finally observe, that the approach used above for proving hardness fails for cp-treewidth, since restricting an optimal clique-partition of a subgraph of H does not necessarily yield an optimal clique-partition of G. An example is shown in Figure 5.2. We therefore explore a different approach for showing the $\mathcal{NP}$-hardness of cp-Treewidth in Chapter 7.

6 On path-decompositions of grids

In this chapter, we prepare the ground for the next chapter, where we show that computing CP-TREEWIDTH is $\mathcal{NP}$-hard. One of the reductions used in that proof constructs an auxiliary graph by adding a grid to the original graph. To establish the correctness of this reduction, we will show that every path-decomposition of the auxiliary graph contains a bag that consists exclusively of vertices of the grid. Since our argument relies on a comparison of bag sizes, we require a lower bound on the sizes of bags in path-decompositions of grids that matches the upper bound on the bag sizes of path-decompositions of the auxiliary graph obtained during the reduction. Accordingly, we prove the following result in this chapter.

Theorem 6.1: *Let $\alpha < \beta$ be two integers. Let Γ be a grid with α rows and β columns and let (P, B) be a path-decomposition of Γ. Then, there exists a subpath $L = (p_\ell(\Gamma), \ldots, p_r(\Gamma))$ of P such that:*

1. *every bag in the subpath L has size at least α,*
2. *for every node p' of the subpath L with size exactly α the following holds: the closest node p'' of L to the right of p' with $B(p'') \neq B(p')$ has at least $\alpha + 1$ vertices in its bag,*
3. *there are at most α^2 vertices that appear only in bags strictly left of $p_\ell(\Gamma)$, and*
4. *there are at most α^2 vertices that appear only in bags strictly right of $p_r(\Gamma)$.*

The approach used to prove this property is to start from a single bag that contains one vertex from each row or one vertex from each column and therefore already satisfies the required size bound. Then, this subpath is iteratively extended to both sides by adding a neighbouring node to the subpath, while preserving the size constraint in every extension step. Eventually, this process reaches a bag where the size bound is violated, and where it can be shown that the subpath constructed up to this point satisfies the desired properties.

To formalize this process, we introduce three rules. First, the termination rule is applied once the remaining part of the path is guaranteed to contain only few vertices. To this end, we identify a condition on bags which, once satisfied, guarantees that a section of the decomposition contains only few vertices. For the extension step, we distinguish between a weak and a strong extension rule. The weak extension rule only preserves the property that each bag of the subpath is sufficiently large. The strong extension rule additionally maintains the property that the current bag contains a vertex from each row or from each column. Moreover, we show that certain connected components induced by a neighbouring bag are always subsets of the connected components induced by the current bag. Consequently, substructures forbidden by a bag cannot reappear for its neighbour. This observation is used to determine the order in which the extension rules are applied applied. To establish the correctness of these rules, we first formalize the key properties mentioned above in a series of auxiliary lemmas. Afterwards, we prove Theorem 6.1 using these auxiliary lemmas.

Notation. Throughout the remainder of this chapter, we use two different notions of connectivity and connected components. For a grid Γ, let Γ^d denote the graph obtained from Γ by adding all diagonal edges $\{(a,b),(a+1,b+1)\}$ and $\{(a,b),(a+1,b-1)\}$. We distinguish the following notions of connectivity. A vertex set is called *connected* if it is connected using only the edges of Γ. It is called ⊠-*connected* if it is connected using the edges of Γ^d. In our arguments, we will typically identify a separator S and the connected components of $\Gamma - S$ using standard connectivity. We then analyse the ⊠-connected components of S.

A criterion for termination. Recall that our approach for proving a lower bound on bag sizes starts with an initial subpath on which the bound already holds. We then apply two extension rules to enlarge this subpath towards both ends. Finally, a termination rule is triggered once the remaining bags contain only few vertices. We begin our analysis of grids by formalizing a condition under which this termination rule can be applied safely. Here, we proceed as follows. First, we classify the connected components induced by a separator into five distinct types. We then use this classification to formulate a termination condition by showing that, for a separator of small size, certain types of components necessarily contain only few vertices. Consequently, a small separator that induces only such component types may be used as the endpoint of the constructed subpath. To prove this result, we study the ⊠-connected components of a bag and then lift the obtained results to the full bag. Therefore, we must show that analysing the ⊠-connected components does not lose any information compared to analysing the bag as a whole.

We begin by classifying the connected components induced by a separator into types. To this end, we first describe the five possible ways in which a ⊠-connected separator S can induce connected components in the grid and then extend this result to ⊠-disconnected separators. We classify the connected components of $\Gamma - S$ according to the number of boundary sides of Γ to which they are incident. Specifically, we distinguish between *type-0-*, *type-1-*, *type-2-*, *type-3-* and *type-4-components*. Here, a *type-i component* is incident to exactly i consecutive boundary sides. These cases are illustrated in Figure 6.1. Using the following remarks we extend these definitions to the case where S is ⊠-disconnected. First, any ⊠-connected component of S that is not a separator of Γ can be treated as if it were absent. Type-0-, type-1-, type-2-, and type-4-components are then defined exactly as before. For type-3-components we allow an additional configuration. Besides being incident to three boundary sides, a type-3-component may be incident to two non-adjacent boundary sides and to two distinct ⊠-connected components of S (see for example Figure 6.1e). In this case, the boundary sides and the ⊠-connected components of S alternate. Informally, this can be thought of as the additional ⊠-connected component replacing one boundary side.

In the following lemmas we are primarily interested in whether connected components of $\Gamma - S$ have type at most two or at least three. We therefore call a connected component of $\Gamma - S$ a *type-≤2-component*, if it is a type-0-, type-1- or type-2-component.

We now show that it is sufficient to restrict our analysis to the ⊠-connected components of a bag. More precisely, we prove that for every type-≤2-component induced by the full bag, there exists a ⊠-connected component of this bag that induces a connected component which is a superset of the original one and has the same type.

Lemma 6.2: *Let Γ be a grid and let (P,B) be a path-decomposition of Γ. Let $p \in V(P)$ be a node of (P,B). For every type-≤2-component C_p of $\Gamma - B(p)$ there exists a ⊠-connected component S of $B(p)$ such that S induces a type-≤2-component C_S with $C_p \subseteq C_S$.*

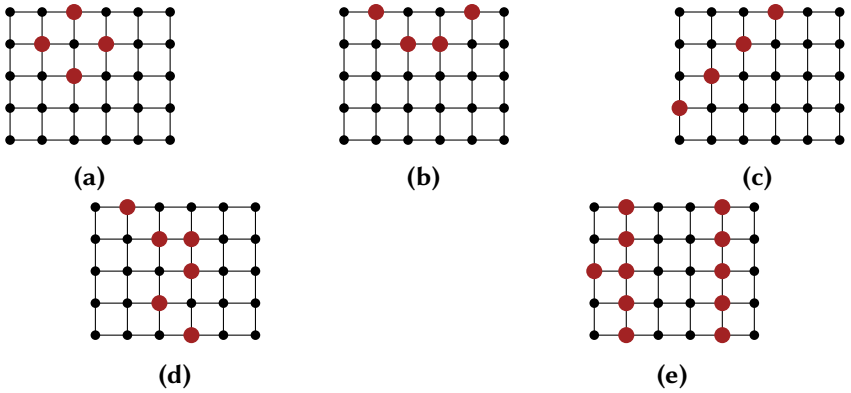
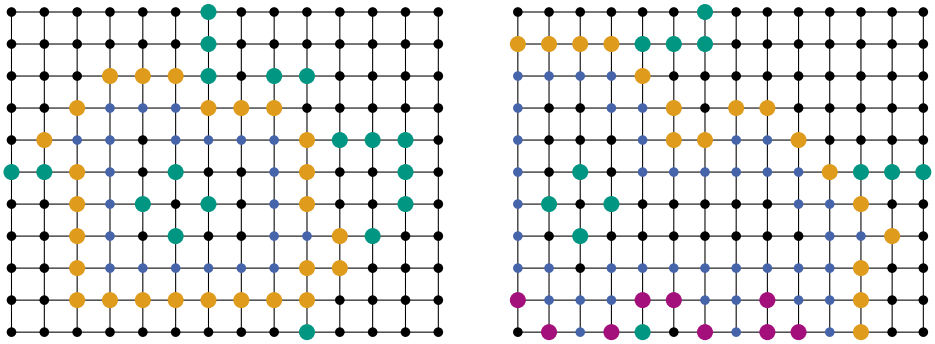


Figure 6.1: The different ways in which a separator (red) can split a grid into two connected components. Part (a) shows a type-0-component and a type-4-component, (b) visualizes a type-1-component and a type-4-component, (c) illustrates a type-2-component and a type-4-component, (d) depicts two type-3-components. Part (e) shows a separator with multiple ⊠-connected components. Here, the middle and right connected components of the grid are type-3-components and the two left connected components of the grid are type-2-components.

Proof. For this proof, we fix the standard embedding of Γ. We identify a ⊠-connected component of $B(p)$ with these properties by considering the vertices along the outer face of C_p and showing that a suitable portion of their neighbourhood is ⊠-connected and induces the desired connected component C_S. We first treat the case of type-0-components, which is simpler because there are no intersections with boundary sides. Afterwards, we extend the argument to type-1- and type-2-components, where the neighbourhood of the outer face is split into multiple parts by the boundary sides. Here, we choose one of these parts. We visualize these arguments in Figure 6.2.

Let C_p be a type-0-component of $\Gamma - B(p)$. Consider the set O of vertices of C_p lying on its outer face. Let I be the intersection of $B(p)$ with the neighbourhood of O along this outer face. Note that this excludes neighbours of O that are located on a different face than the outer face of C_p. We choose S to be the ⊠-connected component of $B(p)$ that contains I. It remains to show that this choice is valid. First, we prove that I is ⊠-connected and hence fully contained in a single ⊠-connected component of $B(p)$. Since C_p does not intersect any boundary side, the set I consist precisely of the vertices in the adjacent row or column to the vertices of O. When traversing the outer face of C_p, consecutive vertices differ by exactly one in either their row or column, and alternating horizontal and vertical steps yield diagonal adjacencies. Consequently, I is ⊠-connected. Next, we prove that S induces a connected component C_S of Γ such that $C_p \subseteq C_S$ and the type of C_S is at most 2. Observe, that S induces a connected component C_S that is fully enclosed by S. By construction C_p is entirely enclosed by S and therefore $C_p \subseteq C_S$. Moreover, C_S does not intersect any boundary side, since it is completely surrounded by S. Hence, C_S has type 0. This establishes the lemma for type-0-components.

We now extend this argument to type-1- and type-2- components, taking into account that boundary sides may split the neighbourhood considered above into several parts. Let C_p be a type-1- or type-2-component of $\Gamma - B(p)$. Again, let O be the set of vertices of C_p on its outer face, and let I be the intersection of $B(p)$ with the neighbourhood of O along this outer face. In contrast to the previous case, I may now consist of several ⊠-connected components. We



(a) (b)

Figure 6.2: This figure illustrates the different sets considered in the proof of Lemma 6.2. Part (a) shows a type-0-component, while part (b) depicts a type-2-component. The inducing bag is marked in orange, purple and green using larger nodes. The proof first identifies the outer face (blue) of the connected component C_p and its neighbourhood (orange and purple) along this outer face. In the type 0 case, this neighbourhood already suffices. Then, the desired ⊠-connected component is obtained by taking this set and, if necessary, adding all ⊠-connected vertices of the bag (part of the green).
In the type 1 and type 2 cases, the neighbourhood is interrupted by the boundary sides. Hence, only one ⊠-connected component (orange) of this neighbourhood is used. The different configurations used in the proof are depicted as follows. First, the rightmost purple ⊠-connected component intersects the boundary side only in a single interval and therefore does not separate the grid on its own. Next, the remaining purple ⊠-connected components lie next to each other. Consequently, a third ⊠-connected component (orange) must encapsulate them. Since this orange portion of the neighbourhood encapsulates all purple portions, the purple parts only subdivide C_p. Together with the ⊠-connected vertices (part of the green) these orange vertices form the desired ⊠-connected component.

choose S to be a subset of those ⊠-connected components of I that minimizes the number of its vertices lying on the boundary sides, while preserving the following two properties. First, S induces a type$\leq$2-component C_S that is a superset of C_p. Second, S, together with the at most two boundary sides incident to C_p, is ⊠-connected. Note that only entire ⊠-connected components may be removed in this process. Clearly, I satisfies these conditions, since we only consider the neighbourhood of the outer face, which is ⊠-connected up to boundary sides. Hence, S is well-defined.

We now show that this minimal set S also is ⊠-connected on its own. Assume for sake of contradiction that this is not the case, and let X and Y be two distinct ⊠-connected components of S. We show that in each possible configuration, at least one of them can be removed from S without violating the two properties, contradicting the minimality of S. First, suppose that one of the components, say X, intersects the boundary sides in exactly one consecutive interval. Then X can be removed. Indeed, one can traverse X, similarly to before, from one part of the boundary side to another, changing the row and column by at most one in each step. The corresponding part of O is then connected and not contained in $B(p)$. Hence, X is not a separator and removing X is correct. Therefore, every ⊠-connected component of S must contain at least two disjoint intervals of the boundary sides. Next, consider the case of nesting, where one of X and Y *encapsulates* the other. That is, one is fully contained in the connected component, incident to the same boundary sides as C_p, that is induced by the

other. W.l.o.g., assume that X is encapsulated by Y. So, Y induces a type-$\leq$2-component of the grid that contains X. Thus, X only subdivides this connected component into smaller ones and may be removed without violating the properties. Finally, suppose that X and Y do not nest, but lie next to each other. If C_p is a part of a connected component between one of X and Y and the boundary sides incident to C_p, then C_p would be disconnected, since it must also be adjacent to the other. This is a contradiction. If C_p lies outside these connected components for of both X and Y, then there must exist a third $\boxtimes$-connected component Z of S that encapsulates both X and Y. Otherwise, C_p would be incident to more than two boundary sides, contradicting its type. In this situation, both X and Y can be removed. In all cases, we obtain a contradiction to the minimality of S. Hence, S must be $\boxtimes$-connected. Finally, observe that the $\boxtimes$-connected component of $B(p)$ that contains S, also is $\boxtimes$-connected and induces a type-$\leq$2-component that is a superset of C_p. Therefore, the lemma holds. ∎

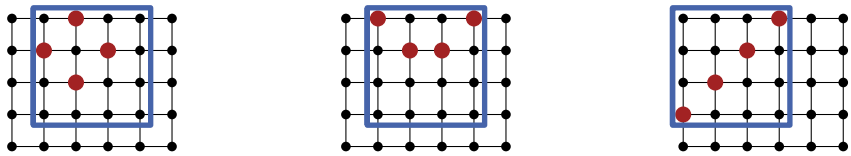
We are now ready to formally introduce the condition required for termination. We first analyse the connected components induced by the $\boxtimes$-connected components of a bag. We then lift the obtained results to the full bag using the previous lemma. In this way, the correctness of this condition can be verified locally on $\boxtimes$-connected components without losing information about the connected components induced by the entire bag.

Lemma 6.3: *Let Γ be a grid and let (P, B) be a path-decomposition of Γ. Let $p \in V(P)$ be a node. Then, all type-$\leq$2-components of Γ induced by $B(p)$, together with $B(p)$ itself, contain at most $|B(p)|^2$ vertices in total.*

Proof. Our goal is to show that all vertices of a type-$\leq$2 component of Γ induced by $B(p)$ lie close to $B(p)$. Consequently, the size of such a component is bounded by the number of vertices that are close to $B(p)$, which in turn depends solely on the size of $B(p)$.

We first consider a single $\boxtimes$-connected component S of $B(p)$ and then later on extend this result to bags with multiple $\boxtimes$-connected components. The first step is to formally define the notion of closeness to S mentioned above. Since S is $\boxtimes$-connected, any vertex of S can be reached from any other vertex of S by a simple $\boxtimes$-path in Γ^d containing only vertices of S. In this path, each edge changes the row index by at most one and the column index by at most one. Hence, the difference between the maximum and minimum row occurring in S is at most $|S|$, and the same holds for the column indices. Let i be the topmost row containing a vertex of S and let j be leftmost column containing a vertex of S. Then, consider an axis-aligned square Q_S that captures $|S|$ rows and $|S|$ columns whose top left corner is the vertex in column j of row i. As no vertex of S is more than $|S|$ rows below row i and no vertex of S is more than $|S|$ columns right of column j, all vertices of S lie within Q_S. Furthermore, the square Q_S contains at most $|S|^2$ vertices. This square is visualized in Figure 6.3.

Let C be a type-$\leq$2-component of the grid induced by S. We now show that all vertices of this connected component also lie inside the square Q_S. By definition C is incident to at most two boundary sides. To separate C from the remaining grid the separator S needs to intersect all boundary sides that C is incident to. So for each vertex of C there is a vertex of S that is in the same column or left, a vertex of S that in the same column or right, a vertex of S that is in the same row or above, and a vertex of S that is in the same row or below. Thus, not only all vertices of S but also all vertices of C lie inside Q_S. Since all type-$\leq$2-components of the grid induced by S as well as the $\boxtimes$-connected component S itself, lie inside Q_S, the total number of vertices contained in them is at most $|S|^2$.



(a) (b) (c)

Figure 6.3: The square Q_S constructed in Lemma 6.3 (blue) for examples of connected components of type 0 (a), type 1 (b) and type 2 (c).

We now extend this result to bags with multiple ⊠-connected components. For every ⊠-connected component S of $B(p)$, the total number of vertices contained in S together with all type-≤2-components of the grid induced by S is at most $|S|^2$, as shown above. We now show the claim that considering those type-≤2-components covers all type-≤2 components induced by $B(p)$. Indeed, if $B(p)$ induces a type-≤2-component, then Lemma 6.2 guarantees the existence of a ⊠-connected component of $B(p)$ that induces a connected component which is a superset of the original one and also has type at most 2. Consequently, the total number of vertices contained in $B(p)$ together with all type≤2-components induced by $B(p)$ is at most the sum over all ⊠-connected components of $B(p)$ of the vertices counted for each such component. Using the bound derived above, we obtain

$$\begin{aligned}
&\sum_{S,\ \boxtimes\text{-connected component of } B(p)} |S|^2 \\
&\leq \Bigg(\sum_{S,\ \boxtimes\text{-connected component of } B(p)} |S| \Bigg)^2 \\
&= |B(p)|^2.
\end{aligned}$$

∎

Extending the subpath preserves the desired properties. In this paragraph, we analyse which structural properties are kept intact when extending the subpath to a neighbouring bag. Our focus is on two aspects: the types of connected components induced by the bag, and the additional property that the bag contains at least one vertex from each row or from each column. For simplicity of presentation, we state all results only for extensions to the left; the corresponding statements for extensions to the right follow symmetrically.

We first study the impact of a simple extension without any additional properties on the types of connected components induced by a bag. To this end, we state a lemma that describes the relationship between the connected components of the grid induced by neighbouring bags in a path-decomposition. Informally, the lemma states that when moving towards one end of the path-decomposition, connected components can only shrink: they may be split into smaller components, but they can never be merged. Consequently, the type of a connected component cannot increase.

Lemma 6.4: *Let G be a graph and let (P, B) be a path-decomposition of G. Let $p, p' \in V(P)$ be nodes of (P, B) with p' being the left neighbour of p. Then, there exists a function that assigns each connected component $C_{p'}$ of $G - B(p')$ whose vertices appear only in bags strictly left of p' to exactly one connected component C_p of $G - B(p)$ whose vertices appear only in bags strictly left of p. If $C_{p'}$ is mapped to C_p, then $C_{p'} \subseteq C_p$.*

Proof. Let C_p be a connected component of G induced by $B(p)$ whose vertices appear only in bags strictly left of p. We first identify vertices of $B(p)$ that also appear in $B(p')$. Using these vertices, we then obtain subsets of C_p that form connected components for p' and map all such subsets to C_p. In the final step, we show that this mapping covers all connected components of $G - B(p')$.

We begin by showing that $B(p')$ still separates C_p from the remaining grid. By the edge coverage property of path-decompositions, for every edge of G there exists a bag containing both of its endpoints. Hence, every neighbour of a vertex in C_p must also occur in at least one bag strictly left of p. In particular, there are edges between C_p and its neighbourhood in $B(p)$, denoted by $\mathcal{N}$. Thus, these vertices of $\mathcal{N}$ must still appear in a bag strictly left of p. By the connectedness property of path-decompositions, the vertices of $\mathcal{N}$ are therefore also contained in $B(p')$. Consequently, $B(p')$ contains all neighbours of C_p and thus this bag also separates the vertices of C_p from all other vertices of the grid.

We now examine how the connected components evolve when moving from p to p' and verify that all components are covered by the mapping we define. Observe that $B(p')$ may additionally contain vertices of C_p. Therefore, the component C_p may be split into several connected components of $G - B(p')$. But, each of these components is a subset of C_p. We map all such components to C_p. It thus remains to show that this mapping covers all connected components induced by $B(p')$. To this end, observe that the set of vertices appearing strictly left of p' is a subset of those appearing strictly left of p. Since the connected components of $G - B(p)$ cover all vertices appearing in bags strictly left of p, the mapping constructed above is defined for every connected component induced by $B(p')$. ∎

This result has direct implications for how the types of connected components may change between neighbouring nodes.

Corollary 6.5: *Let G be a graph and let (P, B) be a path-decomposition of G. Let $p, p' \in V(P)$ be nodes of (P, B) with p' being the left neighbour of p. Let $C_{p'}$ be a connected component of $G - B(p')$ whose vertices appear only in bags strictly left of p' and let C_p be a connected component of $G - B(p)$ whose vertices appear only in bags strictly left of p. If $C_{p'}$ is mapped to C_p by the function of Lemma 6.4, then the type of $C_{p'}$ is at most the type of C_p.*

Proof. Combine Lemma 6.4 and the fact that a subset of a connected component cannot be incident to more boundary sides than its superset, to obtain this result. ∎

For simplicity, we state the next lemmas only for rows, the corresponding statements for columns follow analogously. We show that, under certain conditions, the property of containing a vertex from each row or a vertex from each column is preserved. Moreover, we prove that some bags additionally contain at least two vertices from a single row. This will later be useful in showing that certain bags contain strictly more vertices than there are rows (see part (2) of Theorem 6.1). We begin with a lemma that simplifies the necessary preconditions and therefore is useful for establishing both properties.

Lemma 6.6: *Let Γ be a grid and let (P, B) be a path-decomposition of Γ. Let $p, p' \in V(P)$ be nodes of (P, B) with p' being the left neighbour of p. Let there be a connected component of type at least 3 induced by $B(p)$ that has its vertices in the bags strictly left of p. If $B(p)$ contains a vertex from each row, then the bags strictly to the left of p together contain at least one vertex from each row.*

Proof. We distinguish between type-3- and type-4-components. Let C be the type-3-component induced by $B(p)$ whose vertices lie in bags strictly to the left of p. W.l.o.g., assume that the boundary side not incident to C is a column. Otherwise, rotate the grid and consider the analogue statement for columns. Since $B(p)$ separates C from this boundary side, it follows that $B(p)$ contains at least one vertex from each row. Moreover, C also contains at least one vertex from each row, because C is connected and incident to both the upper and the lower boundary side.

For type-4-components the argument is similar. Let C be the type-4-component induced by $B(p)$ whose vertices lie in bags strictly to the left of p. W.l.o.g., assume that $B(p)$ contains a vertex from each row. Then, the connected component C also contains a vertex from each row, since C is connected and incident to both the top and bottom boundary side of the grid. ∎

We now apply this result to establish the desired properties. We first show that, under certain preconditions, the property of containing a vertex from each row is preserved when extending the subpath by one node. For this purpose, we use the following lemma from the survey of Bodlaender [Bod98], which formalizes the fact that a bag of a path-decomposition must separate all paths between vertices that lie on different sides of this bag.

Lemma 6.7 (Lemma 3(b) [Bod98])**:** *Let G be a graph and let (P, B) be a path-decomposition of G. Let $p_1, p_2, p_3 \in V(P)$ be nodes of (P, B) with p_2 between p_1 and p_3 in P. Then, for any path between a vertex $v \in B(p_1)$ and a vertex $w \in B(p_3)$, the bag $B(p_2)$ must contain at least one vertex from this path.*

We now use this to show our lemma.

Lemma 6.8: *Let Γ be a grid and let (P, B) be a path-decomposition of Γ. Let $p, p' \in V(P)$ be nodes of (P, B) with p' being the left neighbour of p. Let there be a connected component of type at least 3 induced by $B(p)$ that has its vertices in the bags strictly to the left of p. If $B(p)$ contains at least one vertex from each row, then $B(p')$ contains at least one vertex from each row.*

Proof. Due to Lemma 6.6, the bags strictly left of p together contain at least one vertex from each row. We now apply the lemma of Bodlaender to show that $B(p')$ contains a vertex from each row. Fix an arbitrary row and consider the following two cases. If there exists a vertex $v \in B(p)$ in this row that is also contained in $B(p')$, then this row is already covered. Otherwise, let w be a vertex of in this row that is contained in bags strictly to the left p and assume that it is not in $B(p')$. Also fix a vertex $v \in B(p)$ of this row. Then, there exists a path from $v \in B(p)$ to $w \in B(p^*)$ that only uses vertices from this row, where p^*, p' and p are nodes occurring in this order from left to right in P. By Lemma 6.7, the bag $B(p')$ must contain a vertex of this path and therefore a vertex from this row. ∎

Finally, we turn to the second property. Here, we prove that a bag containing a vertex from each row can only lose a vertex when transitioning to its neighbour, if that removed vertex is not the only vertex of this bag in this row.

Lemma 6.9: *Let Γ be a grid and let (P, B) be a path-decomposition of Γ. Let $p, p' \in V(P)$ be nodes of (P, B) with p' being the left neighbour of p such that $B(p) - B(p') \neq \emptyset$. Let there be a connected component of type at least 3 induced by $B(p)$ that has its vertices in the bags strictly to the left of p. If $B(p)$ contains at least one vertex from each row, then $B(p)$ must contain at least two vertices of some row.*

Proof. To establish this result, we identify an edge that would remain uncovered, if $B(p)$ contains exactly one vertex from each row.

Due to Lemma 6.6, the bags strictly to the left of p together contain at least one vertex from each row. Assume for sake of contradiction that $B(p)$ contains exactly one vertex from each row. Let $w \in B(p) - B(p')$ be a vertex that is removed when transitioning from $B(p)$ to $B(p')$. That is, w is only contained in $B(p)$ and bags strictly to the right of p. Consider the at most two neighbours of w that lie in the same row as w. Since $B(p)$ contains exactly one vertex from each row, each of them must appear only in bags either strictly to the left or strictly to the right of p. If one of them only appears in bags strictly to the left of p, then the edge between w and this neighbour is not covered by any bag, violating the edge-coverage property of path-decompositions. This contradicts (P, B) being a valid path-decomposition. If both neighbours appear only in bags strictly to the right of p, then all vertices of this row must only appear in bags strictly to the right of p . This is due to the fact that $B(p)$ only contains one vertex from this row and the connectivity property of path-decompositions forbids the path corresponding to this row from skipping a bag. This contradicts the assumption that the bags strictly to the left of p together contain at least one vertex from each row. Therefore, our assumption was false, and $B(p)$ must contain at least two vertices from some row. ∎

A lower bound on path-decompositions of grids. We are now ready to prove the main result of this chapter which we stated above in Theorem 6.1. This theorem establishes a lower bound on the sizes of bags in path-decompositions of grids. In the next chapter, we apply this theorem to path-decompositions of grids obtained by restricting a path-decomposition of a supergraph. Therefore, we must allow for arbitrary, possibly redundant, path-decompositions.

Theorem 6.1: *Let $\alpha < \beta$ be two integers. Let Γ be a grid with α rows and β columns and let (P, B) be a path-decomposition of Γ. Then, there exists a subpath $L = (p_\ell(\Gamma), \ldots, p_r(\Gamma))$ of P such that:*

1. *every bag in the subpath L has size at least α,*
2. *for every node p' of the subpath L with size exactly α the following holds: the closest node p'' of L to the right of p' with $B(p'') \neq B(p')$ has at least $\alpha + 1$ vertices in its bag,*
3. *there are at most α^2 vertices that appear only in bags strictly left of $p_\ell(\Gamma)$, and*
4. *there are at most α^2 vertices that appear only in bags strictly right of $p_r(\Gamma)$.*

Proof. The proof proceeds as follows. First, we identify a single bag of size at least α, which forms the initial subpath. We then iteratively extend this subpath using two extension rules, each of which preserves (1) and (2). Finally, we show that exhaustive application of these rules eventually triggers the termination rule, at which point Lemma 6.3 applies and yields (3) and (4).

Let p_c be a node whose bag contains at least one vertex from each row or contains at least one vertex from each column. By Claim 88A in the survey of Boedlaender [Bod98] such a node always exists. Since Γ has α rows, β columns and $\beta > \alpha$, the bag of p_c has size at least α. Consequently, this initial subpath satisfies (1) and also trivially satisfies (2). Next, we extent this initial subpath. We describe how we extend the current subpath $(p_\ell, \ldots, p_r)$ to the left. The same arguments apply symmetrically to the right. Let q bet the left neighbour of p_ℓ and consider the following rules. As mentioned before, we distinguish two extension rules. They differ in that the strong extension rule preserves the additional property that the current bag contains one vertex from each row or from each column.

R1 (Termination): If one of the following two conditions is satisfied, then terminate the process and return p_ℓ as the left endpoint. (i) There are no bags left of p_ℓ or (ii) $B(q)$ has size at most α and all connected components induced by $B(p_\ell)$ that have their vertices in the bags strictly to the left of p_ℓ are of type at most 2.

R2 (weak Extension): If the following condition is satisfied, then extend the subpath to $(q, p_\ell, \ldots, p_r)$. $B(q)$ has size at least $\alpha + 1$ and all connected components induced by $B(p_\ell)$ that have their vertices in the bags strictly to the left of p_ℓ are of type at most 2.

R3 (strong Extension): If the following condition is satisfied, then extend the subpath to $(q, p_\ell, \ldots, p_r)$. $B(p_\ell)$ induces a connected component of type at least 3 that has its vertices in the bags strictly to the left of p_ℓ.

First, observe that these rules cover all possible cases. Next, we analyse the order in which these rules are applied to understand the structure of the subpath during the extension process and to prepare the ground for establishing the correctness properties of the construction. We claim that the rules are applied in the following order. Starting from the initial subpath, the strong extension rule may be applied iteratively, then the weak extension rule may be applied several times, and finally the termination rule is applied exactly one. We justify this claim as follows. First, note that the termination rule R1 must eventually be applied, since each extension to the left moves the subpath closer to the leftmost node of P. Next, consider the situation where all connected components of the grid induced by $B(p_\ell)$ that are fully contained in bags strictly to the left of p_ℓ are of type at most 2. By Corollary 6.5, which relates the types of the connected components induced by $B(q)$ to the types of those induced by $B(p_\ell)$, the connected components induced by $B(q)$ that are fully contained in bags strictly to the left of q are also of type at most 2. Consequently, once the weak extension rule R2 has been applied, the strong extension rule can no longer be applied. Therefore, the order of rule applications is exactly as claimed.

It remains to show that the termination rule R1 guarantees that at most α^2 vertices appear only in bags strictly left of the subpath (property (3)) and that both extension rules keep property (1), that is every bag has size at least α, and property (2), that is at least every second distinct bag has size $\alpha + 1$. We begin with the termination rule R1. If there are no nodes left of p_ℓ, property (3) is satisfied trivially. So assume there are some nodes to the left of p_ℓ. Then, by definition of R1, all connected components of Γ induced by $B(p_\ell)$ that are contained in bags strictly to the left of p_ℓ are of type at most 2. Hence, since q is to the left of p_ℓ, similarly as above, all connected components of the grid induced by $B(q)$ that are fully contained in bags strictly to the left of q are of type at most 2. Therefore, we can apply Lemma 6.3 to the node q and can thus bound the size of these connected components together with $B(q)$ itself. As a result, the number of vertices that are contained exclusively in bags strictly to the left of $B(p_\ell)$ is bounded and property (3) holds.

Now consider the weak extension rule R2 and show that the properties (1) and (2) are satisfied for the neighbouring bag q. Since the weak extension rule applies, $B(q)$ must have size at least $\alpha + 1$, among other required properties. Therefore, the size constraint (1) is satisfied. Moreover, the condition for bags of size exactly α in (2) also holds trivially, since $B(q)$ does not have this size.

Finally, consider the strong extension rule R3, which is applied only if a type-3- or a type-4-component is induced. Here, we show the claim that both bags $B(p_\ell)$ and $B(q)$ contain at least one vertex from each row or from each column. To see this, note that the current subpath was obtained from the initial subpath by a sequence of strong extensions, as argued above. By Lemma 6.8, this property is preserved under strong extension. Hence, the claim holds. As a result of this claim, the bag $B(q)$ contains at least α vertices and the size constraint

(1) is satisfied. To verify the condition for bags of size exactly α in property (2), assume that $B(q)$ contains exactly α vertices. Then, $B(q)$ contains exactly one vertex from each row. We must show that the closest node to the right of q whose bag differs from $B(q)$ has size at least $\alpha + 1$. If $B(q) = B(p_\ell)$, then we are done, since $B(p_\ell)$ already satisfies property (2) and since a bag is distinct from $B(q)$ if and only if it is distinct from $B(p_\ell)$. If $B(q) \subsetneq B(p_\ell)$, we are also done, as $B(p_\ell)$ is a strictly larger bag immediately to the right of q. If $B(p_\ell) \subsetneq B(q)$, then $B(q)$ contains at least $\alpha + 1$ vertices, since $B(p_\ell)$ contains at least α vertices by property (1). This contradicts our assumption, and hence this case cannot occur. It remains to consider the case where $B(q) - B(p_\ell) \neq \emptyset$ and $B(p_\ell) - B(q) \neq \emptyset$. Applying Lemma 6.9, which addresses the removal of a vertex from a bag containing one vertex per row, to p_ℓ, we conclude that $B(p_\ell)$ contains at least two vertices from some row. Having at least two vertices in one row and at least one vertex in every other row implies that $B(p_\ell)$ contains at least $\alpha + 1$ vertices. Hence, we identified the strictly larger bag immediately to the right of q. Therefore, property (2) is maintained by the strong extension rule.

We have thus shown that both extension rules preserve properties (1) and (2), and that the termination rule guarantees properties (3) and (4). Furthermore, we verified that the initial subpath satisfies properties (1) and (2), that the rules cover all possible cases and that the termination rule must eventually be applied. Therefore, the theorem holds. ∎

7 A lower bound for clique-partitioned Treewidth

In Chapter 5, we presented two approaches for computing the cp-treewidth of a graph. To asses the quality of these algorithms and to determine whether they are asymptotically tight, we turn to exploring conditional lower bounds. In this section, we show that a polynomial-algorithm for computing cp-treewidth is unlikely to exist. More precisely, we prove that cp-Treewidth is $\mathcal{NP}$-hard. We further show that, with minor modifications to the proof, this hardness extends to approximating cp-Treewidth within any additive constant. Finally, we discuss how this proof relates to a variant of cp-treewidth called gp-treewidth. Throughout this chapter, we use the multiplicative formulation of cp-weight introduced in the preliminaries, since we only concerned with relative rather than absolute weights.

Our proof proceeds via a reduction from Pathwidth to cp-treewidth. The reduction is split into two parts and uses an intermediate problem which we call Skew-sensitive Pathwidth. In this problem the task is to compute a newly defined parameter that is based on path-decompositions. As this intermediate problem is rather technical and not particularly intuitive, we present the reduction steps in a different order to improve readability.

We first prove that Pathwidth remains $\mathcal{NP}$-hard on a graph class that meets some structural requirements. We then briefly summarise the reduction from Pathwidth to Treewidth introduced by Bodlaender et al. [Bod+23]. Next, we modify the metric on bags to obtain a reduction from our intermediate problem to cp-Treewidth. After this, we introduce the formal definition of the intermediate problem with the necessary context already in place. Finally, we give a reduction from Pathwidth to the intermediate problem. Combined with the hardness result for Pathwidth shown by Bodlaender et al. [Bod+23], this chain of reductions implies the $\mathcal{NP}$-hardness of cp-Treewidth.

7.1 Pathwidth remains hard for graphs with additional structure

In the next two sections, we rely on certain structural properties of the graphs in our instances. In this section, we show that any input graph can be transformed so that these properties hold, while preserving the pathwidth. From a formal perspective, this constitutes a reduction from Pathwidth to Pathwidth restricted to a particular graph class. We therefore prove the following lemma.

Lemma 7.1: *Let $G = (V, E)$ be a graph. Then, we can compute a graph G' in time $\mathcal{O}(|V|^2)$ that satisfies the following conditions:*

1. *the pathwidth of G' is exactly $\mathrm{pw}(G) + 1$,*
2. *G' has $\mathcal{O}(|V|^2)$ vertices,*
3. $\mathrm{pw}(G')^2 \leq \frac{|V(G')|}{10}$,

(a) (b)

Figure 7.1: Example of applying our construction to a path (a, b, c) of length 3. Part (a) shows the graph G' where the universal vertex u and the path $(v_1, v_2, \dots)$ have been added. Part (b) illustrates the path-decomposition (P', B') of G' constructed during the proof.

4 *every vertex of G' has at most $\frac{|V(G')|}{10}$ neighbours, and*

5 *there exists an algorithm that computes two vertices $a, b \in V(G')$ in time $\mathcal{O}(|V(G')|)$ such that there exists an optimal path-decomposition $(P = (p_1, \dots, p_q), B)$ of G' that has $a \in B(p_1)$ and $b \in B(p_q)$.*

Proof. The proof proceeds in two steps. First, we add a single vertex so that the pathwidth increases by exactly one. Then, we increase the number of vertices while introducing only few additional edges.

Let (P, B) be a path-decomposition of G of width $\mathrm{pw}(G)$. We first add a universal vertex u. This increases the pathwidth of G by exactly one [CLTV17]. Since u is adjacent to all vertices of G, we obtain a path-decomposition (P^*, B^*) of width $\mathrm{pw}(G) + 1$ for the new graph by adding u to every bag of (P, B). In particular, u appears in the first bag, which allows us to attach additional vertices at one end of the path-decomposition. To construct G', we attach a path $(v_1, \dots, v_{10|V|^2})$ consisting of $10|V|^2$ vertices to u. Concatenating an optimal path-decomposition of this added path, the bag $\{v_1, u\}$, and (P^*, B^*) yields a path-decomposition (P', B') of G'. An example for the construction of G' and (P', B') is illustrated in Figure 7.1.

We now verify that all required conditions are satisfied. Since all paths have pathwidth one, the decomposition (P', B') has width $\mathrm{pw}(G) + 1$. As pathwidth is monotone and $G + u$ is an induced subgraph of G' with pathwidth $\mathrm{pw}(G) + 1$, it follows that G' has pathwidth at least $\mathrm{pw}(G) + 1$. Therefore, G' has pathwidth exactly $w + 1$. Next, we observe that G' contains $10|V|^2 + |V| + 1$ vertices. Since $\mathrm{pw}(G) \leq |V| - 1$, we verify that $10\,\mathrm{pw}(G')^2 = 10(\mathrm{pw}(G) + 1)^2 \leq 10|V|^2 \leq |V(G')|$. In the next step we bound the number of neighbours of each vertex. For any vertex $v \in V + u$, its neighbours lie in $V + u + v_1$ and thus v has at most $|V| + 2 \leq \frac{|V(G')|}{10}$ neighbours. Each vertex of the added path has at most two neighbours. So requirement (4) is satisfied. Note that for $a = v_{10|V|^2}$ and $b = u$ the path-decomposition (P', B') of G' constructed above satisfies the conditions of requirement (5). We can identify those two vertices in linear time by finding the added path. Hence, the graph G' satisfies all requirements.

Finally, note that adding $10|V|^2 + 1$ vertices to the graph can be carried out in time $\mathcal{O}(|V|^2)$. ∎

We define $\mathcal{G}$ as the graph class of all connected graphs that satisfy conditions (3), (4) and (5) of Lemma 7.1. Since PATHWIDTH is $\mathcal{NP}$-hard for connected graphs [ACP87 | Bod+23], we get:

Corollary 7.2: *PATHWIDTH is $\mathcal{NP}$-hard for graphs of $\mathcal{G}$.*

We take this result as the initial step in our chain of reductions. Hence, in the remainder of this chapter we assume that all graphs under consideration are drawn from the class $\mathcal{G}$.

7.2 From Pathwidth to Treewidth

In this subsection, we present the reduction from Pathwidth to Treewidth of Bodlaender et al. [Bod+23]. An example of this reduction can be seen in Figure 7.2.

We are given an instance of Pathwidth, consisting of a connected graph $G = (V, E)$ and an integer w. The key idea of the reduction is to construct a graph H_G whose pathwidth is a function of the pathwidth of G, and for which there is an optimal tree-decomposition that is a path. In their paper Bodlaender et al. [Bod+23] show that co-bipartite graphs have precisely this property. For completeness we provide a sketch of the proof of this fact.

Lemma 7.3 (Lemma 2.2 of Bodlaender et al. [Bod+23])**:** *Let $G = (V, E)$ be a co-bipartite graph, with $V = X \cup Y$ where X and Y are cliques. Then:*

- $\mathrm{tw}(G) = \mathrm{pw}(G)$.
- *G admits a path-decomposition $(P = (p_1, \ldots, p_q), B)$ with width $\mathrm{tw}(G)$ such that $X \subseteq B(p_1)$ and $Y \subseteq B(p_q)$.*

Proof Sketch. Let (T, B) be an optimal tree-decomposition of G. We first observe that there is a bag containing X and that there is a bag containing Y as these are cliques (see Lemma 3.6). Additionally, we note that leaves whose bags are subsets of the bag of their unique neighbour can be removed without changing the width of the decomposition. Repeating this removal results in the desired path-decomposition. ∎

In the remainder of this section we construct such a graph and show how its pathwidth relates to the pathwidth of the original graph. That is we prove the following lemma.

Lemma 7.4: *Let $G = (V, E)$ be a graph. Then, we can construct in time $\mathcal{O}(|V(G)|^2)$ a graph H_G such that:*

1. *H_G is co-bipartite,*
2. *$\mathrm{tw}(H_G) = \mathrm{pw}(H_G) = |V| + \mathrm{pw}(G)$ and*
3. *$|V(H_G)| = 2 \cdot |V(G)|$.*

We construct the co-bipartite graph H_G as follows. For every $v \in V$, we introduce two copies, denoted v and v', in H_G. Let $V' = \{v' | v \in V\}$. To ensure that H_G is co-bipartite, we add edges to turn both V and V' into cliques. For every $v \in V$, we add the edge vv', and for every $uv \in E$, we add the edges uv' and $u'v$. This can be done in time $\mathcal{O}(|V(G)|^2)$. By Lemma 7.3, we know that $\mathrm{tw}(H_G) = \mathrm{pw}(H_G)$. Thus, it remains to show $\mathrm{pw}(H_G) = |V| + \mathrm{pw}(G)$. We prove this in two steps. First, we show that $\mathrm{pw}(H_G) \leq |V| + \mathrm{pw}(G)$ by constructing a sufficiently small path-decomposition of H_G from a path-decomposition of G. Next, we show the reverse inequality $\mathrm{pw}(H_G) \geq |V| + \mathrm{pw}(G)$ by proving the equivalent statement $\mathrm{pw}(G) \leq \mathrm{pw}(H_G) - |V|$. Here, we construct a sufficiently small path-decomposition of G from a path-decomposition of H_G.

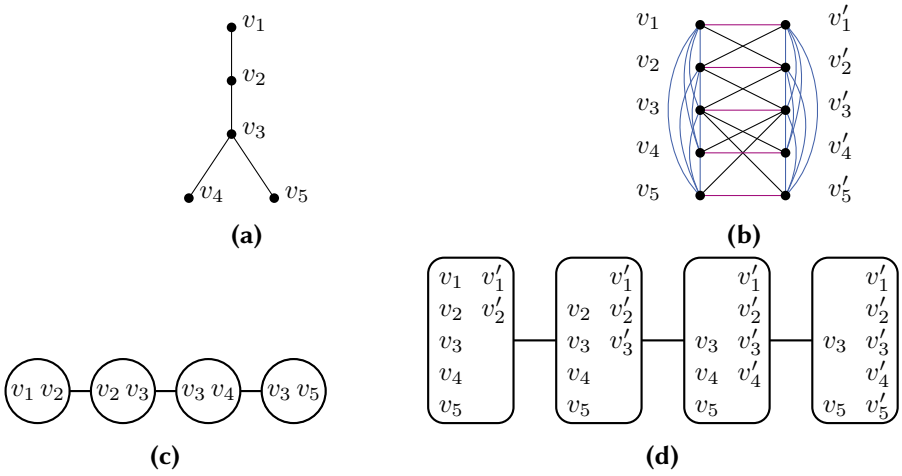


Figure 7.2: Example of applying our construction to a graph G. Part (a) shows the graph. Part (b) depicts the graph H_G with edges inherited from G coloured black, clique edges coloured blue and matching edges coloured purple. Part (c) visualizes a path-decomposition of G. Part (d) illustrates the corresponding path-decomposition of H_G.

A path-decomposition of H_G. Let (P_G, B_G) be a path-decomposition of G of width $\mathrm{pw}(G)$. We construct a path-decomposition (P_H, B_H) of H_G from it. Set $P_H = P_G = (p_1, \ldots, p_q)$, so the decompositions differ only in the contents of their bags. Intuitively, we construct the bags of P_H as follows. Whenever a bag $B_G(p)$ contains a vertex $v \in V$, we insert both v and v' into $B_H(p)$. We then propagate v to all bags with smaller index and v' to all bags with larger index. More formally, for each $v \in V$ and $p_i \in V(P_G)$, such that there is a $j \geq i$ with $v \in B_G(p_j)$, add v to $B_H(p_i)$. Similarly, for each $v \in V$ and $p_i \in V(P_G)$, such that there is a $j \leq i$ with $v \in B_G(p_j)$, add v' to $B_H(p_i)$. It is straightforward to verify that this constitutes a valid path-decomposition of H_G and that each bag $B_H(p)$ contains exactly $|V|$ additional vertices compared to the corresponding bag $B_G(p)$. Hence, the width of this path-decomposition is $|V| + \mathrm{pw}(G)$. We omitted some details in this proof as they are not necessarily needed for understanding our modified reduction. These omitted details can be found in the work of Bodlaender et al. [Bod+23].

A path-decomposition of G. We begin by showing that there always is an optimal path-decomposition with a certain structure.

Corollary 7.5: *Let $G = (V, E)$ be a co-bipartite graph, with $V = X \cup Y$ where X and Y are cliques. Then, there exists an optimal path-decomposition $(P = (p_1, \ldots, p_q), B)$ of G such that for each edge xy with $x \in X$ and $y \in Y$, there is a node $p_i \in V(P)$ with all bags $B(p_1), \ldots, B(p_i)$ containing x and all bags $B(p_i), \ldots, B(p_q)$ containing y.*

Proof. By Lemma 7.3 there is an optimal path-decomposition $(P = (p_1, \ldots, p_q), B)$ with $X \subseteq B(p_1)$ and $Y \subseteq B(p_q)$. Due to the edge coverage-property of path-decompositions there must be a bag p_i in (P, B) containing both x and y for the edge xy in this path-decomposition. We note that x is in p_i as well as p_1 and that y is in p_i as well as p_q and apply the connectedness property of path-decompositions to obtain the desired result. ∎

Let (P_H, B_H) be a path-decomposition of H_G of width $\mathrm{pw}(H)$ that has the structure described in Corollary 7.5. We construct a path-decomposition (P_G, B_G) of G from it. We set $P_G = P_H$, so the decompositions differ only in the content of their bags. Furthermore, for each node $p \in V(P_G)$, we add v to $B_G(p)$ if and only if both v and v' are in $B_H(p)$. As before, it is straightforward to verify that this constitutes a valid path-decomposition of G and that each bag $B_G(p)$ contains exactly $|V|$ fewer vertices than the corresponding bag $B_H(p)$. So the above construction changes the size of each bag by exactly $|V|$. As before, the omitted details can be found in the work of Bodlaender et al. [Bod+23].

7.3 Observations on clique-partitions and a modified reduction

In this subsection, we modify the reduction from Pathwidth to Treewidth so as to obtain a reduction from a new intermediate problem to cp-Treewidth. We begin by adapting Lemma 7.3 to the clique-partioned variants of the parameters.

Corollary 7.6 (Lemma 7.3 for cp-treewidth)**:** *Let $G = (V, E)$ be a co-bipartite graph, with $V = X \cup Y$ where X and Y are cliques. Then:*

- $\mathrm{cptw}(G) = \mathrm{cppw}(G)$.
- *G admits a path-decomposition $(P = (p_1, \ldots, p_q), B)$ with weight $\mathrm{cptw}(G)$ such that $X \subseteq B(p_1)$ and $Y \subseteq B(p_q)$.*

Proof Sketch. Let (T, B) be an optimal cp-tree-decomposition of G. As before, there is a bag containing X and there is a bag containing Y as these are cliques (see Lemma 3.6). Again, we note that leaves whose bags are subsets of the bag of their unique neighbour can be removed without changing the weight of the decomposition. This is due to the fact that the weight of a clique-partition is monotone under taking of induced subgraphs. Repeating this removal results in the desired cp-path-decomposition. ∎

Let G be a graph from the graph class $\mathcal{G}$ and let H_G be be constructed from G as described in the last subsection. As a result of this corollary, the only modification we need in our reduction to produce an instance of cp-Treewidth is to analyse the clique-partition of each bag in the path-decomposition (P_H, B_H) of H_G. As before, we may assume that the path-decomposition has the structure described in Corollary 7.5. Thus, each bag contains some vertices from V and some from V' and has total size $|V| + \mathrm{pw}(G) + 1$.

In the following, we use that due to requirement (4) of Lemma 7.1 the number of neighbours of each vertex in G is bounded. With this, we bound some clique sizes in H_G. Remember, that there are only two types of edges between V and V' in H_G. The first type consists of the edges vv' connecting a vertex to its copy. The second type consists of edges uv' representing an edge uv in G. So each vertex of V is adjacent to at most $\frac{|V(G)|}{10} + 1$ vertices of V'. Similarly, each vertex of V' is adjacent to at most $\frac{|V(G)|}{10} + 1$ vertices of V. Due to this, every clique C in H_G that is no subset of either V or V' satisfies $|C| \leq 2 \cdot \left(\frac{|V(G)|}{10} + 1\right) = \frac{|V(G)|}{5} + 2$. So such a clique is smaller than the number of vertices by a constant factor.

Using this requirement, we can describe the structure of optimal clique-partitions of bags exactly.

Lemma 7.7: *Let $G \in \mathcal{G}$ be a graph and let H_G be the corresponding auxiliary graph. Let $p \in V(P_H)$ be a node of a path-decomposition (P_H, B_H) of H_G with the structure from Corollary 7.5. Then, any weight-minimal clique-partition of $B_H(p)$ partitions $B_H(p)$ into the two cliques $B_H(p) \cap V$ and $B_H(p) \cap V'$.*

Proof. Let $\mathcal{P}$ be a weight-minimal clique-partition of $B_H(p)$. Define $C_1 = B_H(p) \cap V$ and $C_2 = B_H(p) \cap V'$ as the restrictions of V and V' to the current bag. Note, that both sets are non-empty due to the structure of path-decompositions of H_G introduced in Corollary 7.5. Assume, for sake of contradiction, that $\mathcal{P}$ does not partition $B_H(p)$ into the two cliques C_1 and C_2. We analyse the sizes of the different cliques and derive a contradiction using Lemma 3.5. First, we consider the clique sizes in the partition into the cliques C_1 and C_2. W.l.o.g., assume that $|C_1| \geq |C_2|$. Then, $|C_1| \geq \frac{|V(G)|}{2}$, since the bag $B_H(p)$ contains at least $|V(G)| + 1$ vertices. Then, we consider the sizes in $\mathcal{P}$. Any clique that contains vertices from both C_1 and C_2 has size at most $\frac{|V(G)|}{5} + 2$, as argued above. Hence, all such cliques are strictly smaller than C_1. In particular, $\mathcal{P}$ contains at least two cliques. All remaining cliques contain vertices from only one of the sets C_1 or C_2 and are therefore subsets of one of them. Consequently, their sizes are also bounded by the size of C_1. In summary, we have shown that $\mathcal{P}$ contains at least two cliques and that all cliques in this partition are smaller than C_1. In particular, the largest clique in $\mathcal{P}$ is smaller than C_1.

We may now apply Lemma 3.5. We set $k = 2$, $s_1 = |C_1|$, and $s_2 = |C_2|$, and use the non-increasing sequence of clique sizes induced by $\mathcal{P}$ for $\langle r_1, \ldots, r_\ell \rangle$. Since all preconditions are satisfied, the corollary implies that $\mathcal{P}$ has strictly larger weight than the partition into C_1 and C_2. This contradicts the assumption that $\mathcal{P}$ is weight-minimal, and therefore the lemma follows. ∎

We thus know the structure of an optimal clique-partition of a bag $B_H(p_i)$. However, to compute the weight of a bag $B_H(p_i)$, we additionally need know how many vertices from V and how many from V' it contains. For this, we consider the corresponding bag $B_G(p_i)$ in the corresponding path-decomposition $(P_G = (p_1, \ldots, p_q), B_G)$ of G. Let $b(p_i)$ be the size of $B_G(p_i)$. Let $\ell(p_i)$ the number of vertices that appear in some bag among $B_G(p_1) - B_G(p_i), \ldots, B_G(p_{i-1}) - B_G(p_i)$. Let $r(p_i)$ be the number of vertices that appear in some bag among $B_G(p_{i+1}) - B_G(p_i), \ldots, B_G(p_q) - B_G(p_i)$. We can use these to compute the weight of the bag.

Lemma 7.8: *Let $G \in \mathcal{G}$ be a graph and let H_G be the corresponding auxiliary graph. Also let $(P_H = (p_1, \ldots, p_q), B_H)$ be a path-decomposition of H_G with the structure from Corollary 7.5 and let $(P_G = (p_1, \ldots, p_q), B_G)$ be the corresponding path-decomposition of G. Let $p \in V(P_H)$ be a node. Then, the weight of the optimal clique-partition of the bag $B_H(p)$ is $(\ell(p) + b(p) + 1) \cdot (r(p) + b(p) + 1)$.*

Proof. For each vertex $v \in V$ that is contained in one of the bags $B_G(p_1), \ldots, B_G(p)$, the vertex v' is in $B_H(p)$. Similarly, for each vertex $v \in V$ that is contained in one of the bags $B_G(p), \ldots, B_G(p_q)$, the vertex v is in $B_H(p)$. Then, $B_H(p)$ contains $\ell(p) + b(p)$ vertices of V' and $r(p) + b(p)$ vertices of V. By Lemma 7.7 the clique-partition into the subsets of V and V' is optimal. ∎

We observe that this weight depends only on properties of the corresponding path-decomposition of G. This allows us to define a new cost-metric on path-decompositions of G that directly mirrors the weight of bags in path-decompositions of H_G. For a node p in a path-decomposition (P_G, B_G) of G, we define the *cost* of this node as $(\ell(p)+b(p)+1)\cdot(r(p)+b(p)+1)$.

The *cost of a path-decomposition* is the maximum cost of any of its nodes. The *skew-sensitive pathwidth* of G is the minimum cost over all the path-decompositions of G. We now define the Skew-sensitive Pathwidth problem as the decision problem asking whether the skew-sensitive pathwidth of a given graph is at most an integer k.

We conclude this section by summarizing the reduction from Skew-sensitive Pathwidth to cp-Treewidth.

Lemma 7.9: *There exists a polynomial-reduction from Skew-sensitive Pathwidth on graphs of $\mathcal{G}$ to cp-Treewidth.*

Proof. Given an instance (G, k) of Skew-sensitive Pathwidth on graphs of $\mathcal{G}$, we construct the graph H_G for G in time $\mathcal{O}(n^2)$ and keep the integer k unchanged. Constructing the auxiliary graph doubles the number of vertices.

If G has skew-sensitive pathwidth at most k, then the reduction produces a graph H_G with cp-treewidth at most k. This follows by constructing a path-decomposition of H_G from a path-decomposition of G with cost k, as described in the last subsection. Then, for every bag, the weight in the decomposition of H_G equals the corresponding cost in the decomposition of G by Lemma 7.8.

Conversely, if the constructed graph H_G has cp-treewidth at most k, then G has skew-sensitive-pathwidth at most k. To see this, we first observe that, by Corollary 7.6, there exists an optimal cp-tree-decomposition that is a path. We then construct a path-decomposition of G from a path-decomposition of H_G with weight k, as described in the last subsection. Finally, for every bag, the cost in the decomposition of G equals the corresponding weight in the decomposition of H_G by Lemma 7.8. ∎

7.4 From Pathwidth to Skew-sensitive pathwidth

In this section we present a reduction from Pathwidth on graphs of $\mathcal{G}$ to Skew-sensitive Pathwidth on graphs of $\mathcal{G}$. We first present our construction to transform an instance of Pathwidth into an instance of Skew-sensitive Pathwidth. Then, we prove correctness in two parts.

The reduction. Let $G = (V, E) \in \mathcal{G}$ be a n-vertex graph and let w be an integer which is the candidate for the width of G. We construct a graph $G_{a,b}$, which, together with an integer k whose value we choose later, forms the instance of Skew-sensitive Pathwidth. The graph $G_{a,b}$ consists of the original graph G together with two grids Γ_a and Γ_b. Each grid has w rows and contains n^c vertices in total, where c is a constant chosen later. We choose the number of columns s in each grid accordingly to meet the vertex count. By requirement (3) of Lemma 7.1, namely $w^2 \leq \frac{n}{10}$, each grid has strictly fewer rows than columns. Let a and b be the special vertices of requirement (5) in Lemma 7.1. Remember, that there is an optimal path-decomposition of G such that a is in the first and b is in the last bag. We then connect the w vertices along one of the two smaller boundary sides of Γ_a to a, and similarly the boundary vertices of Γ_b to b. Here, the core idea is that we can place the two grids at different ends of the path-decomposition of G. An example of this construction can be seen in Figure 7.3.

Structural properties of $G_{a,b}$. Since we apply the reduction of Lemma 7.9 immediately after this reduction in our chain of reductions, we must ensure that the constructed graph $G_{a,b}$ satisfies all its preconditions. To this end, we show that $G_{a,b}$ belongs to the class $\mathcal{G}$. First, note that G is connected. Next, requirement (4) holds, since adding the grids increases the total

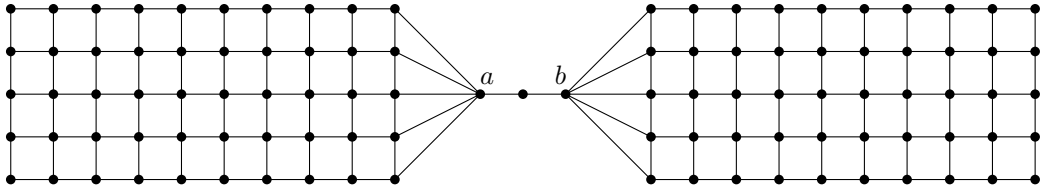


Figure 7.3: The graph $G_{a,b}$ for G being a path of length 3 and $w = 5$.

number of vertices but does not increase the size of any vertex neighbourhood beyond the bounds required for $\mathcal{G}$. Furthermore, any vertex of the boundary with size w of Γ_a that is not adjacent to a can serve as the first vertex in requirement (5) of Lemma 7.1. The second vertex can be chosen analogously from Γ_b. It remains to verify requirement (3). For this, we must bound the pathwidth of $G_{a,b}$ to ensure that the required inequality holds. As we construct a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ with width w in the next subsection, we defer this argument until then.

Together with the deferred argument, we obtain the following lemma.

Lemma 7.10: *Let $G \in \mathcal{G}$ be a graph. Then, the corresponding auxiliary graph $G_{a,b}$ constructed above is contained in the graph class $\mathcal{G}$.*

On correctness. For correctness we need to show two parts. First, we find an upper bound for the skew-sensitive pathwidth for the graph $G_{a,b}$, under the precondition that the original graph G has pathwidth at most w. This upper bound is then used as the integer value k. Next, we show that if the graph $G_{a,b}$ has a path-decomposition with cost less than this upper bound k, then we can find a path-decomposition of G with width w. Together these results show that the reduction is correct.

7.4.1 An upper bound for the skew-sensitive pathwidth of $G_{a,b}$

Let $G = (V, E) \in \mathcal{G}$ be a graph with $\mathrm{pw}(G) \leq w$. In this section we find an upper bound for the skew-sensitive pathwidth of the graph $G_{a,b}$ constructed above.

We structure this section as follows. First, we quote a lemma describing the pathwidth of grids. We then use path-decompositions of Γ_a, Γ_b and G to construct a path-decomposition of $G_{a,b}$. To find an upper bound for the cost of this path-decomposition we identify the bag with maximum cost, show that all other bags have smaller cost than this bag and finally find an upper bound for the cost of this maximum bag.

A path-decomposition of $G_{a,b}$. We begin the construction of the path-decomposition by considering the pathwidth of grids in general.

Lemma 7.11 (Corollary 89 of Boedlaender [Bod98])**:** *Let Γ be a grid with α rows and β columns. Then, the pathwidth of Γ is $\min(\alpha, \beta)$.*

Since $\mathrm{pw}(G) \leq w$, requirement (5) of Lemma 7.1 guarantees, that there is a path-decomposition (P_G, B_G) of G of width w that has a in its first bag and b in its last bag. By the previous lemma, there exist path-decompositions $(P_{\Gamma_a}, B_{\Gamma_a})$ of Γ_a and $(P_{\Gamma_b}, B_{\Gamma_b})$ of Γ_b, each of width w. Then, define the following path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ which is illustrated in

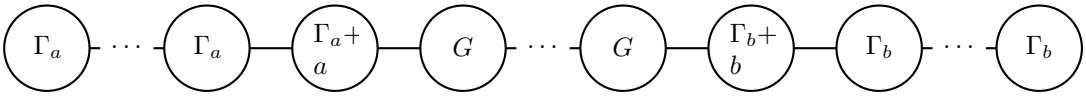


Figure 7.4: The path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$. Each visualization of a bag is labelled with a superset of the vertices contained in this bag.

Figure 7.4. Set the path to be $P_{a,b} = (P_{\Gamma_a}, p_a, P_G, p_b, P_{\Gamma_b})$. Here, the bag $B_{a,b}(p_a)$ contains the vertex a and all its neighbours in Γ_a, and the bag $B_{a,b}(p_b)$ contains b and all its neighbours in Γ_b. For all other nodes in the path, we use the bags from the corresponding path-decompositions of Γ_a, G, and Γ_b.

Note that this path-decomposition has width w and thus completes the argument deferred earlier. Consequently, $G_{a,b}$ indeed belongs to the class $\mathcal{G}$ and Lemma 7.10 holds.

Bounding the cost. We now show that every bag of the path-decomposition $(P_{a,b}, B_{a,b})$ has cost at most k. In the following we identify the bag with the maximum cost and then find an upper bound for the cost of this bag.
We can assume that each bag of $(P_{a,b}, B_{a,b})$ has size exactly $w + 1$, since all bags have size at most $w + 1$ and we can add vertices to bags of smaller size if necessary. So we know that all bags of $(P_{a,b}, B_{a,b})$ have the same size. We thus need some tools to compare the cost of bags with the same size. For this we introduce some terminology and then prove an auxiliary lemma. Remember, that we used $b(p)$ for the bag size of a node p and that we introduced $\ell(p)$ and $r(p)$ to describe the vertex counts in the components of $P_{a,b} - p$. For a node p in $(P_{a,b}, B_{a,b})$ we call the difference $|\ell(p) - r(p)|$ the *imbalance* of p. In the following lemma we show that among a sequence of bags of the same size the bag with the smallest imbalance is the one with maximum cost. Keep in mind that we defined the cost of a node p as $(\ell(p) + b(p) + 1) \cdot (r(p) + b(p) + 1)$.

Lemma 7.12: *Let $G = (V, E) \in \mathcal{G}$ be a graph and let (P, B) be a path-decomposition of G. Also let $p, p' \in V(P)$ be two nodes of (P, B) with $b(p) = b(p')$. If $|\ell(p) - r(p)| > |\ell(p') - r(p')|$, then the cost of p is strictly smaller than the cost of p'.*

Proof. W.l.o.g assume that $\ell(p) > r(p)$ and $\ell(p') > r(p')$ hold. Set $v = \ell(p) + b(p), x = r(p) + b(p), y = \ell(p') + b(p), z = r(p') + b(p)$. Then, the cost of p is $(v + 1) \cdot (x + 1)$ and the cost of p' is $(y + 1) \cdot (z + 1)$. By applying Lemma 3.4 we obtain the desired result. The lemma requires $|v - x| > |y - z|$, which is met as $|\ell(p) - r(p)| > |\ell(p') - r(p')|$ and $b(p) = b(p')$. ∎

Due to Lemma 7.12 and the fact that all bags in $(P_{a,b}, B_{a,b})$ have the same size, the node p_i of $(P_{a,b}, B_{a,b})$ with minimum imbalance has the maximum cost. So it only remains to find an upper bound for the cost of p_i. For an upper bound on the cost we assume this node to have imbalance $|\ell(p_i) - r(p_i)| = 0$ as by Lemma 7.12 any larger difference decreases the cost. Remember, that $G_{a,b}$ has $2n^c + n$ vertices and that $B_{a,b}(p_i)$ contains exactly $w + 1$ vertices. So there are exactly $2n^c + n - (w + 1)$ vertices not in the bag of p_i. As the imbalance of p_i is zero,

exactly half of these vertices are contained in bags left of p_i and exactly half are in bags to its right. So, this bag has $\ell(p_i) = r(p_i) = n^c + \frac{n}{2} - \frac{w+1}{2}$ and $b(p_i) = w + 1$. Therefore, the cost $(\ell(p_i) + b(p_i) + 1) \cdot (r(p_i) + b(p_i) + 1)$ of this bag is:

$$\begin{aligned}
&\left(\left(n^c + \frac{n}{2} - \frac{w+1}{2}\right) + (w+1) + 1\right) \cdot \left(\left(n^c + \frac{n}{2} - \frac{w+1}{2}\right) + (w+1) + 1\right) \\
&= \left(n^c + \frac{n}{2} + \frac{w+1}{2} + 1\right) \cdot \left(n^c + \frac{n}{2} + \frac{w+1}{2} + 1\right) \\
&= n^{2c} + n^{c+1} + n^c \cdot w + 3n^c + \frac{n^2}{4} + \frac{n \cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4}.
\end{aligned}$$

Note, that this forms an upper bound due to the assumption of perfect balance. Consequently, we have bounded the cost of the bag p_i with the maximum cost and thus the cost of all bags. We summarize this part of the correctness proof in the following lemma.

Lemma 7.13: *Let $G = (V, E) \in \mathcal{G}$ be a graph of pathwidth at most w and let $G_{a,b}$ be the auxiliary graph constructed above. Then, there exists a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ with cost at most $n^{2c} + n^{c+1} + n^c \cdot w + 3n^c + \frac{n^2}{4} + \frac{n \cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4}$.*

We use this upper bound as the integer value k in the SKEW-SENSITIVE PATHWIDTH instance we construct. So, given a graph of pathwidth at most w the reduction returns a graph of skew-sensitive pathwidth at most k. For the correctness it remains to show that any graph with skew-sensitive pathwidth at most k originates from a graph of pathwidth at most w. This is done in the next subsection.

7.4.2 An upper bound for the pathwidth of G

Let $G = (V, E) \in \mathcal{G}$ be a graph and let w be an integer. In this subsection we show that if the graph $G_{a,b}$ has skew-sensitive pathwidth at most $k = n^{2c} + n^{c+1} + n^c \cdot w + 3n^c + \frac{n^2}{4} + \frac{n \cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4}$, then the original graph G has pathwidth at most w. Note that this value of k matches the upper bound for the cost shown in Lemma 7.13 and depends on the integer w. Throughout this subsection we fix an integer w and consider the value k for this w.

We structure the proof as follows. First, we show that any bag of any path-decomposition of cost at most k whose imbalance is at most $3n$ has size at most $w + 1$. Once we establish that there exists a path-decomposition such that every vertex of G appears only in such bags, the claimed result follows immediately. To this end, we first introduce several auxiliary results that describe how imbalance behaves in our special graph constructions. Using these results, we identify a path-decomposition with a subpath R that contains exactly $3n$ vertices and consists solely of bags whose imbalance is at most $3n$. By the first step, all bags of R have size at most $w + 1$. In the next step, we use the known size of R and its position within the path-decomposition to show that every vertex of G appears only in bags of R. Finally, combining the first and last step, we conclude that all vertices of G occur only in bags of size $w + 1$. This yields a path-decomposition of G with width $w + 1$.

An upper bound for the size of bags with imbalance at most 3n. In the first step, we show that all bags with imbalance less than $3n$ have size at most $w + 1$. We begin with an observation on the behaviour of the cost function and then prove the main lemma of this step.

Observation 7.14: *Let p be a node of a path-decomposition. Then, the cost-function used in SKEW-SENSTIVE PATHWIDTH is monotone in $b(p)$. That is increasing $b(p)$ increases the cost of p, even if this decreases $\ell(p)$ or $r(p)$.*

Proof. To see this we compare the cost of p to the cost of p with one vertex moved from left of p into the bag of p. That is

$$\begin{aligned}&((\ell(p)-1)+(b(p)+1)+1)\cdot(r(p)+(b(p)+1)+1)\\&>(\ell(p)+b(p)+1)\cdot(r(p)+b(p)+1)\,.\end{aligned}$$

■

We use this result to bound the size of bags.

Lemma 7.15: *Let $G \in \mathcal{G}$ be a graph, let $G_{a,b}$ be the corresponding auxiliary graph constructed above and let k be as chosen above. Let $(P_{a,b}, B_{a,b})$ be a path-decomposition of $G_{a,b}$ with cost at most k. Let $p \in V(P_{a,b})$ be a node with imbalance at most $3n$. Then, $B_{a,b}(p)$ has size at most $w+1$.*

Proof. We show that if $B_{a,b}(p)$ has size more than $w+1$, then it has cost more than k. This would contradict $(P_{a,b}, B_{a,b})$ having cost k.

First, note that by Lemma 7.12 a smaller imbalance leads to a larger cost. Thus, it suffices to show that the cost of this bag already exceeds k for imbalance exactly $3n$. We thus assume that p has imbalance exactly $3n$. By definition, the imbalance of p fixes the distribution of vertices on either side of this bag, up to symmetry. Specifically, the number of vertices exclusively in bags to the left of p is $n^c - n - \frac{|B_{a,b}(p)|}{2}$ and number of vertices exclusively in bags to the right is $n^c + 2n - \frac{|B_{a,b}(p)|}{2}$. These values follow from the fact that $G_{a,b}$ contains $2n^c + n$ vertices and that $\ell(p) + b(p) + r(p) = |V(G_{a,b})|$.

To show that a bag exceeds cost k for size at least $w+2$, it suffices to show that the cost of this bag already exceeds k for size exactly $w+2$ by Observation 7.14. So assume for sake of contradiction that $B_{a,b}(p)$ has size exactly $w+2$. Then, the cost $(\ell(p)+b(p)+1)\cdot(r(p)+b(p)+1)$ of p is:

$$\begin{aligned}&\left(\left(n^c - n - \frac{w+2}{2}\right) + (w+2) + 1\right)\cdot\left(\left(n^c + n + n - \frac{w+2}{2}\right) + (w+2) + 1\right)\\&= \left(n^c - n + \frac{w+2}{2} + 1\right)\cdot\left(n^c + 2n + \frac{w+2}{2} + 1\right)\\&= n^{2c} + n^{c+1} + n^c\cdot w + \mathbf{4n^c} - 2n^2 + \frac{n\cdot w}{2} + 2n + \frac{w^2}{4} + 2w + 4\\&> n^{2c} + n^{c+1} + n^c\cdot w + \mathbf{3n^c} + \frac{n^2}{4} + \frac{n\cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4}\\&= k\end{aligned}$$

Here, we used $n^c + \frac{n}{2} + \frac{w}{2} + \frac{7}{4} \geq \frac{9n^2}{4}$ for $c \geq 3$. This shows that a bag of size $w+2$ and imbalance $3n$ would already exceed the allowed cost k. Hence, the bag of p has size at most $w+1$. ■

$G_{a,b}$ has a path-decomposition with exact imbalance values. In the previous step, we used imbalance to bound the size of certain bags. We now aim to identify a subpath consisting of such bags that contains a prescribed number of vertices. To this end, we introduce two auxiliary lemmas that describe the behaviour of imbalances in path-decompositions of $G_{a,b}$. We first show that there exists bags with imbalance at most $3n$. Next, we prove that there is a path-decomposition containing nodes with exact imbalance values. Then, the second result allows us to use particular imbalance values to define the desired subpath.

Lemma 7.16: *Let $G \in \mathcal{G}$ be a graph, let $G_{a,b}$ be the corresponding auxiliary graph constructed above and let k be as chosen above. Let (P_o, B_o) be a path-decomposition of $G_{a,b}$ with cost at most k. Then, the minimum imbalance among all nodes of P_o is strictly less than n.*

Proof. We use a similar approach as used when showing that bags of imbalance at most $3n$ have size at most $w + 1$. We we first note that the bag $B_o(p)$ of minimum imbalance has size at least its imbalance, since at this bag the path switches from being imbalanced towards one side to being imbalanced towards the other side. We thus have to show that if the bag of minimum imbalance has imbalance at least n and thus size at least n, this bag has cost more than k. But such a cost would contradict (P_o, B_o) having cost at most k. To show that a bag exceeds cost k for size at least n, it suffices to show that the cost of this bag already exceeds k for size exactly n by Observation 7.14. So assume for sake of contradiction that p has size n.

To determine the cost $(\ell(p) + b(p) + 1) \cdot (r(p) + b(p) + 1)$ of this node we need to identify $\ell(p)$ and $r(p)$. We know that $|\ell(p) - r(p)| \leq n$, $b(p) = n$ and $\ell(p) + b(p) + r(p) = 2n^c + n$. By Lemma 7.12 smaller imbalance increases the cost, so we assume imbalance n to find a lower bound. Then, we have $\ell(p) = n^c + \frac{n}{2}$ and $r(p) = n^c - \frac{n}{2}$ up to symmetry.

$$
\begin{aligned}
&\left(\left(n^c + \frac{n}{2}\right) + n + 1\right) \cdot \left(\left(n^c - \frac{n}{2}\right) + n + 1\right) \\
&= \left(n^c + \frac{3n}{2} + 1\right) \cdot \left(n^c + \frac{n}{2} + 1\right) \\
&= n^{2c} + \mathbf{2n^{c+1}} + 2n^c + \frac{3n^2}{4} + 2n + 1 \\
&> n^{2c} + \mathbf{n^{c+1}} + n^c \cdot w + 3n^c + \frac{n^2}{4} + \frac{n \cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4} \\
&= k
\end{aligned}
$$

Here, we used $n^{c+1} + \frac{n^2}{2} + \frac{n}{2} \geq n^c \cdot w + n^c + \frac{n \cdot w}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{5}{4}$ for $c \geq 1$ and $w^2 \leq \frac{n}{10}$. This shows that if the bag $B_o(p)$ of minimum imbalance has imbalance n and size n, then this bag would already exceed the allowed cost k. Hence, this bag has size and thus imbalance less than n. Note, that the bound is not necessarily tight as seen by Lemma 7.15. ∎

We use this result to find a path-decomposition with nodes having exact values of imbalance. Here, the previous lemma allows us to find a simple precondition.

Lemma 7.17: *Let $G = (V, E) \in \mathcal{G}$ be a graph and let $G_{a,b}$ be the corresponding auxiliary graph constructed above and let k be as chosen above. Let i be an integer satisfying $n \leq i \leq 2n^c + n$. If the skew-sensitive pathwidth of $G_{a,b}$ is at most k, then there exists a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ with cost at most k that contains two nodes p_ℓ and p_r with distinct bags that have imbalance exactly i.*

Proof. As the skew-sensitive pathwidth of $G_{a,b}$ is at most k, there exists a path-decomposition (P_o, B_o) of $G_{a,b}$ with cost at most k. We first show that we can modify (P_o, B_o) such that there exists one node with imbalance exactly i. Let $p, p' \in V(P_o)$ be two neighbouring nodes such that p has imbalance at most i and p' has imbalance at least i. The node p exists because $i \geq n$ and by Lemma 7.16 there must be a node of imbalance less than n. The node p' exists because $i \leq 2n^c + n$ and a leaf with an empty bag has imbalance $2n^c + n$. If either p or p' has imbalance exactly i we are done with this part. Otherwise, the imbalances of p and p' differ

by at least two. We now modify (P_o, B_o) by repeatedly inserting a node between p and p' that has imbalance closer to i. W.l.o.g., assume $\ell(p) > r(p)$. Under this assumption p' is right of p as p' has higher imbalance. To increase the imbalance from p to p', that is to increase $|\ell(p) - r(p)|$, one of the following must occur:

1. $\ell(p)$ is increased to obtain $\ell(p')$,
2. $r(p)$ is decreased to obtain $r(p')$ or
3. both changes occur simultaneously.

We insert a node p^* between p and p' that mirros the change but with exactly one unit. In case (1), some vertices are removed from the bag of p. Then, we set p^* to be identical to p with one vertex removed. In case (2) some vertices are added to the bag of p. Then, we set p^* to be identical to p with one vertex added. In case (3) both occur. Then, we set p^* to be identical to p with one vertex removed. It is straightforward to verify that the cost of p^* is bounded by the cost of p' in case (2) and by the cost of p in cases (1) and (3). Moreover, p^* has imbalance one unit closer to i than p. Repeating this process, we eventually obtain a node with imbalance exactly i.

It remains to show that there are two such pairs p, p' and thus two nodes with imbalance exactly i. Remember that i is strictly larger than the node of minimum imbalance by Lemma 7.16. To find a second node with imbalance exactly i, observe that the imbalance increases monotonously when moving away from the node of minimum imbalance i_o in either direction along P_o. Hence, by applying the above construction independently to each side of the minimum-imbalance node, we obtain two distinct nodes p_ℓ and p_r with imbalance exactly i. This concludes the proof. ∎

To conclude this subsection, note that the subpath between these two nodes of exact imbalance must contain the node of minimum imbalance. This follows from the fact that the imbalance increases monotonically when moving away from the node of minimum imbalance i_o in either direction along P_o. Moreover, this increase is strictly monotonic whenever the bags are non-redundant.

Defining the subpath R. Using the auxiliary result on imbalance, we define a subpath R of the path-decomposition. Our goal is to ensure that R contains exactly $3n$ vertices and that each bag in R has imbalance at most $3n$. To achieve this, we select two nodes whose imbalance attains a desired value exactly.

Lemma 7.18: *Let $G \in \mathcal{G}$ be a graph, let $G_{a,b}$ be the corresponding auxiliary graph constructed above and let k be as chosen above. If the skew-sensitive pathwidth of $G_{a,b}$ is at most k, then there exists a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ of cost at most k together with a subpath $R = (p_\ell, \ldots, p_r)$ of $P_{a,b}$ such that:*

1. *the bags of R together contain exactly $3n$ vertices,*
2. *there are exactly $n^c - n$ vertices that are exclusively contained in bags strictly left of R,*
3. *there are exactly $n^c - n$ vertices that are exclusively contained in bags strictly right of R, and*
4. *each bag $B_{a,b}(p)$, for $p \in R$, has size at most $w + 1$.*

Proof. As the skew-sensitive pathwidth of $G_{a,b}$ is at most k, there exists a path-decomposition (P_o, B_o) of $G_{a,b}$ with cost at most k. We now apply Lemma 7.17 with $i = 3n - (w+1)$ to obtain a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ with cost k. As $n \leq i = 3n - (w+1) \leq 2n^c + n$ the precondition of this lemma is met. So this lemma guarantees the existence of a node that has imbalance exactly $3n - (w+1)$. Let p_ℓ be the leftmost such node. By Lemma 7.15 this node has size at most $w+1$. Furthermore, we can assume that it has size exactly $w+1$ by adding the same number of vertices from the left and right to the bag. As seen before, the imbalance of p_ℓ fixes the distribution of vertices on either side of this bag. Specifically, the number of vertices exclusively in bags to the left of p_ℓ is $n^c - n$ and number of vertices exclusively in bags to the right is $n^c + 2n - (w+1)$. These values follow from the fact that $G_{a,b}$ contains $2n^c + n$ vertices and that $\ell(p_\ell) + b(p_\ell) + r(p_\ell) = |V(G_{a,b})|$. Similarly, the lemma guarantees the existence of a node p_r with a bag distinct from that of p_ℓ, lying strictly to the right of p_ℓ, and having the same imbalance $3n - (w+1)$. We define R as the subpath of $P_{a,b}$ from p_ℓ to p_r, including both endpoints.

By construction there are exactly $n^c - n$ vertices exclusively in the bags left of p_ℓ and $n^c - n$ vertices exclusively in the bags right of p_r. Thus, there are exactly $3n$ vertices that appear in bags of R. As all bags in R have imbalance less than $3n$, each bag in R has size at most $w+1$ by Lemma 7.15. ∎

The structure of path-decompositions of $G_{a,b}$. So far, we have identified a subpath R of size $3n$ such that each bag in R has size at most $w+1$ and such that there are $n^c - n$ vertices in the bags on either side of R. In addition to this result, we use the structural properties of path-decompositions of grids established in Chapter 6. In particular, we rely one the fact that, in every path-decomposition of a grid, there exists a subpath L that contains almost all nodes of the decomposition and whose bags satisfy a lower bound on their size (Theorem 6.1). In the next and final step, we show that the vertices of G occur only in bags belonging to R.

To this end, we analyse how R intersects the grids Γ_a and Γ_b in the path-decomposition $(P_{a,b}, B_{a,b})$ from Lemma 7.18. The key idea is to show that some bag of R contains $w+1$ vertices of one of the grids. By the connectedness of the remaining graph, this implies that all vertices of G and the other grid lie on the same side of this node. We therefore first show that the subpath L of $(P_{a,b}, B_{a,b})$, restricted to one of the grids, intersects R.

Lemma 7.19: *Let $G \in \mathcal{G}$ be a graph and let $G_{a,b}$ be the corresponding auxiliary graph constructed above. Let $(P_{a,b}, B_{a,b})$ be the path-decomposition of $G_{a,b}$ with the subpath R as defined in Lemma 7.18. Let $p_\ell(\Gamma_a), p_r(\Gamma_a), p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$ be as defined in Theorem 6.1. Then, the following statements are true:*

1. *exactly one of $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ lies inside R and*
2. *exactly one of $p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$ lies inside R.*

Proof. For each grid there are four possible cases where the two nodes can be located. First, both nodes can be on the same side of R. Second, both can be on opposite sides of R. Third, both can be inside R. Fourth, one can be inside R and one can be outside R. We show that the first three cases result in contradiction and that due to this case four must occur.

We start by showing that $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ cannot lies on the same side of R. Remember that Γ_a contains n^c vertices and that there exactly $n^c - n$ vertices on either side of R. Hence at least n vertices of Γ_a must lie inside R or on the opposite side of R from the remaining ones. Consider the nodes $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ obtained from Theorem 6.1 applied to the path-decomposition

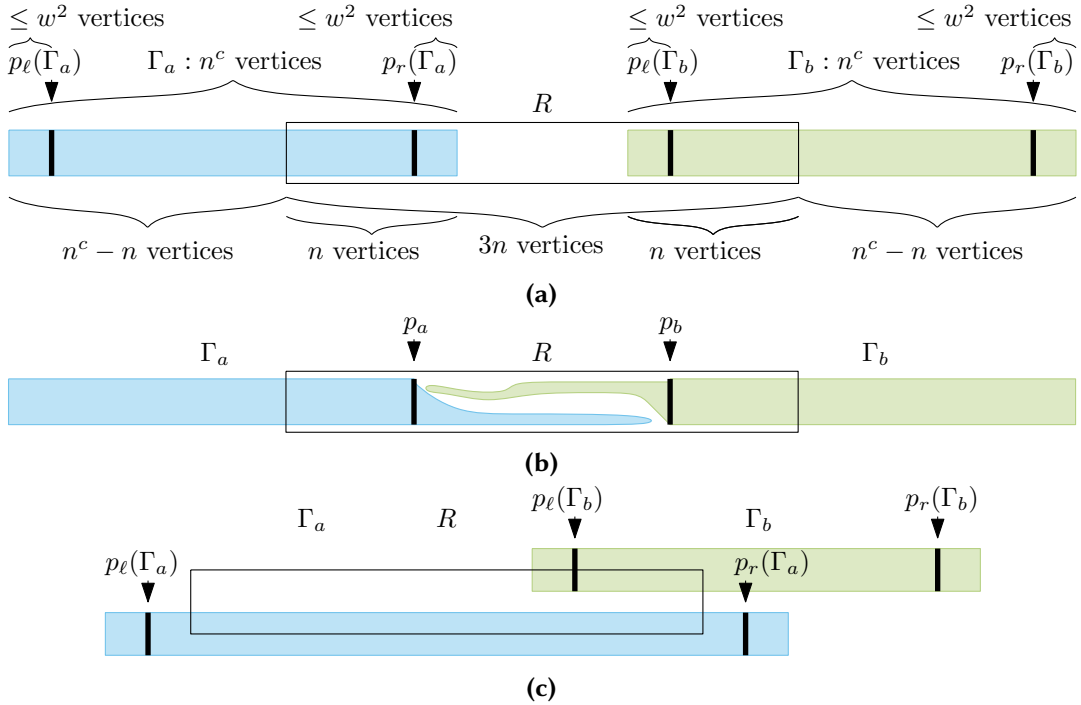


Figure 7.5: The layout of the path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$. For each grid the nodes mentioned in Theorem 6.1 are marked. The bold vertical lines mark bags that contain $w + 1$ vertices of one grid. Inside R such bags enforce that all vertices not in the grid of this bag are on the same side of the line. Part (a) shows an idealized layout and the vertex counts of the different sections. Part (b) illustrates that Γ_a (blue) and Γ_b (green) might actually extend a little bit further. Part (c) depicts a failed attempt for a different layout: A visual comparison and a closer look at the numbers reveals that this must fail, also the bold vertical line inside R is passed by with other vertices.

$(P_{a,b}, B_{a,b})$ restricted to Γ_a. By Theorem 6.1, the number of vertices between $p_\ell(\Gamma_a)$ and the corresponding end of the decomposition of Γ_a is at most $w^2 \leq \frac{n}{10}$. Here, we used requirement (3) of Lemma 7.1 and $G_{a,b} \in \mathcal{G}$ to compare $w^2 = \mathrm{pw}(G_{a,b})^2$ and n. The same holds for $p_r(\Gamma_a)$. Consequently, $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ cannot lie on the same side of R. The same argument applies to $p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$. The different size constraints are visualized in Figure 7.5.

Next, we consider the second case. So assume for sake of contradiction that $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ lie on opposite sides of R. Then, every bag in the subpath R must contain at least w vertices of Γ_a due to Theorem 6.1. However, $p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$ cannot lie on the same side of R as seen before. So either $p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$ lie on opposite sides of R or at least one of them lies inside R. In each case at least one bag of R additionally contains at least w vertices of Γ_b. Then, this bag would contain $2w$ vertices contradicting property (4) of Lemma 7.18. So $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ cannot lie on opposite sides of R. By symmetry the same holds for Γ_b.

Finally, consider the case that both $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ lie inside R. This cannot occur as R contains $3n$ vertices by property (1) of Lemma 7.18 and there are at least $n^c - 2w^2$ vertices in the bags between $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ by Theorem 6.1. By symmetry the same holds for Γ_b.

As all other cases lead to a contradiction exactly one of $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ lies inside R and exactly one of $p_\ell(\Gamma_b)$ and $p_r(\Gamma_b)$ lies inside R. ∎

Note that it does not suffice to show that R intersects the subpath L for each grid. This is due to the fact that it is possible that all bags of R are identical when restricted to the grid. In this case, Theorem 6.1 would yield only a lower bound of w rather than $w + 1$.

We obtain the following corollary by combining property (2) of Theorem 6.1, Lemma 7.18, and the previous lemma.

Corollary 7.20: *Let $G \in \mathcal{G}$ be a graph and let $G_{a,b}$ be the corresponding auxiliary graph constructed above. Let $(P_{a,b}, B_{a,b})$ be the path-decomposition of $G_{a,b}$ with the subpath R as defined in Lemma 7.18. Then, there are nodes $p_a, p_b \in R$ such that $B_{a,b}(p_a)$ contains only vertices of Γ_a and $B_{a,b}(p_b)$ contains only vertices of Γ_b.*

Proof. By Lemma 7.19, one of $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$ lies inside R. As noted earlier, at least n vertices of Γ_a must lie inside R, and thus at least one additional node from the subpath L between $p_\ell(\Gamma_a)$ and $p_r(\Gamma_a)$, that is distinct from the endpoints, also lies inside R. By property (2) of Theorem 6.1, either the endpoint of L in R or its distinct neighbour inside L contains $w + 1$ vertices of Γ_a. Moreover, by property (4) of Lemma 7.18, this bag contains no other vertices. By symmetry, the same argument applies to Γ_b. ∎

We use these two nodes to argue that all vertices of G occur only in bags of R. Since Lemma 7.18 guarantees that all bags in R contain at most $w + 1$ vertices, this provides the final ingredient needed to bound the pathwidth of G. A visualization of the structure we have enforced so far can be seen in Figure 7.5a and Figure 7.5b.

Lemma 7.21: *Let $G \in \mathcal{G}$ be a graph and let $G_{a,b}$ be the corresponding auxiliary graph constructed above. Let $(P_{a,b}, B_{a,b})$ be the path-decomposition of $G_{a,b}$ with the subpath R as defined in Lemma 7.18. Then, all vertices of G are only contained in bags of R.*

Proof. By Corollary 7.20, there exits a node p_a whose bag only contains vertices of Γ_a and a node p_b whose bag only contains vertices of Γ_b. Assume W.l.o.g. that p_a lies left of p_b. Since Γ_a is connected, all its vertices lie entirely to the left of p_b. Similarly, since Γ_b is connected, all its vertices lie entirely to the right of p_a. Now, as G is connected, all its vertices must lie either entirely to the left of p_a, entirely between p_a and p_b, or entirely to the right of p_b. If G were entirely left of p_a, then no path could connect G to Γ_b, contradicting the connectedness of $G_{a,b} - \Gamma_a$. Similarly, if G were entirely right of p_b, no path could connect G to Γ_a, contradicting the connectedness of $G_{a,b} - \Gamma_b$. Hence G must lie entirely between p_a and p_b. It follows that all vertices of G lie inside bags of R, establishing the lemma. ∎

Note that these lemmas implicitly require that the skew-sensitive pathwidth of $G_{a,b}$ is at most k (for k as chosen above), since they invoke Lemma 7.18.

A path-decomposition of G. We are ready to conclude our correctness proof.

Lemma 7.22: *Let $G = (V, E) \in \mathcal{G}$ be a graph and let $G_{a,b}$ be the auxiliary graph constructed above. Also let w be an integer. If $G_{a,b}$ has skew-sensitive pathwidth at most $k = n^{2c} + n^{c+1} + n^c \cdot w + 3n^c + \frac{n^2}{4} + \frac{n \cdot w}{2} + \frac{3n}{2} + \frac{w^2}{4} + \frac{3w}{2} + \frac{9}{4}$, then G has pathwidth at most w.*

Proof. By Lemma 7.18 and Lemma 7.21, there exists a path-decomposition $(P_{a,b}, B_{a,b})$ of $G_{a,b}$ with cost k, together with a subpath R such that each bag in the subpath R has size at most $w+1$ and all vertices of G appear only in bags of R. We now construct a path-decomposition of G by removing all vertices not in G from $(P_{a,b}, B_{a,b})$. This clearly yields a path-decomposition of G. Furthermore, the width of this decomposition is at most w. ∎

7.4.3 Concluding the reduction from PATHWIDTH to SKEW-SENSITIVE PATHWIDTH

We summarize the reduction presented in this section in the following lemma.

Lemma 7.23: *There exists a polynomial-reduction from PATHWIDTH on graphs of $\mathcal{G}$ to SKEW-SENSITIVE PATHWIDTH on graphs of $\mathcal{G}$.*

Proof. Given an instance (G, w) of PATHWIDTH on graphs of $\mathcal{G}$, we compute the vertices a and b using the algorithm of property (5) of Lemma 7.1 and construct the graph $G_{a,b}$ for G in time $\mathcal{O}(n^c)$. By Lemma 7.10, the graph $G_{a,b}$ is contained in the graph class $\mathcal{G}$. Furthermore, we choose k as described in Lemma 7.13 and fix $c = 3$. Constructing the auxiliary graph increases the vertex count by $2n^3$. We note that if G has pathwidth of at most w, then the reduction produces a graph $G_{a,b}$ of skew-sensitive pathwidth at most k by Lemma 7.13. If the constructed graph $G_{a,b}$ has skew-sensitive pathwidth at most k, then G has pathwidth at most w by Lemma 7.22. ∎

7.5 Hardness-results for CLIQUE-PARTITIONED TREEWIDTH and its consequences

In this final section, we combine the two reductions to prove the $\mathcal{NP}$–hardness of CP-TREEWIDTH and discuss the implications of this result.

We start with Corollary 7.2 which states that PATHWIDTH is hard for graphs of $\mathcal{G}$. Using the reduction of Lemma 7.23 from PATHWIDTH to SKEW-SENSITIVE PATHWIDTH we get:

Lemma 7.24: *SKEW-SENSITIVE PATHWIDTH is $\mathcal{NP}$-hard for graphs of $\mathcal{G}$.*

We then apply the reduction of Lemma 7.9 from SKEW-SENSITIVE PATHWIDTH to CP-TREEWIDTH to obtain the desired result.

Theorem 7.25: *CP-TREEWIDTH is $\mathcal{NP}$-complete.*

Proof. The $\mathcal{NP}$-hardness is given by Lemma 7.9 and Lemma 7.24.

It is easy to see that the problem is contained in $\mathcal{NP}$ as we only need to check if the given cp-tree-decomposition is valid and whether the weight of each bag is sufficiently low. ∎

We can easily adapt this proof to consider approximation algorithms for additive constants.

Theorem 7.26: *If $\mathcal{P} \neq \mathcal{NP}$, then no polynomial-time approximation algorithm $\mathcal{A}$ for CP-TREEWIDTH can guarantee $\mathcal{A}(\mathcal{I}) - \mathcal{OPT}(\mathcal{I}) \leq \gamma$ on all instances $\mathcal{I}$ for a fixed constant γ.*

Proof. Let γ be an integer. Assume that there exists a polynomial-algorithm $\mathcal{A}$ that approximates CP-TREEWIDTH within the additive constant γ. That is, given a graph G and an integer k, the algorithm $\mathcal{A}$ either returns a cp-tree-decomposition of G with cost at most $k + \gamma$, or correctly reports that the cp-treewidth of G is greater than k. We use this algorithm to determine whether a graph $G \in \mathcal{G}$ has pathwith at most w, by applying the reductions described in Lemma 7.1, Lemma 7.23 and Lemma 7.9. These reductions produce a graph G_γ and an integer k such that the following holds: (1) If $\mathcal{A}$ returns a cp-tree-decomposition of G_γ of weight at most $k + \gamma$, then the original graph G has pathwidth at most w. (2) If $\mathcal{A}$ reports that the cp-treewidth of G_γ is greater than k, then the original graph G has pathwidth greater than

w. As the resulting algorithm for PATHWIDTH clearly runs in polynomial time, the existence of $\mathcal{A}$ would contradict the $\mathcal{NP}$-hardness of PATHWIDTH [ACP87 | Bod+23]. Hence, such an algorithm $\mathcal{A}$ cannot exists, unless $\mathcal{P} = \mathcal{NP}$, and approximating CP-TREEWIDTH within any additive constant γ is $\mathcal{NP}$-hard.

It remains to show that the constructed algorithm for PATHWIDTH is correct. First, we prove that if the cp-treewidth of G_γ is greater than k, then the pathwidth of G is greater than w. Second, we show how to construct a path-decomposition of G with width at most w from a path-decomposition of G_γ with weight at most $k + \gamma$.

For the first part, we argue by contraposition. That is, we show that for any graph $G \in \mathcal{G}$ with pathwidth at most w the constructed graph G_γ admits a path-decomposition with cp-treewidth at most k. This follows directly from Lemma 7.13 and Lemma 7.8 without modification.

For the second part, we use arguments analogous to those used in the proofs of Lemma 7.22, Lemma 7.8 and Corollary 7.6. Some adjustments are required, since we now start with a cp-tree-decomposition of cost $k + \gamma$ rather than k. In particular, when comparing the target integer k with potential bag costs in Lemma 7.15 and Lemma 7.16, we compare against $k + \gamma$ instead of k. Since we may assume that n is larger than the constant γ and the two sides of the relevant inequalities differ by at least n, all required inequalities continue to hold for $k + \gamma$. Therefore, the remaining arguments apply without further modification, and we obtain a path-decomposition of G with width at most w.

This completes the correctness proof and establishes the claimed hardness. ∎

We now analyse what the implications of this result for our goal of determining the complexity of CP-TREEWIDTH. So far, we have seen that this problem can be solved in XP-time, but not in polynomial time, unless $\mathcal{P} = \mathcal{NP}$. It therefore remains to determine whether CP-TREEWIDTH can be solved in FPT-time. In the following chapter, we establish a slightly weaker result. We consider gp-treewidth, a variant of cp-treewidth that additionally requires that all local clique-partitions of the bags are consistent with a single global partition of the input graph. Using this additional property, we present an FPT-algorithm for this variant.

To argue that this algorithm is essentially tight with respect to complexity classes, we extend the hardness results of this section to this new variant. Consider the reduction from SKEW-SENSITIVE PATHWIDTH on graphs of $\mathcal{G}$ to CP-TREEWIDTH presented in Section 7.3. By Lemma 7.7, the optimal clique-partition splits each bag into subsets of V and and its copy V'. Hence, the clique-partition of every bag is simply the restriction of the global partition of G_H into V and V'. Consequently, this proof also yields a reduction from SKEW-SENSITIVE PATHWIDTH to GP-TREEWIDTH.

Corollary 7.27: *There exists a polynomial-reduction from SKEW-SENSITIVE PATHWIDTH on graphs of $\mathcal{G}$ to GP-TREEWIDTH.*

Using this reduction from SKEW-SENSITIVE PATHWIDTH to GP-TREEWIDTH instead of the reduction of Lemma 7.9, we obtain the analogous hardness results for GP-TREEWIDTH. Note that in this reduction the global clique-partition is known. Hence, the problem remains hard even when the clique-partition is given as part of the input.

Theorem 7.28: *GP-TREEWIDTH is $\mathcal{NP}$-complete.*

Theorem 7.29: *If $\mathcal{P} \neq \mathcal{NP}$, then no polynomial-time approximation algorithm $\mathcal{A}$ for GP-TREEWIDTH can guarantee $\mathcal{A}(\mathcal{I}) - \mathcal{OPT}(\mathcal{I}) \leq \gamma$ on all instances $\mathcal{I}$ for a fixed constant γ.*

8 An exact FPT-algorithm for a global variant of the parameter

In this chapter, we present an FPT-algorithm for computing the gp-treewidth of a graph $G = (V, E)$. In fact, we prove a more general statement. We identify a class of parameters for which the algorithm of Bodlaender and Kloks [BK96 | AZ21] for computing treewidth in FPT-time can be adapted in a straightforward manner. Hence, for every parameter ρ in this class, there exists an FPT-algorithm that decides whether $\rho(G) \leq k$ for a given graph G and parameter value k.

The underlying idea of the algorithm of Bodlaender and Kloks [BK96 | AZ21] is to enumerate all tree-decompositions of G. Since there exists an optimal tree-decomposition of G with $|V|$ nodes (see Lemma 3.7), a naive approach would consider all tree topologies with $|V|$ nodes together with all assignments of subsets of size at most k to each node. A first refinement of this idea is to perform this enumeration via a dynamic program on a tree-decomposition (T, B). In this approach, for each node of T, the algorithm enumerates all tree-decompositions of the subgraph processed so far. However, both the naive enumeration and this direct dynamic programming approach are not fixed-parameter tractable for the chosen parameter. To overcome this issue, the authors introduce a normalization and an equivalence relation on the constructed tree-decompositions. For the normalization, they show that for every graph there exists an optimal tree-decomposition satisfying two key properties. Hence, we may restrict the enumeration to such normalized tree-decompositions and assume that all considered tree-decompositions satisfy those two properties. While this normalization alone does not substantially reduce the number of tree-decomposition that need to be enumerated, the properties it guarantees are crucial for the equivalence relation, which in turn enables a significant reduction of this number. This equivalence relation partitions the set of all normalized partial tree-decompositions into different equivalence classes whose members share fundamental properties, such as their width. Consequently, it suffices to enumerate a single representative tree-decomposition per equivalence class. To enable the enumeration of such a representative for each equivalence class, every class is associated with a compact description, referred to as *limited information*, from which a representative can be reconstructed. Thus, it suffices to bound the number of relevant equivalence classes and their associated limited information, and to show that this limited information can be enumerated efficiently.

We generalize this approach from traditional treewidth to our class of tree-decomposition based parameters. Here, the key observation is that an analogous equivalence relation can be defined for these parameters and that the correctness arguments as well as the algorithmic framework remain valid, provided that certain requirements on the parameter are satisfied.

We proceed as follows. First, we define the class of parameters by formulating five requirements that the weight function associated with a parameter has to satisfy. Next, we describe our adaptation of the algorithm of Bodlaender and Kloks [BK96 | AZ21] to parameters in this class. Using this generalized result, we then return to gp-treewidth. We first show that, when a clique-partition of the input graph is provided as a part of the input and thus fixed in advance, the parameter gp-treewidth belongs to our class. Consequently, it can be computed

in FPT-time. Finally, we demonstrate that, by a small modification of the algorithm, the clique-partition of the input graph can be computed simultaneously with a tree-decomposition that minimizes the gp-treewidth.

8.1 A class of efficiently computable parameters

We are now ready to formally define a class of tree-decomposition-based parameters that can be computed efficiently. We first explain how such parameters are related to weight functions and then restrict the admissible weight functions using five requirements to form this class of parameters. These requirements ensure both the correctness and the claimed running time bounds of the underlying classical approach.

We begin by formalizing the framework of the tree-decomposition based parameters we are studying and how those parameters can be induced by a weight function. This can be regarded as a generalization of the way in which the parameters treewidth, cp-treewidth, and tree-independence number, introduced so far, are defined by assigning a weight to each bag. Let w be a weight function that assigns a non-negative integer weight to every subgraph of the input graph. For a vertex set $X \subseteq V$, we use the shorthand notation $w(X)$ instead of $w(G[X])$ to denote the *weight* of the induced subgraph $G[X]$. Then, the *weight* of a tree-decomposition (T, B) of a graph G is the maximum weight of any bag in (T, B). Finally, the *w-treewidth* $\mathrm{tw}_w(G)$ of a graph G is the minimum weight of any tree-decomposition of G. Note, that by defining the weight in this way, the weight of a bag depends solely on the subgraph induced by the bag and not on any other bags of the decomposition.

To define our class we introduce five requirements on the weight function inducing a parameter. These requirements can be grouped into three types. The first two requirements ensure that the weight behaves monotonously under the two fundamental operations of adding and removing a vertex. The following two requirements specify which information may be used when computing the weight of a set after minor modifications similar to introduce- and join-nodes in dynamic programs on tree-decompositions. The final requirement establishes a relationship between the weight of a bag and its size and thus forms the foundation of our running time analysis.

We define the class of *efficiently computable parameters* $\mathcal{ECP}$ as the class containing all tree-decomposition-based parameters w-treewidth induced by a weight function w that satisfies the following requirements (R1) to (R5)[1].

(R1) *Subset monotonicity*: Let X be a vertex set and let $S \subseteq X$ be a subset of X. Then, $w(S) \leq w(X)$.

(R2) *Superset monotonicity*: Let X, Y be two vertex sets and let $v \notin X \cup Y$ be a vertex such that $N(v) \cap X = N(v) \cap Y$. If $w(X) \leq w(Y)$, then $w(X + v) \leq w(Y + v)$.

(R3) *Vertex addition computability*: Let X be a vertex set and let $v \notin X$ be a vertex. Then, $w(X + v) = w(X) + g(N(v) \cap X)$ where $g(x)$ can be computed in time $2^{\mathcal{O}(|N(v) \cap X|)}$.

(R4) *Union computability*: Let X, Y be two vertex sets such that no vertex of $X - Y$ is adjacent to a vertex of $Y - X$. Then, $w(X \cup Y) = w(X) + w(Y) - h(X \cap Y)$ where $h(x)$ can be computed in time $2^{\mathcal{O}(|X \cap Y|)}$.

[1]For our applications, linear running time bounds are sufficient. We nevertheless state a more general version that permits larger running times, as it captures a broader class of parameters without affecting the overall running time.

(R5) *Boundedness*: There exists a computable function f such that $|X| \leq f(w(X))$ for all vertex sets X.

It is easy to see that any weight function satisfying vertex addition computability (R3) also satisfies superset monotonicity (R2). However, since (R2) is used explicitly in later arguments, we state it as a separate requirement.

Lemma 8.1: *Let w be a weight function satisfying (R3). Then, w also satisfies (R2).*

Proof. Let X, Y be two vertex sets and let $v \notin X \cup Y$ be a vertex such that $N(v) \cap X = N(v) \cap Y$. Also, let $w(X) \leq w(Y)$. By (R3), we know that the weight difference $w(X + v) - w(X)$ can be described as $g(N(v) \cap X)$. Similarly, by (R3), the weight difference $w(Y + v) - w(Y)$ can be described as $g(N(v) \cap Y)$. Since the two neighbourhoods $N(v) \cap X$ and $N(v) \cap Y$ are the same by assumption, these two weight differences are equal. Hence,

$$\begin{aligned} w(X + v) &= w(X) + g(N(v) \cap X) \\ &= w(X) + g(N(v) \cap Y) \\ &\leq w(Y) + g(N(v) \cap Y) \\ &= w(Y + v), \end{aligned}$$

where we used $w(X) \leq w(Y)$, ∎

In the following, we consider parameters belonging to this class and rely on the requirements introduced above in our arguments. For better intuition, the reader may keep the parameter treewidth, together with the weight function $w_{\text{tw}}(G[X]) = |X| - 1$, in mind as those are the simplest example of this class. It is straightforward to verify that this weight function satisfies all of the stated requirements, as witnessed by the following observation.

Observation 8.2: *Let $G = (V, E)$ be a graph. Then, the weight function $w_{\text{tw}}(G[X]) = |X| - 1$ satisfies the requirements (R1) to (R5).*

Proof. We verify the five requirements one by one.

(R1) *Subset monotonicity*: The size of a set cannot increase when elements are removed from it.

(R2) *Superset monotonicity*: Inserting one element into a set increases its size by exactly one. Hence, $w_{\text{tw}}(X+v) = w_{\text{tw}}(X)+1$, $w_{\text{tw}}(Y+v) = w_{\text{tw}}(Y)+1$, and $w_{\text{tw}}(X+v) \leq w_{\text{tw}}(X+v)$.

(R3) *Vertex addition computability*: As observed above, the weight change is exactly one and thus $g(x) = 1$ satisfies all conditions.

(R4) *Union computability*: Uniting two sets counts duplicate elements twice. Thus, we choose $h(X \cap Y) = |X \cap Y|$, which satisfies all conditions.

(R5) *Boundedness*: We choose $f(x) = x + 1$.

Hence, all requirements (R1) to (R5) are satisfied. ∎

In the next subsection, we prove that all parameters of the class $\mathcal{ECP}$ can be computed in FPT-time. Formally, we show the following result, where f is the function introduced in (R5).

Theorem 8.3: *Let $G = (V, E)$ be a graph and let k be an integer. Let (T, B) be a given tree-decomposition of G with weight $\ell \in \mathcal{O}(k)$. Consider a parameter of $\mathcal{ECP}$ with weight function w. Then, there exists an algorithm that decides whether G has w-treewidth at most k in time $2^{\mathcal{O}(f(\ell)^3 + k \cdot f(\ell)^2)} \cdot |V|$.*

8.2 An FPT-algorithm for computing the parameters of $\mathcal{ECP}$

We now show that every parameter in the class $\mathcal{ECP}$ can be computed in FPT-time. To this end, we summarize and adapt the simplified description by Althaus and Ziegler [AZ21] of the FPT-algorithm for computing treewidth. This result was originally introduced by Bodlaender and Kloks [BK96]. We begin by introducing the notation used throughout this subsection and then describe the structure of this subsection.

Let w be a weight function such that its induced parameter w-treewidth belongs to the class $\mathcal{ECP}$. Let $G = (V, E)$ be a connected graph and let k be an integer. Moreover, let (T, B) be a nice tree-decomposition of G with weight $\mathcal{O}(k)$. To distinguish this decomposition from those constructed by our algorithm, we refer to it and its bags using the prefix *host*. We fix this notation for the remainder of this section.

In our adaptation of the algorithm to the parameters of $\mathcal{ECP}$ we follow the outline of the traditional approach presented above. In the first subsection, we define a normalization that allows us to assume that all considered tree-decompositions satisfy two key properties. We then explain how the equivalence relation is used, formulate the requirements for correctness, and provide intuition for the notion of equivalence. In this part we also detail what exactly the algorithm computes, that is we formalize the structure of partial solutions used in this approach. We then present the equivalence relation together with its correctness arguments. Here, we detail the aforementioned limited information associated with each equivalence class. The final two subsections are devoted to actually computing the limited information. First, we formally show a bound for the number of partial solutions that the dynamic program needs to consider at each node. Then, we show how these partial solutions can be computed within the desired running time.

8.2.1 A normalization and two key properties

We begin improving the naive enumeration by normalizing the tree-decompositions we construct. More precisely, we show that if a tree-decomposition of a given weight exists, then there also exists a tree-decomposition of the same weight that satisfies two additional structural properties. Consequently, it suffices to restrict our attention when enumerating to tree-decompositions satisfying these properties.

We begin with the first property which states that the bags of some leaves remain unchanged after processing a given host-node. Recall, that G_t refers to the subgraph of G induced by the vertices processed by the host-decomposition before the host-node t. In our proof we use the following operation, called the *addition of a leaf*. Let t_c be a node with bag X. The addition of a leaf to t_c is the operation that adds a new node t'_c that is adjacent to t_c and assigns it a bag $Y \subseteq X$. Clearly, this operation does not increase the weight of the decomposition due to subset monotonicity (R1).

Lemma 8.4 (Lemma 2 of Althaus and Ziegler [AZ21])**:** *Let G be the input graph and let (T, B) be the host-decomposition. If the w-treewidth of G is at most k, then there exists a tree-decomposition $(\tilde{T}, \tilde{B})$ of G with weight at most k such that for every host-node t the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t satisfies the following property.*
Let $\ell \in V(T_t)$ be a leaf with adjacent node $a \in V(T_t)$ for which the following preconditions hold:

1. *$B_t(\ell)$ contains at least one vertex forgotten by the host-decomposition before reaching t and*
2. *the intersection $B(t) \cap B_t(\ell)$ is a subset of the intersection $B(t) \cap B_t(a)$.*

(a) (b)

Figure 8.1: This figure illustrates an example of the proof of Lemma 8.4. In part (a), the state before the modification is shown. Here, the intersection with $B(t)$ (black vertices) of the right leaf is a subset of the intersection for its parent and this leaf contains already forgotten vertices (purple). Hence, we can modify the decomposition by assigning the vertices of Z, which are in $(\tilde{T}, \tilde{B})$ but not (T_t, B_t), to a newly created leaf (see part (b)).

Then, no vertex from $V - V(G_t)$ is contained in $\tilde{B}(\ell)$.

Proof. Let $(\tilde{T}, \tilde{B})$ a tree-decomposition of G of weight at most k that has the minimum number of violations of this property. We prove that $(\tilde{T}, \tilde{B})$ indeed has no violations using proof by contradiction. This proof is visualized in Figure 8.1.

So assume for sake of contradiction that there exists a host-node t and a leaf ℓ in the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t such that t and ℓ violate the property. That is, the set $Z = \tilde{B}(\ell) - V(G_t)$ of vertices from $V - V(G_t)$ contained in the bag of ℓ is non-empty. We modify $(\tilde{T}, \tilde{B})$ without increasing the weight to ensure that t and ℓ no longer violate the property. The intuition for this modification is to split a leaf that contains vertices forgotten before t, vertices in $B(t)$, and vertices introduced after t into two leaves. Here, the first leaf contains the forgotten vertices together with the vertices of $B(t)$ and the other contains the vertices not yet introduced together with the vertices of $B(t)$. Formally, we add a new leaf ℓ' adjacent to a and assign it the bag $(\tilde{B}(\ell) \cap B(t)) \cup Z$. Finally, remove the vertices in Z from the bag $\tilde{B}(\ell)$ to obtain the new bag $\tilde{B}(\ell) - Z$ of ℓ.

All vertices and edges of the graph remain covered, since the vertices forgotten by the host-decomposition before t and those introduced after t are not connected. Hence, $(\tilde{T}, \tilde{B})$ remains a valid tree-decomposition. Moreover, after the modification, both the bag of ℓ and the bag of the newly added leaf are subsets of the original bag of ℓ. Hence, the weight does not increase by monotonicity under taking subsets (R1). Thus, this modification yields a tree-decomposition of G with at most the same weight and fewer violations. This is a contradiction to minimality and thus the lemma holds. ∎

Observe, that this result extends naturally from leaves to certain subtrees. To formalize this, we define an *attached subtree* as a connected subtree in which exactly one node, called the *root*, has neighbours outside the subtree, and this root has exactly one such neighbour. For a host-node t, we call an attached subtree of a tree-decomposition $(\tilde{T}, \tilde{B})$ *t-nice*, if every node in the subtree satisfies the conditions on the leaf ℓ in the above lemma. In particular, consider the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t. Then, for every node ℓ in this subtree, the following conditions have to hold:

1. the bag $B_t(\ell)$ contains at least one vertex forgotten by the host-decomposition before reaching t and
2. the intersection $B(t) \cap B_t(\ell)$ is a subset of the intersection $B(t) \cap B_t(a)$, where a is the parent of ℓ.

Note, that for the root of such a subtree, its parent lies outside of the attached subtree but is still considered in the above condition. Using this definition, we generalize the above lemma from considering a single leaf ℓ to considering an entire nice attached subtree.

Corollary 8.5: *Let G be the input graph and let (T, B) be the host-decomposition. If the w-treewidth of G is at most k, then there exists a tree-decomposition $(\tilde{T}, \tilde{B})$ of G of weight at most k such that for every host-node t the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t satisfies the following property.*
Let S be a t-nice attached subtree. Then, for every $\ell \in V(S)$, no vertex from $V - V(G_t)$ is contained in $\tilde{B}(\ell)$.

Proof. We proceed analogously to the proof of Lemma 8.4 but instead of adding a single leaf, we add a copy of the subtree S. For each bag, we define its contents as above: We retain the vertices of G_t in the original structure and add the vertices of $V - V(G_t)$, together with the vertices of $B(t)$, to the copied structure.

It remains to verify that the resulting structure is still a valid tree-decomposition. In particular, we must check the connectivity condition. It suffices to consider the vertices of $B(t)$, since all other vertices occur only in one of the two copies. For vertices in $B(t)$, observe that by condition (2) in the definition of nice attached trees, any such vertex contained in a bag is also contained in all bags along the path to the root, as well as in the bag of the parent of the root. Hence, the connectivity condition is satisfied. Therefore, the resulting structure is a valid tree-decomposition, and the lemma follows. ∎

We now extend this normalization to a second property, which states that along certain paths, all nodes are modified in the same way after processing a given host-node. Here, we use the following definitions. In graph theory a *caterpillar* is a tree that contains a path such that all vertices have distance at most one to this path. We extend this notion to tree-decompositions. For a tree-decomposition $(\tilde{T}, \tilde{B})$, a path P is called a *caterpillar-path*, if all nodes of $\tilde{T} - P$ that are incident to non-endpoints of P are roots of attached subtrees. Those attached subtrees are called *caterpillar-legs*. For a host-node t, a caterpillar-path $P = (t_i, \ldots, t_j)$ is called *t-nice*, if $B(t) \cap \tilde{B}(t_i) = \cdots = B(t) \cap \tilde{B}(t_j)$ and if its caterpillar-legs are t-nice. The reader may think of the caterpillar-legs as leaves, since we only consider the content of the root bag and the distinction thus lies solely in whether we apply Lemma 8.4 or Corollary 8.5.

In proving this property, we use the following operation, called *duplication of a node*. Let $t_c t_c'$ be an edge of a tree-decomposition. The duplication of the node t_c is the operation that subdivides this edge $t_c t_c'$ by adding a node and assigning it the same bag as t_c. This operation does not increase the weight of the decomposition, due to subset monotonicity (R1).

Lemma 8.6 (Lemma 3 of Althaus and Ziegler [AZ21])**:** *Let G be the input graph and let (T, B) be the host-decomposition. If the w-treewidth of G is at most k, then there exists a tree-decomposition $(\tilde{T}, \tilde{B})$ of weight at most k of G such that for every host-node t the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t satisfies the following property.*
Let $P = (t_i, \ldots, t_j)$ be a t-nice caterpillar-path in (T_t, B_t) where $w(B_t(t_i)) \neq w(B_t(t_{i+1}))$ is the minimum and $w(B_t(t_j))$ is the maximum weight of bags of this path (or vice versa). Then, the following properties hold.

1. *For every vertex $v \in V - V(G_t)$, v is contained in one of the bags along P if and only if this vertex is contained in all of the bags along P and*
2. *for every node ℓ in some caterpillar-leg of P, no vertex from $V - V(G_t)$ is contained in $\tilde{B}(\ell)$.*

Proof. Let $(\tilde{T}, \tilde{B})$ a tree-decomposition of G of weight at most k that has the minimum number of violations of this property. We prove that $(\tilde{T}, \tilde{B})$ indeed has no violations using proof by contradiction. We visualize this proof in Figure 8.2.

So assume for sake of contradiction that there exists a host-node t and a path $P = (t_i, \ldots, t_j)$ in the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t such that t and P violate the property. We modify $(\tilde{T}, \tilde{B})$ without increasing the weight to ensure that t and P no longer violate the property. Let $(a_i, \ldots, a_j)$ be the weights of the nodes $(t_i, \ldots, t_j)$ in (T_t, B_t). We consider the case where a_i is the minimum weight and a_j is the maximum weight of this path. The other case follows symmetrically. Similar to before the approach is to move the vertices introduced after t to newly added copies. First, we duplicate the node t_i with the minimum weight exactly $j - i$ times. Hence, instead of one node with weight a_i the decomposition contains $j - i + 1$ nodes $t_i^1, \ldots, t_i^{j-i+1}$, all having weight a_i in (T_t, B_t). Let $Z_i, \ldots, Z_j$ be the vertices of $V - V(G_t)$ in the bags $\tilde{B}(t_i), \ldots, \tilde{B}(t_j)$, respectively. We now move these sets to the nodes $t_i^1, \ldots, t_i^{j-i+1}$. That is, each bag of a node t_i^ℓ, for $\ell \in \{1, \ldots, j - i + 1\}$, now has content $B_t(t_i) \cup Z_\ell$. For the original nodes $t_{i+1}, \ldots, t_j$, we use the set Z_j instead of $Z_{i+1}, \ldots, Z_j$. That is, each bag of a node t_ℓ, for $\ell \in \{i + 1, \ldots, j\}$, now has content $B_t(t_\ell) \cup Z_j$. Hence, condition (1) is satisfied. All other bags remain unchanged.

Next, we ensure that that this modification preserves the validity of the tree-decomposition. Consider the caterpillar-legs before our modification. Let C_r be a root-bag of some caterpillar-leg, and let I be the incident bag on the path. By assumption $C_r \cap B(t)$ is a subset of $I \cap B(t)$. Hence, two cases can occur for C_r. First, if C_r contains a vertex forgotten by a host-node before t, by Corollary 8.5, C_r only contains vertices of G_t. Second, if C_r contains no vertices forgotten by a host-node before t, then C_r is a subset of $B(t) \cup (V - V(G_t))$. Hence, all caterpillar-legs whose root-bags are subsets of $B(t) \cup (V - V(G_t))$ must be moved together with the reassigned vertex sets $Z_i, \ldots, Z_j$. Caterpillar-legs whose root-bags are subsets of $V(G_t)$ need not be moved, since their vertices are adjacent only to the vertices of G_t, which remain unchanged. Thus, this modification yields a valid tree-decomposition and meets (2).

Finally, we must verify that the weight of the decomposition does not increase. Each added node t_i^ℓ gets the bag $B_t(t_i) \cup Z_\ell$. We compare its weight after the modification to the weight of t_ℓ before the modification using superset monotonicity (R2). This node t_ℓ had the bag $B_t(t_\ell) \cup Z_\ell$ before the modification. Choose X to be $B_t(t_i)$ and choose Y to be $B_t(t_\ell)$. These are exactly the bags of the nodes without Z_ℓ. Since X has weight a_i and P is chosen such that a_i is the minimum value of the weights of the nodes in P, we have that $w(X) \leq w(Y)$. Choose v to be some vertex of Z_ℓ. Then, the condition on the neighbourhoods is satisfied, as X and Y have the same intersection with $B(t)$ and since the only vertices of $X, Y \subseteq V(G_t)$ to which v may be adjacent are those of $B(t)$. Then (R2) guarantees $w(X + v) \leq w(Y + v)$. We can apply this iteratively to obtain that the bag of t_i^ℓ after the modification has weight at most the weight of the bag of t_ℓ before the modification.
Each original node t_ℓ is assigned the bag $B_t(t_\ell) \cup Z_j$ during the modification. We compare its weight in after the modification to the weight of t_j before the modification using superset monotonicity (R2). This node t_j had the bag $B_t(t_j) \cup Z_j$ before the modification. Choose X to be $B_t(t_\ell)$ and choose Y to be $B_t(t_j)$. These are exactly the bags of the nodes without Z_j. Since Y has weight a_j and P is chosen such that a_j is the maximum value of the weights of the nodes in P, we have that $w(X) \leq w(Y)$. As before, we can apply (R2) iteratively to add the vertices of Z_j and thus obtain that the bag of t_ℓ after the modification has weight at most that of the bag of t_j before the modification. Consequently, for each bag after the modification, we can identify a bag before the modification whose weight is at least as large.

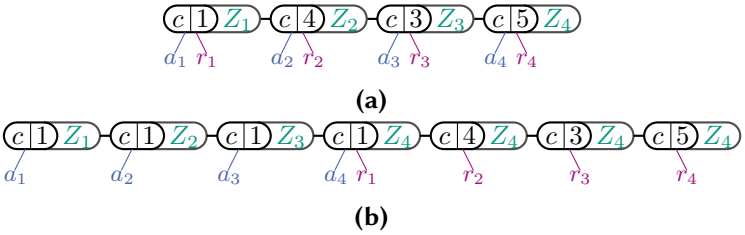


Figure 8.2: An example of the proof of Lemma 8.6. In part (a), the state before the modification is shown. Here, all vertices of the path have the same intersection with $B(t)$ (letters) but different weights (numbers). The operation moves all vertices which are in $(\tilde{T}, \tilde{B})$ but not (T_t, B_t) (the sets Z_i), as well as all caterpillar-legs whose roots contain no forgotten vertices (blue) to copies of the node with the smallest weight. Caterpillar-legs whose roots contain no vertices of $V - V(G_t)$ (purple) remain attached to their original nodes. The original nodes are all assigned the same set Z_j. The resulting configuration is depicted in part (b). For simplicity, all caterpillar-legs are depicted as leaves.

Hence, the overall weight of the tree-decomposition does not increase. Thus, this modification yields a tree-decomposition of G with at most the same weight and fewer violations. This is a contradiction to minimality and thus the lemma holds. ∎

We now combine these two results.

Lemma 8.7: *Let G be the input graph and let (T, B) be the host-decomposition. If the w-treewidth of G is at most k, then there exists a tree-decomposition $(\tilde{T}, \tilde{B})$ of weight at most k of G such that for every host-node t the restriction (T_t, B_t) of $(\tilde{T}, \tilde{B})$ to G_t satisfies the following properties.*

1. *Let S be a t-nice attached subtree. Then, for every $\ell \in V(S)$, no vertex from $V - V(G_t)$ is contained in $\tilde{B}(\ell)$.*
2. *Let $P = (t_i, \ldots, t_j)$ be a t-nice caterpillar-path in (T_t, B_t) where $w(B_t(t_i)) \neq w(B_t(t_{i+1}))$ is the minimum and $w(B_t(t_j))$ is the maximum weight of bags of this path (or vice versa). Then, the following properties hold.*
   - I *For every vertex $v \in V - V(G_t)$, v is contained in one of the bags along P if and only if this vertex is contained in all of the bags along P and*
   - II *for every node ℓ in some caterpillar-leg of P, no vertex from $V - V(G_t)$ is contained in $\tilde{B}(\ell)$.*
3. *$(\tilde{T}, \tilde{B})$ has at most $|V|$ nodes.*

Proof. Note that Corollary 8.5 and Lemma 8.6 describe the same property for different parts of a decomposition and thus can be combined. Also note that $(\tilde{T}, \tilde{B})$ can be made non-redundant as described in Lemma 3.7 without adding any violations. ∎

We call such a tree-decomposition $(\tilde{T}, \tilde{B})$ *normalized*. In the following, we restrict our enumeration to normalized tree-decompositions and exploit these two structural properties (1) and (2) in the definition of the equivalence relation.

8.2.2 Enumerating tree-decompositions with a detour

As a first refinement of the naive enumeration approach, we use a dynamic programming algorithm on the given non-optimal host-tree-decomposition (T, B) to enumerate all normalized tree-decompositions of G with weight at most k. The key idea of this algorithm is to enhance the naive enumeration of all normalized tree-decompositions by enumerating them in a structured manner based on how the constructed tree-decompositions interact with a subtree of the host-decomposition. We describe this algorithm by first specifying the partial solutions it maintains. Afterwards, we present the recursive rules used to construct these partial solutions. In the following, we will refer to this algorithm as the *inefficient-DP* and use the *constructibility*-relation it defines between partial tree-decompositions. Formally, a partial tree-decomposition of G_t, for some host-node t, is said to be *constructible* from partial tree-decompositions of $\{G_c\}_{c\in\text{children}(t)}$, if it can be obtained by applying one of the recursive rules defined by the inefficient-DP.

For each node $t \in V(T)$, a partial solution is a normalized tree-decomposition on $|V|$ nodes with weight at most k of the subgraph G_t visited so far. Using the recursive rules described below, we construct, for every node $t \in V(T)$, the set of all normalized tree-decompositions of G_t with weight at most k. Any non-normalized tree-decomposition encountered, is discarded. Similarly, any tree-decomposition with weight more than k can be discarded safely as the weight is monotone under taking subsets by (R1). Consequently, at the root of (T, B), the dynamic program enumerates all normalized tree-decompositions with $|V|$ nodes of G with weight at most k.

For a **leaf-node**, we store all tree-decompositions consisting of an arbitrary tree on at most $|V|$ nodes with empty bags.

For a **forget-node** t with child t', we have $G_t = G_{t'}$. Thus, the DP keeps all tree-decompositions constructed for t' unchanged.

For an **introduce-node** t with child t' and introduced vertex v, for every tree-decomposition of G_t there exists a tree-decomposition of $G_{t'}$ that can be extended to this decomposition by adding v to a subtree. Thus, for each enumerated tree-decomposition of $G_{t'}$, the DP adds v to all valid subtrees of the decomposition to obtain the different tree-decompositions of G_t. A subtree is considered valid, if every neighbour of v in G_t appears together with v in at least one bag.

For a **join-node** t with children t_1 and t_2, for every tree-decomposition of G_t the restriction to G_1 and G_2, respectively, is a valid tree-decomposition for that subgraph. Thus, the DP considers pairs of tree-decompositions of G_{t_1} and G_{t_2} that share the same tree topology and contain the same vertices of $B(t)$ in each pair of corresponding bags. From such a pair, it constructs a tree-decomposition of G_t by reusing the common tree topology and taking, for each node, the union of the corresponding bags of the two child-decompositions.

It can be seen that the running time of this approach is bounded by $2^{n^{\mathcal{O}(1)}}$ and is therefore not in FPT. The correctness of the approach follows from the fact that all possible differences between partial decompositions for parent- and child-nodes are considered, and thus every feasible construction is accounted for.

8.2.3 An introduction to the use of equivalences

Before formally introducing our equivalence relation, we explain how it is used to refine the inefficient-DP into an FPT-algorithm. Recall, that this DP computes, for each node t of the host-tree-decomposition, the set of all normalized partial tree-decompositions of G_t. Consequently,

a tree-decomposition of the full graph can be obtained by checking whether a suitable partial decomposition is computed for the root. In the following sections, we define an equivalence relation that groups the normalized partial tree-decompositions enumerated at each host-node into equivalence classes. The key property is that all partial decompositions within a class have the same weight. Hence, it suffices to consider each class as a unit and consider one representative from each class. However, we do not compute these equivalence classes explicitly, or alternatively a representative for each class. Instead, each class is represented by some *limited information*, that captures the shared core of equivalent tree-decompositions. The size of such a state of limited information is bounded by a function of the parameter. This allows us to enumerate all possible states of this limited information at each node of the host-tree-decomposition efficiently. However, this full enumeration may also include states that do not correspond to valid partial decomposition produced by the inefficient-DP. To address this, we introduce the notion of *compatibility*, which is visualized in Figure 8.3a. Intuitively, this extends the notion of constructibility by the inefficient-DP to limited information. Formally, a state of limited information for a node t is said to be *compatible* with a state for its child node t' if there exist partial tree-decompositions of G_t and $G_{t'}$ corresponding to these states such that the former is constructed from the latter by the inefficient-DP. A state of limited information is then considered *valid* if it is compatible with some valid state of limited information for its child. These notions extend naturally to host-nodes with two children by considering pairs of decompositions. Our algorithm then iteratively computes, for each host-node, the set of all valid states of limited information. The algorithm outputs that the graph has w-treewidth at most k if and only if a valid state is computed at the root. Thus, the algorithm effectively simulates the inefficient-DP on the level of limited information without explicitly constructing the underlying tree-decompositions.

To reconstruct a full tree-decomposition from the computed states of limited information, and thus to ensure correctness, we require the following property. If the algorithm finds a valid state at the root, then there must exist a complete underlying tree-decomposition whose restrictions to the respective subtrees match the equivalence classes associated with the computed limited information. In other words, some partial decompositions contained in the equivalence class associated with these states of limited information must be consistent with respect to constructibility by the inefficient-DP along the tree. To guarantee this consistency, we show that whenever a decomposition of G_t can be constructed by the inefficient-DP from some decomposition of $G_{t'}$, then for every equivalent decomposition of $G_{t'}$, there exists an equivalent decomposition of G_t that can be constructed from it by the inefficient-DP. This property allows us to consistently select a representative tree-decomposition for each state of limited information found above by proceeding in a bottom-up manner. We illustrate this idea in Figure 8.3. This consistency requirement is central to our correctness arguments. The full correctness proof is given in Section 8.2.4, once the concrete equivalence relation has been introduced.

For readability, we present our equivalence relation in three stages such that each stage refines the previous ones. Each stage defines a valid equivalence relation in the sense described above and further reduces the number of possible states of limited information. We conclude this part with a brief overview over these three stages. In all cases, the limited information consists of a variant of a tree-decomposition. To ensure fixed-parameter tractability, these equivalence relations must bound the number of nodes in each tree-decomposition, and thus the number of tree-topologies, as well as the number of possible assignments of vertices to bags by a function of k. The first equivalence relation reduces the number of assignments by restricting the content of each bag to the vertices of the host-bag. The second equivalence

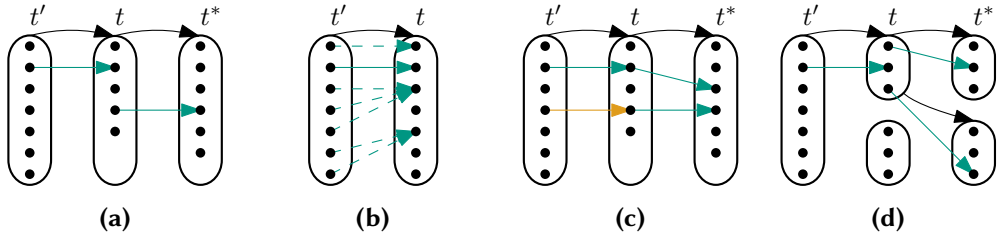


Figure 8.3: The figure illustrates a host-node t^* together with its child t and its grandchild t'. For each host-node equivalence classes (depicted as ovals) and the partial tree-decompositions belonging to these classes (depicted as dots) are shown.
Part (a) visualizes the definition of compatibility. The green arrows indicate that a partial decomposition at one node can be constructed from a partial decomposition at another node by the inefficient-DP. The black arrows indicate compatibility between the equivalence classes and follow directly from the green arrows. However, this definition does not guarantee the existence of a path from a partial decomposition at t' to a partial decomposition at t^*.
Part (b) illustrates the claimed property that constructibility by the inefficient-DP extends to equivalent decompositions. More precisely, if one partial decomposition at t' has an outgoing green arrow to a partial decomposition at t, then all equivalent partial decompositions at t' have arrows to some equivalent partial decomposition at t. The additional arrows implied by this property are shown as dashed.
Part (c) considers the implications of the above property. We can find a path (at the top) of green arrows (that is, constructibility steps of the inefficient-DP) from t' via t to t^* by starting at an arrow from t' to t and then invoking the above property for arrows from t to t^*. The partial decompositions at the endpoint of these arrows can therefore serve as valid representatives for this equivalence class at each node. The other way for constructing this path by starting at t^* and working backwards to t and t' fails, since the orange arrow does not necessarily exist. This is due to the fact that the statement only guarantees that all partial decompositions at t' have outgoing arrows, but not that all partial decompositions at t have incoming arrows.
Part (d) shows an example with multiple equivalence classes at the nodes t and t'. Since the top equivalence class for t has partial decompositions with arrows to decompositions in the top class for t^* and has partial decompositions with arrows to decompositions in the bottom class for t^*, it is compatible with both. Note, that the above property thus implies two outgoing arrows (one to a decomposition of the top class and one for the bottom class) for each decomposition of this class of t. The bottom class is not compatible with any of the shown classes of t' and t^*.

relation extends the first and bounds the number of leaves (and thus also nodes of degree 3) by removing leaves to which, due to Lemma 8.4, no vertices are added after the current host-node. The third equivalence relations extends the first two and bounds the number of nodes of degree 2 by representing paths as integer sequences and removing nodes which, by Lemma 8.6, receive same vertices as their neighbouring nodes after the current host-node.

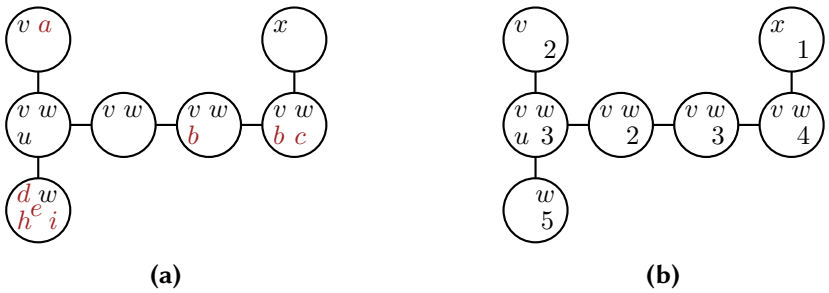


Figure 8.4: The figure illustrates the initial stage of limited information. Part (a) depicts a partial tree-decomposition. The nodes of the host-bag are shown in black, the forgotten nodes are shown in red. Part (b) illustrates the restricted bag representation of this decomposition, here the numbers illustrate the weights obtained from the weight function based on bag-sizes.

8.2.4 The initial equivalence relation: Restricted bag representations

We introduce the first equivalence relation on partial tree-decompositions with the aim of bounding the number of vertex assignments to each individual bag. This is achieved by restricting the set of vertices that may appear in a bag and only selecting vertices from the current host-bag. We begin by defining the limited information characterizing this equivalence and then consider the correctness arguments.

Let $t \in V(T)$ be a host-node and let X be a constructed tree-decomposition of G_t. Then, the *restricted bag representation* of X with respect to t is constructed from X as follows: For each bag of X, we restrict its content to the vertices contained in $B(t)$ and additionally store the weight of each original bag. We refer to a single bag in such a representation as a *restricted bag*. The *size* of a restricted bag representation is defined as the product of the number of nodes and edges in the tree of the decomposition, the number of vertices contained in each restricted bag, and the number of possible weights. This representation constitutes the limited information for each node in this step. Two tree-decompositions of G_t are called *bag-equivalent* if their restricted bag representations with respect to t coincide. Since equality on restricted bag representations is an equivalence relation, it follows that bag-equivalence is itself an equivalence relation. We visualize restricted bag representations in Figure 8.4.

We now verify that this equivalence relation satisfies the necessary correctness arguments. In particular, we show that enumerating all restricted bag representations for each host-node that are compatible with those enumerated for its children yields a valid restricted bag representation at the root if and only if the w-treewidth is at most k. Recall, that a restricted bag representation for a node t is said to be compatible with a restricted bag representation for its child node t' if there exist partial tree-decompositions of G_t and $G_{t'}$ corresponding to these restricted bag representations such that the former is constructed from the latter by the inefficient-DP. We proceed as follows. For each direction, we first provide a brief overview on the correctness argument and identify the properties that bag-equivalence must satisfy. We then show that these properties indeed hold. Finally, we argue correctness using these properties.

We begin with the forward direction. Suppose that there exists a tree-decomposition of G with weight at most k. Then, we require that, for each host-node t, the restriction of this decomposition to G_t belongs to some equivalence class and is therefore represented by a restricted bag representation. Since the inefficient-DP constructs these restricted decom-

positions recursively, the corresponding restricted bag representations are compatible by definition. Hence, the algorithm enumerates a valid restricted bag representation at the root and concludes that the w-treewidth is at most k.

This direction only requires that every partial decomposition is represented by some restricted bag representation. Since the construction of the restricted bag representation defined above applies to any tree-decomposition, this indeed is the case.

Lemma 8.8: *Let t be a host-node and let X be a tree-decomposition of G. Then, there exists a restricted bag representation of X with respect to t.*

For the other direction, we must show how to construct a representative tree-decomposition of G when the enumeration algorithm outputs, at each host-node, a valid restricted bag representation. To this end, we first establish several key properties and then provide the full correctness argument. First, we show that all bag-equivalent decompositions have the same weight. Consequently, the particular choice of representative within an equivalence class is irrelevant. Next, we prove that the constructibility relation of the inefficient DP extends to all bag-equivalent decompositions. As discussed above, this guarantees the existence of an underlying tree-decomposition. We then show that a representative for each equivalence class can be reconstructed efficiently. Finally, we must ensure that the algorithm enumerates, at each host-node, all restricted bag representations that are compatible with those computed for its children. This aspect will be addressed later, combined for all three equivalence relations, in the description of the algorithm in Section 8.2.8.

We begin with the first key property and show that all equivalent decompositions have the same weight. Therefore, constructing any representative tree-decomposition of a equivalence class suffices.

Lemma 8.9: *Let t be a host-node. Let X_t and Y_t be two tree-decompositions of G_t that are bag-equivalent. Then, X_t and Y_t have the same weight.*

Proof. Two decompositions are bag-equivalent only if they have the same tree topology and if corresponding bags have the same weight. Hence, their maximum bag weights coincide. ∎

We continue with the second key property and show that such a representative decomposition exists. To this end, we prove that for a sequence of compatible restricted bag representations, the underlying partial tree-decompositions are constructible by the inefficient-DP. As motivated above, the correctness of the inefficient-DP then implies the existence of a representative decomposition.

Lemma 8.10 (Lemma 1 of Althaus and Ziegler [AZ21] for restricted bag representations)**:** *Let t be a host-node. For each child c of t, let X_c and Y_c be two bag-equivalent normalized tree-decompositions of G_c. Let X_t be a normalized tree-decomposition of G_t constructed by the inefficient-DP from the decompositions $\{X_c\}_{c \in children(t)}$. Then, there exists a normalized tree-decomposition Y_t of G_t that is bag-equivalent to X_t such that the inefficient-DP constructs Y_t from the decompositions $\{Y_c\}_{c \in children(t)}$.*

Proof. We distinguish the possible types of the node t. In each case, we analyse the operation that is used to construct X_t from the decompositions $\{X_c\}_{c \in \text{children}(t)}$. We then apply this operation to the decompositions $\{Y_c\}_{c \in \text{children}(t)}$ and obtain Y_t. We have to show that this construction yields a decomposition that is bag-equivalent to X_t. The following general observations hold in all cases. First, since, for each child c of t, X_c and Y_c are bag-equivalent,

they have the same tree topology, and corresponding bags have identical restricted bags and identical weights. Second, since the inefficient-DP does not modify the tree topology, X_c and X_t have the same topology, for each child c of t. Analogously, Y_c and Y_t have the same topology. Hence, it suffices to compare the restricted bags and the weights of the unrestricted bags for corresponding nodes in X_t and Y_t. We use these observations to describe, for each node-type, the operations required to construct Y_t and the arguments establishing bag-equivalence between X_t and Y_t.

If t is a forget-node t with child c, then we have $X_t = X_c$ and we define $Y_t = Y_c$. In the restricted bag representations, only the restricted bags may change. However, since $B(t) \subseteq B(c)$, the intersection is taken with a smaller set, and therefore the restricted bags remain unchanged.

If t is an introduce-node with child c and introduced vertex v, then the inefficient-DP constructs X_t from X_c by adding the vertex v to a valid subtree S. We construct Y_t from Y_c by adding v to the same subtree S. This choice covers all neighbours of v in G_t, since v is only adjacent to vertices contained in $B(c)$, which appear in the same bags in both decompositions. Furthermore, this construction ensures identical restricted bags, so it remains to compare the weights of the unrestricted bags. By vertex addition computability (R3), the weight of a bag depends only on its previous weight and on the neighbourhood of v within the bag. By edge coverage of the host-decomposition, v is adjacent only to vertices contained in $B(t)$. Consequently, these neighbourhoods are identical for corresponding nodes in X_t and Y_t, and therefore the weights coincide.

If t is a join-node with children c_1 and c_2, then the inefficient-DP constructs X_t by taking the union of corresponding bags in X_{c_1} and X_{c_2}. We construct Y_t analogously from Y_{c_1} and Y_{c_2}. Again, the construction guarantees identical restricted bags, so it remains to compare the weights of the unrestricted bags. By union computability (R4), the weight of the union of two bags depends only on the weights of the two bags and on their intersection. Since these are identical for corresponding bags in X_t and Y_t, the weights coincide. ∎

Finally, we show that a representative decomposition can be constructed efficiently. This is the third key property. To this end, we introduce the following definitions. First, we extend the definition of compatibility to descendants. Let t and t' be two host-nodes such that t' is a descendant of t. A restricted bag representation of a tree-decomposition of G_t is said to be *compatible* with a restricted bag representation of a tree-decomposition of $G_{t'}$ if, for every host-node on the subpath of T between t and t', there exists a restricted bag representation that is compatible with the representation chosen for its parent. We say that a tree-decomposition (T^R, B^R) of G *realizes* a valid restricted bag representation R for the host-root if, for every host-node t, the restricted bag representation of the restriction of (T^R, B^R) to G_t is compatible with R. The *recursive size* $s(R)$ of a valid restricted bag representation R for the host-root is defined as the maximum size over all restricted bag representations that are compatible with R.

Lemma 8.11: *Let $r \in V(T)$ be the host-root. Let R be a given valid restricted bag representation of a constructed tree-decomposition of G_r. Then, a normalized tree-decomposition of G that realizes R can be computed in time $s(R)^{\mathcal{O}(1)} \cdot |V(G)|$.*

Proof. We begin with, for each host-leaf, a restricted bag representation that is compatible with R and construct the decomposition bottom-up along the host-tree. Observe, that Lemma 8.10 ensures that, in each step, there exists a decomposition that can be constructed by the inefficient-DP from the current decompositions for the child nodes.

We fix the representative decomposition to have the same topology as the restricted bag representations. Here, recall that the inefficient-DP does not modify the topology and hence all restricted bag representations that are compatible with R have the same topology. In each of the $|V(T)|$ steps, we follow the construction of the inefficient-DP, using the given restricted bag representations to guide choices whenever the inefficient-DP allows multiple options. More precisely, for forget-nodes we leave the representative decomposition unchanged, for introduce-nodes we add the introduced vertex to the subtree prescribed by the restricted bag representation, and for join-nodes we unite the corresponding bags. In each step, the content of a single bag can be computed in time linear in the bag size.

By performing these iterative computations, we obtain a full tree-decomposition of G that realizes R. An induction argument shows that, for every host-node, the restricted bag representation of the corresponding restriction is indeed compatible with R. ∎

We now present the full correctness argument for the second direction. Suppose that the algorithm returns that the w-treewidth is at most k, that is, it computes a valid restricted bag representation at the root. We now explain how in this case a tree-decomposition of weight at most k can be constructed. By correctness of the enumeration algorithm, for every host-node, there exists a restricted bag representation such that the representations for every parent-child-pair are compatible. By definition of compatibility, for every edge tt' of the host-tree, there exist tree-decompositions of G_t and $G_{t'}$ corresponding to these states such that the former is constructed from the latter by the inefficient-DP. By Lemma 8.10, this compatibility extends to the individual partial tree-decompositions. Using this property, we select, at each host-node, a partial tree-decomposition that can be obtained by the inefficient-DP from the decompositions chosen for its children. Proceeding in this way yields a representative tree-decomposition of the entire graph. Since all decompositions within the same equivalence class have the same weight, the exact choice of representative does not affect the result. Consequently, the constructed representative decomposition has weight k, and therefore the w-treewidth of G is at most k. For a constructive algorithm, we use Lemma 8.11 to reconstruct such representatives solely from the restricted bag representations.

8.2.5 Refining the equivalence relation: Cores

We introduce a second equivalence relation on partial tree-decompositions which extends the first one with the aim of bounding the number of leaves in each tree. The key idea is that Corollary 8.5 allows us to restrict our search to extensions of the current partial decomposition where certain leaves do not contain vertices not yet introduced by the host-decomposition. Hence, these leaves need not be stored explicitly. We begin by defining the limited information characterizing this equivalence and then consider the correctness arguments.

Let $t \in V(T)$ be a host-node and consider the restricted bag representation of a constructed tree-decomposition with respect to t. We define the *t-core* of this constructed tree-decomposition by the exhaustive application of the following rule to the restricted bag representation. If there exists a leaf whose restricted bag with respect to t is a subset of the restricted bag with respect to t of its parent, remove this leaf from the current tree. This operation can also be applied to nodes that became leaves through previous applications. Note, that this iterative removal of leaves corresponds to the removal of nice attached subtrees (see also Corollary 8.5). Additionally, the weight of the largest bag is stored, since this operation may remove the bag of maximum weight. The *size* of a t-core is defined as the product of the number of nodes and edges in the tree of the decomposition, the number of vertices contained

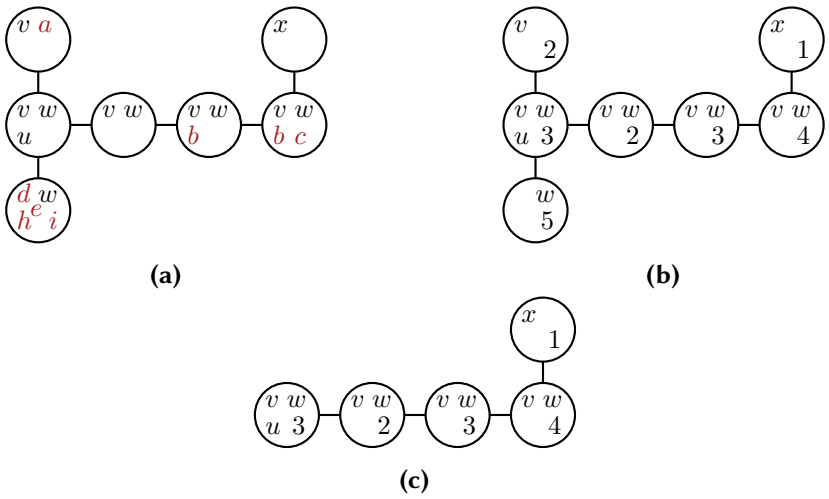


Figure 8.5: The figure illustrates the different stages of limited information extending Figure 8.4. Part (a) depicts a partial tree-decomposition. The nodes of the host-bag are shown in black, the forgotten nodes are shown in red. Part (b) illustrates the restricted bag representation of this decomposition, here the numbers illustrate the weights obtained from the weight function based on bag-sizes. Part (c) shows the resulting t-core. Here, the maximum weight of 5 is additionally stored.

in each restricted bag, and the number of possible weights. Two tree-decompositions are called *core-equivalent* if their t-cores coincide. This includes both the restricted bags of the core and the weight of the maximum bag. Since equality on cores is an equivalence relation, it follows that core-equivalence is itself an equivalence relation. We visualize t-cores in Figure 8.5.

We verify that the correctness arguments outlined for restricted bags in Section 8.2.4 still hold when considering cores instead of restricted bag representations. As before, every tree-decomposition admits a core, and core-equivalent decompositions have the same weight.

Lemma 8.12: *Let t be a host-node and let X be a tree-decomposition of G. Then, there exists a t-core of X.*

Lemma 8.13: *Let t be a host-node. Let X_t and Y_t be two tree-decompositions of G_t that are core-equivalent. Then, X_t and Y_t have the same weight.*

Similar to before, we show that the constructibility relation of the inefficient-DP extends to all core-equivalent decompositions. Here, we use that the inefficient-DP only constructs normalized tree-decompositions. In particular, we may assume the property of Corollary 8.5. To eliminate differences between core-equivalent partial tree-decompositions, we allow the inefficient-DP to add leaves before applying its operation. In particular, this modification preserves the width, validity, and containment in an equivalence class of the decomposition. The only change to the correctness arguments this necessitates is that, when constructing the representative decomposition, we now may need to add some leaves before each step.

Lemma 8.14 (Lemma 1 of Althaus and Ziegler [AZ21] for cores)**:** *Let t be a host-node. For each child c of t, let X_c and Y_c be two core-equivalent normalized tree-decompositions of G_c. Let X_t be a normalized tree-decomposition of G_t constructed by the inefficient-DP from the decompositions $\{X_c\}_{c \in children(t)}$. Then there exists a normalized tree-decomposition Y_t of G_t that is core-equivalent to X_t such that the inefficient-DP constructs Y_t from the decompositions $\{Y_c\}_{c \in children(t)}$ with leaves added to them.*

Proof. We proceed similarly to Lemma 8.10. In particular, we analyse the operation that is used to construct X_t from the decompositions $\{X_c\}_{c\in\text{children}(t)}$. We then apply this operation to the decompositions $\{Y_c\}_{c\in\text{children}(t)}$ with added leaves and obtain Y_t. Since this lemma formalizes how the operation of the inefficient-DP is chosen, it remains to determine which leaves must be added to the decompositions $\{Y_c\}_{c\in\text{children}(t)}$ so that the construction is valid, and to show that the resulting decomposition is core-equivalent to X_t. For the first part, we add the leaves required to ensure that every relevant node of X_c has a corresponding node in Y_c, where c is a child of t. For the second part, we use that the restricted bags of X_t and Y_t are identical.

We begin with the addition of leaves and distinguish cases according to the type of t. If t is a forget-node with child c, then the inefficient-DP performs no modification. Consequently, we do not need to change the topology of Y_c.

If t is an introduce-node with child c and introduced vertex v, then we must ensure that, for every bag of X_c to which v can be added, there exists a corresponding bag in Y_c. Observe, that by Corollary 8.5, every node outside the core of X_c that receives v contains no forgotten vertices. Hence, its unrestricted bag coincides with its restricted bag, which itself is a subset of the restricted bag of its parent. Therefore, whenever such a leaf is not already present in Y_c, we add a corresponding leaf with the same unrestricted bag and the same weight to the corresponding parent in Y_c. This procedure can be applied iteratively and therefore also allows the addition of entire attached subtrees.

If t is a join-node with children c_1 and c_2, then we ensure that Y_{c_1} and Y_{c_2} have identical topologies. This is achieved by iteratively adding leaves that are present in one decomposition but not in the other. All such added leaves receive empty bags.

We now address the second property and show that X_t ad Y_t are core-equivalent when Y_t is constructed as above. Since all nodes shared by X_t and Y_t have identical restricted bags, and since the removal of leaves depends only on the restricted bags, it suffices to consider nodes that appear in exactly one of X_t and Y_t. For these nodes, we must show that they are removed during the construction of the respective t-cores. Observe, that all such nodes were already removed in the construction of the core of the child decompositions, and that their restricted bags are not modified by the construction. Consequently, these nodes can again be removed when constructing the t-cores of X_t and Y_t, respectively. Moreover, the weights of these removed bags remain unchanged. Hence, the maximum weight among the nodes of X_t and Y_t is identical. ∎

Finally, we show that a representative for each equivalence class can be reconstructed efficiently. Recall, that a tree-decomposition is said to realize a restricted bag representation R if, for every host-node, the restricted bag representation of the corresponding restriction is compatible with R. Moreover, the recursive size $s(R)$ was defined as the maximum size among all restricted bag representations that are compatible with R. We extend these definitions from restricted bag representations to cores.

Lemma 8.15: *Let $r \in V(T)$ be the host-root. Let C be a given valid r-core of a constructed tree-decomposition of G_r. Then, a normalized tree-decomposition of G that realizes C can be computed in time $s(C)^{\mathcal{O}(1)} \cdot |V(G)|$.*

Proof. We begin with, for each host-leaf, a restricted bag representation that is compatible with R and construct the decomposition bottom-up along the host-tree. Observe, that Lemma 8.14 ensures that, in each step, there exists a decomposition that can be constructed by the inefficient-DP from the current decompositions for the child-nodes.

In each step, we may first add some leaves to the current representative decompositions for the child-host-nodes and then perform a single operation of the inefficient-DP. To this end, we first determine the required changes to the topology and then use the arguments leading to Lemma 8.11 to identify the bag contents. For leaf host-nodes, we directly adopt the topology of the given core, which consists of a single bag. For internal host-nodes, we compare the current decomposition with the prescribed core. If the core contains additional attached subtrees that are not present in the current decomposition, we add them to the decomposition and assign to each new node the restricted bag specified by the core as its bag. Moreover, decompositions that are to be joined must have identical topologies. As before, this is achieved by adding the necessary leaves.

To compute these modifications, we iteratively maintain a mapping between corresponding nodes in the cores of decompositions of the parent- and child-host-nodes. Using this mapping, the topology changes in each step can be performed in time linear in the size of the corresponding core times the size of each bag.

By performing these iterative computations, we obtain a full tree-decomposition of G that realizes C. The following two observations are immediate. First, adding leaves preserves the validity and width of the tree-decomposition. Second, leaves that are removed during the construction of the core of a child-decomposition and contain forgotten vertices are also removed during the construction of the core of the parent-decomposition, since, by Corollary 8.5, they do not receive additional vertices in subsequent steps. Using this, an induction argument shows that, for every host-node, the core of the corresponding restriction is indeed compatible with C. ∎

8.2.6 A further refinement of the equivalence relation: Compressed cores

We introduce the third equivalence relation on partial tree-decompositions with the aim of bounding the length of paths of nodes of degree 2 in each tree. Here, the key idea is that Lemma 8.6 allows us to restrict our search to extensions of the current partial decomposition where some nodes receive the same vertices as their neighbours. Thus, those nodes do not have to be stored explicitly. In the following, we assume for the constructed tree-decompositions that each path of nodes of degree 2 contains at most $|V|$ nodes instead of bounding the number of nodes in each tree-decomposition by $|V|$. We begin by defining the limited information characterizing this equivalence and then consider the correctness arguments.

Let $t \in V(T)$ be a host-node and consider the t-core of a constructed tree-decomposition. Consider a maximum length path of nodes of degree 2 in this core such that all nodes on this path have the same restricted bag. Observe, that such a path forms a t-nice caterpillar-path in the full decomposition whose caterpillar-legs are not present in the t-core. We replace each such path with a single node that has the same restricted bag and, at this node, additionally store the sequence of weights of the bags along the original path. We call the resulting object the *compact representation* of a core.

In the next step, we remove certain elements from these weight sequences in order to reduce their length. The *typical sequence* $\tau(A)$ of an integer sequence $A = (a_1, \ldots, a_q)$ is defined by exhaustive application of the following two cutting-operations. First, we may remove consecutive duplicates. Formally, if $a_i = a_{i+1}$ for $1 \leq i \leq q-1$, we delete a_{i+1} from the sequence, i.e., we replace $(a_1, \ldots, a_q)$ with $(a_1, \ldots, a_i, a_{i+2}, \ldots, a_q)$. Second, we may remove intervals whose values stay between their endpoints. Formally, let $1 \leq i < j \leq q$ be

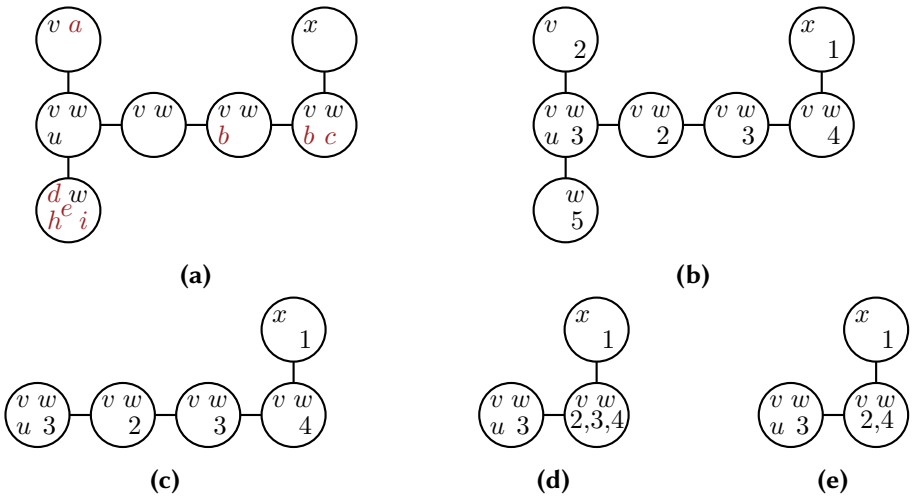


Figure 8.6: The figure illustrates the different stages of limited information extending Figure 8.4 and Figure 8.5. Part (a) depicts a partial tree-decomposition. The nodes of the host-bag are shown in black, the forgotten nodes are shown in red. Part (b) illustrates the restricted bag representation of this decomposition, here the numbers illustrate the weights obtained from the weight function based on bag-sizes. Part (c) shows the resulting t-core. Part (d) depicts the compact representation of this core. Part (e) shows the compressed core. For the representations depicted in part (c) to (e) the maximum weight of 5 is additionally stored.

indices such that $\min(a_i, a_j) \leq a_\ell \leq \max(a_i, a_j)$ for all entries a_ℓ with $i \leq \ell \leq j$. Then, we remove all entries strictly between i and j from the sequence, i.e., we replace $(a_1, \ldots, a_q)$ with $(a_1, \ldots, a_i, a_j, \ldots, a_q)$.

Using this, two integer sequences are considered equivalent, if their typical sequences coincide. We use these typical sequences to formally define our last equivalence relation on tree-decompositions. The *compressed core* of a tree-decomposition is obtained from its compact core by replacing each integer sequence associated with a node with its typical sequence. The *size* of a compressed core is defined as the product of the number of nodes and edges in the tree of the decomposition, the number of vertices contained in each restricted bag, the length of an integer sequence, and the number of possible weights. Then, two tree-decompositions are considered *sequence-equivalent*, if their compressed cores coincide. Since equality on compressed cores is an equivalence relation, it follows that sequence-equivalence is itself an equivalence relation. Observe, that the entries removed from a integer sequence correspond exactly to those nodes that by Lemma 8.6 receive the same vertices as their neighbours in future steps. We visualize compressed cores in Figure 8.6.

We continue by showing that typical sequences are well-defined. To this end, we must prove that $\tau(A)$ is unique.

Lemma 8.16 (Property (P1) of [AZ21], shown in Section 3.5)**:** *Let A be an integer sequence. Then, the typical sequence $\tau(A)$ of A is unique.*

Proof Sketch. We provide a brief proof sketch, as this is a general property of typical sequences. A full proof can be found in Section 3.5 of [AZ21]. Consider the following claim. For all indices $i < j$ and all $\ell \notin \{i+1, \ldots, j-1\}$, the entry a_ℓ can be removed from the sequence $(a_1, \ldots, a_q)$ if and only if this integer can be removed from the sequence $(a_1, \ldots, a_i, a_j, \ldots, a_q)$. It is straightforward to see that the lemma follows from this claim. The claim itself can be

proven by a case distinction over all possible applications of the two reduction rules. For each case, it must be verified that either the relevant extremal values remain in place or that an even smaller or larger value exists which justifies the same reduction. ∎

We verify that the correctness arguments outlined for restricted bags in Section 8.2.4 still hold when considering compressed cores instead of cores and restricted bag representations. As before, each tree-decomposition has a compressed core and sequence-equivalent decompositions have the same weight.

Lemma 8.17: *Let t be a host-node and let X be a tree-decomposition of G. Then, there exists a compressed t-core of X.*

Lemma 8.18: *Let t be a node of the given tree-decomposition. Let X_t and Y_t be two tree-decompositions of G_t that are sequence-equivalent. Then, the weight of X_t equals the weight of Y_t.*

Similar to the previous equivalence relations, we show that the constructibility relation of the inefficient-DP extends to all sequence-equivalent decompositions. Here, we use the fact that the inefficient-DP only constructs normalized tree-decompositions. In particular, we may assume the property of Lemma 8.6. Similar to before, we now additionally allow the inefficient-DP to duplicate nodes before applying its operation.

We first state several auxiliary lemmas. For two integer sequences A and B, we write $A \cdot B$ for their concatenation.

Lemma 8.19: *Let A_1, A_2, B_1 and B_2 be integer sequences. If $\tau(A_1) = \tau(B_1)$ and $\tau(A_2) = \tau(B_2)$, then $\tau(A_1 \cdot A_2) = \tau(B_1 \cdot B_2)$.*

Proof. We compute the typical sequences of $A_1 \cdot A_2$ and $B_1 \cdot B_2$ in two stages and show that the resulting sequences coincide. We first apply to $A_1 \cdot A_2$ the same operations that are used to compute $\tau(A_1)$ and $\tau(A_2)$ from A_1 and A_2, respectively. This yields a representation of $A_1 \cdot A_2$ as $\tau(A_1) \cdot \tau(A_2)$. Analogously, we obtain a representation of $B_1 \cdot B_2$ as $\tau(B_1) \cdot \tau(B_2)$. Using the assumptions, we have $\tau(A_1) \cdot \tau(A_2) = \tau(B_1) \cdot \tau(B_2)$. Applying the same reduction operations to both representations therefore yields $\tau(A_1 \cdot A_2) = \tau(B_1 \cdot B_2)$. ∎

For the next result, we write $A + c$ for the sequence obtained by adding the integer c to every element of the integer sequence A. Similarly, $A - c$ represents the element-wise subtraction.

Lemma 8.20: *Let A be an integer sequences and let c be an integer. Then, $\tau(A + c) = \tau(A) + c$ and $\tau(A - c) = \tau(A) - c$.*

Proof. Adding or subtracting the same constant to all entries preserves all relative comparisons between elements of the sequence. Hence, the reduction operations defining the typical sequence apply in exactly the same way to A and to $A \pm c$, which proves the claim. ∎

Similarly, in the next lemma, $A + B$ denotes the element-wise sum of two sequences of equal length.

Lemma 8.21: *Let A and B be two integer sequences of the same length, and let C be the integer sequence corresponding to the path obtained by pairwise taking the union of the bags of the paths represented by A and B. Then, there exist integer sequences A' and B' obtained from $\tau(A)$ and $\tau(B)$, respectively, by duplicating entries, such that $\tau(C) = \tau(A' + B' - c)$. Here, c is some value depending on the intersection of the paths corresponding to A and B.*

Proof. We interpret the construction of C from A and B as iteratively adding a vertex of the path corresponding to B to the bags of the path corresponding to A. By Lemma 8.6, each vertex is added either to no node to all nodes between two nodes corresponding to entries of the typical sequence. Since all nodes between two nodes that are represented in the typical sequence receive the same vertices, union computability (R4) implies that they also receive the same additional weight. In particular, the total weight of a node is obtained by adding the initial weights and subtracting a correction term that depends on the intersection. Consequently, by Lemma 8.20, the typical sequence of C is obtained by pairwise combining suitable entries of the typical sequences of A and B. This induces an alignment between the two typical sequences, specifying which entry of $\tau(A)$ is combined with which entry of $\tau(B)$. Such an alignment can be realized by duplicating entries in the two sequences until all corresponding entries occur at the same indices. ∎

We now apply these auxiliary results to prove the lemma.

Lemma 8.22 (Lemma 1 of Althaus and Ziegler [AZ21] for compressed cores)**:** *Let t be a host-node. For each child c of t, let X_c and Y_c be two sequence-equivalent normalized tree-decompositions of G_{c_i}. Let X_t be a normalized tree-decomposition of G_t constructed by the inefficient-DP from the decompositions $\{X_c\}_{c\in children(t)}$. Then there exists a normalized tree-decomposition Y_t of G_t that is sequence-equivalent to X_t such that the inefficient-DP constructs Y_t from the decompositions $\{Y_c\}_{c\in children(t)}$ with leaves added to them and nodes duplicated in them.*

Proof. We proceed similarly to Lemma 8.14. In particular, we analyse the operation that is used to construct X_t from the decompositions $\{X_c\}_{c\in\text{children}(t)}$. We then apply this operation to the decompositions $\{Y_c\}_{c\in\text{children}(t)}$ with added leaves and duplicated nodes and obtain Y_t. Since this lemma formalizes how the operation of the inefficient-DP and the addition of leaves is chosen, it remains to determine which nodes must be duplicated in the decompositions $\{Y_c\}_{c\in\text{children}(t)}$ so that the construction is valid, and to show that the resulting decomposition is sequence-equivalent to X_t. We use that the bag contents and the general tree topologies of X_t and Y_t are identical. Consequently, it suffices to consider paths of degree 2 nodes and, in particular, the corresponding integer sequences.

We first make several general observations and then distinguish cases according to the type of t. Observe, that which underlying paths are represented by a node of the compressed core can change only in two ways as a consequences of changes to the restricted bags. Either two neighbouring paths are assigned the same restricted bag and the corresponding nodes of the compressed core merge, which corresponds to concatenating their integer sequences. Or alternatively some nodes of a path are assigned different restricted bags and the corresponding node of the compressed core splits into two, which corresponds to splitting an integer sequence. Next, observe that by Lemma 8.6, the decomposition X_t contains two consecutive nodes with identical bags at every split point, since otherwise the nodes directly before and after the split point would form a violating sequence.

If t is a forget-node with child c, then either no changes to the typical sequences of X_c occur or neighbouring nodes are merged and their sequences are concatenated. By Lemma 8.19, performing the same operations on Y_c yields equivalent sequences for X_t and Y_t.

If t is an introduce-node with child c and introduced vertex v, then a integer sequence only changes when v is added to some bag corresponding to an entry of the sequence. By Lemma 8.6, the first and last affected entries are represented in the typical sequence. We modify Y_c analogously in order to obtain Y_t. At every split point, we duplicate the corresponding

entry so that it appears in both resulting subsequences, as described above, and then add v to the bags of the corresponding subpath. Note, that the weight change depends only on the restricted bag by vertex-addition computability (R3), and that the restricted bags are identical along the corresponding paths in X_c and Y_c. Therefore, by Lemma 8.20, the resulting sequences remain equivalent.

If t is a join-node with children c_1 and c_2, then the bags in the underlying paths are united. By Lemma 8.21, the typical sequence of a node in the compressed core of X_t can be obtained from the typical sequences of the corresponding nodes in the compressed cores of X_{c_1} and X_{c_2} by duplicating suitable entries so that a particular alignment is realized, and then performing element-wise addition minus a common value. We construct the sequence for the corresponding node in the compressed core of Y_t analogously from the compressed cores of Y_{c_1} and Y_{c_2} by using the same alignment. To this end, we duplicate sufficiently many nodes between each pair of nodes represented in the typical sequence so that the required alignment is achieved. Consequently, the resulting typical sequences coincide. ∎

Finally, we show that a representative for each equivalence class can be reconstructed efficiently. Recall, that a tree-decomposition is said to realize a restricted bag representation R if, for every host-node, the restricted bag representation of the corresponding restriction is compatible with R. Moreover, the recursive size $s(R)$ was defined as the maximum size among all restricted bag representations that are compatible with R. We extend these definitions from restricted bag representations and cores to compressed cores.

Lemma 8.23: *Let $r \in V(T)$ be the host-root. Let C be a given valid compressed core of a constructed tree-decomposition of G_r. Then, a normalized tree-decomposition of G that realizes C can be computed in time $s(C)^{\mathcal{O}(1)} \cdot |V(G)|$.*

Proof. We begin with, for each host-leaf, a restricted bag representation that is compatible with R and construct the decomposition bottom-up along the host-tree. Observe, that Lemma 8.22 ensures that, in each step, there exists a decomposition that can be constructed by the inefficient-DP from the current decompositions for the child-nodes.

In each step, we may first duplicate certain nodes of the current representative decompositions for the child-host-nodes and then perform a single operation of the inefficient-DP. For this operation, we use the arguments leading to Lemma 8.15 to determine the overall tree topology and the bag contents. For paths of degree 2 nodes with identical restricted bags, we refine the construction in order to match the prescribed typical sequences.

We distinguish cases based on the type of the current host-node. For forget-nodes, the representative decomposition remains unchanged.

For an introduce-node, we must determine which node of the representative decomposition corresponds to the entry of the typical sequence that is the last to receive the introduced vertex. This entry can be identified in the required time by iteratively maintaining a marking of the typical entries of a integer sequence throughout the construction. We then proceed as established above: More precisely, we duplicate the node corresponding to this entry and add the introduced vertex to all nodes up to the first occurrence of the duplicated node. Again, this can be done in the required time.

For join-nodes, we must determine which bags are united. As seen before, by Lemma 8.21, it suffices to compute an alignment of the entries of the typical sequences of the current decompositions that yields the target typical sequence prescribed by the given compressed core. Equivalently, we must determine which entries of the typical sequences need to be duplicated so that corresponding entries align. To this end, we create a queue for each current

typical sequence and iteratively combine entries to produce the parent typical sequence. In each step, we add up the two front elements and remove one of the two front elements from its queue so that the resulting sequence matches the target typical sequence. This procedure is linear in the length of the typical sequences. Finally, we duplicate each node as often as necessary to ensure its index aligns with the index of the corresponding node. This can be done in time polynomial in the size of the compressed core.

By performing these iterative computations, we obtain a full tree-decomposition of G that realizes C. An induction argument shows that, for every host-node, the compressed core of the corresponding restriction is indeed compatible with C. ∎

Therefore, we have shown that enumerating, for each host-node, all compressed cores that are compatible with those computed for its children suffices to correctly determine the w-treewidth of G.

8.2.7 Bounding the number of compressed cores

In the next step, we show that the number of compressed cores per host-node is sufficiently small so that enumeration is feasible. We first bound this number in terms of the width of the host-decomposition and then use boundedness (R5) to extend this to w-treewidth.

We begin by bounding the number of compressed cores enumerated at a single node of the host-tree-decomposition (T, B) of width tw. To this end, we bound the number of possibilities for the following components: the maximum weight, the tree topology, the assignment of vertices to bags, and the typical sequences. To build a foundation for those bounds, we begin by bounding the number of nodes in a compressed core. Note, that each leaf of a compressed core must contain a unique vertex of the host-bag, since otherwise its restricted bag would be a subset of the restricted bag of its adjacent node and the leaf would have been removed from the compressed core. Hence, a compressed core contains at most tw leaves and at most tw nodes of degree 3. To bound the number of nodes in a compressed core, it remains to bound the number of nodes of degree 2. We show that along each path between branching nodes, there are at most 2 tw such nodes. This is since, each bag along such a path must differ from its predecessor, and in the worst case, each vertex of the host-bag is added once and removed once. Hence, the total number of nodes in a compressed core is at most $2\,\mathrm{tw}^2$.

Choosing a tree topology on at most $2 \cdot \mathrm{tw}^2$ nodes can be described as making $(2\,\mathrm{tw}^2 - 1)$ choices, each selecting one of $4 \cdot \mathrm{tw}^4$ potential edges (or no edge in case fewer nodes are used). The resulting topology is then formed from the selected edges together with their incident vertices. Using this argument, the number of all tree topologies on at most $2\,\mathrm{tw}^2$ nodes is bounded by

$$\begin{aligned}
&\left(4\,\mathrm{tw}^4 + 1\right)^{2\,\mathrm{tw}^2 - 1} \\
&= 2^{\log(4\,\mathrm{tw}^4 + 1) \cdot (2\,\mathrm{tw}^2 - 1)} \\
&\leq 2^{4 \cdot \log(4\,\mathrm{tw} + 1) \cdot (2\,\mathrm{tw}^2 - 1)} \\
&\in 2^{\mathcal{O}(\mathrm{tw}^3)}.
\end{aligned}$$

We now turn to the assignments from vertices to bags. For each node, we choose a restricted bag, i.e., a subset of $B(t)$, together with a weight. There are at most 2^{tw} subsets and at most $k + 1$ possible weights, which yields at most $(k + 1) \cdot 2^{\mathrm{tw}}$ possibilities per node. Since we choose a restricted bag and weight for each of the $2\,\mathrm{tw}^2$ nodes of the tree, this gives at most $\left((k + 1) \cdot 2^{\mathrm{tw}}\right)^{2\,\mathrm{tw}^2} = (k + 1)^{2\,\mathrm{tw}^2} \cdot 2^{2\,\mathrm{tw}^3}$ possible assignments from vertices to bags.

Finally, we turn to typical sequences. First, we show that the number of typical sequences can be bounded by a function of the number of distinct values occurring in those sequences.

Lemma 8.24 (Property (P2) of [AZ21], shown in Section 3.5)**:** *Let q be an integer. Then, there are at most $2 \cdot 2^{2q}$ typical sequences containing q distinct values.*

Proof Sketch. We provide a brief proof sketch as this is a general property of typical sequences. A full proof can be found in Section 3.5 of [AZ21]. We first claim that a typical sequence with q distinct values has length at most $2q$. To show this claim first observe that every typical sequence can be written as the concatenation of a subsequence ending with the minimal value and a subsequence beginning with the maximal value (or vice versa). We now show that each of these two subsequences has length at most q. Such a subsequence must alternate between the smallest and largest value not yet present in this sequence and each value can occur only once. This is, since otherwise the subsequence between the extremal occurrences of the minimum and maximum value could be removed by a reduction step. Thus, the claim follows.

We use this claim and the above discovered structure of typical sequences to bound their number. Note that each of the two subsequences introduced above is uniquely determined by the subset of values it contains, since the order is fixed by the alternation. Hence, there are at most 2^q possibilities per subsequence, and therefore at most $(2^q)^2 = 2^{2q}$ combinations of the two subsequences. Finally, we multiply by two to account for the choice of whether the minimal or maximal value appears first. ∎

Using this lemma it suffices to bound the number of distinct weight values to bound the number of typical sequences for one node. Since all constructed decompositions have weight at most k and weights are integers, each sequence contains at most $k+1$ distinct values. Therefore, there are at most $2 \cdot 2^{2k+2} \subseteq 2^{\mathcal{O}(k)}$ typical sequences per node. Choosing a typical sequence for each of the $2\,\mathrm{tw}^2$ nodes yields at most $\left(2^{\mathcal{O}(k)}\right)^{2\,\mathrm{tw}^2} \subseteq 2^{\mathcal{O}(k \cdot \mathrm{tw}^2)}$ possibilities.

Combining all bounds and noting that the maximum bag weight is an integer in $\{0, \ldots, k\}$, we obtain a total of at most

$$
\begin{aligned}
&\underbrace{(k+1)}_{\text{weight}} \cdot \underbrace{2^{\mathcal{O}(\mathrm{tw}^3)}}_{\text{topology}} \cdot \underbrace{(k+1)^{2\,\mathrm{tw}^2} \cdot 2^{2\,\mathrm{tw}^3}}_{\text{assignments to bags}} \cdot \underbrace{2^{\mathcal{O}(k \cdot \mathrm{tw}^2)}}_{\text{sequences}} \\
&= (k+1)^{2\,\mathrm{tw}^2+1} \cdot 2^{\mathcal{O}(\mathrm{tw}^3 + k \cdot \mathrm{tw}^2)} \\
&= 2^{\log(k+1) \cdot (2\,\mathrm{tw}^2+1)} \cdot 2^{\mathcal{O}(\mathrm{tw}^3 + k \cdot \mathrm{tw}^2)} \\
&\subseteq 2^{(k+1) \cdot (2\,\mathrm{tw}^2+1)} \cdot 2^{\mathcal{O}(\mathrm{tw}^3 + k \cdot \mathrm{tw}^2)} \\
&= 2^{\mathcal{O}(\mathrm{tw}^3 + k \cdot \mathrm{tw}^2)} \\
&= 2^{\mathcal{O}(f(\ell)^3 + k \cdot f(\ell)^2)}
\end{aligned}
$$

compressed cores per host-node, where $\ell \in \mathcal{O}(k)$ is the weight of the host-decomposition. In the last step we used that all host-bags have weight at most ℓ and that, by boundedness (R5), the size of each host-bag, and thus tw, is bounded by $f(\ell)$. Consequently, the number of compressed cores per host-node is bounded by a function of k.

8.2.8 An enumeration algorithm for compatible compressed cores

We now present an algorithm for enumerating all valid compressed cores in FPT-time. To this end, we employ a dynamic program on the given host-tree-decomposition, which we call the *FPT-DP*. Here, a partial solution is represented by a compressed core, i.e., a tree in which each node is associated with a restricted bag, an integer sequence and a weight. Furthermore, the maximum weight over all bags in the tree-decomposition represented by this compressed core is stored.

Recall, that we want to compute, for a given host-node, all compressed cores that are compatible with the cores enumerated for its children. For this computation we may only use the cores enumerated at the children as input. We distinguish the different node types and, for each type, first describe how compatible compressed cores and their underlying partial tree-decompositions may differ and then derive recursive rules for their computation. Finally, we analyse the running time and show that each recursive computation can be performed in time polynomial in the number of compressed cores.

Leaf-node. Let t be a leaf-node. A partial tree-decomposition of G_t is a tree with empty bags. Hence, all such decompositions have the same t-core. This core consists of a single node with empty bag and a weight sequence containing the single element $w(\emptyset)$, which is also the maximum weight.

Forget-Node. Let t be a forget-node with child t' and forgotten vertex v. Also let (T_t, B_t) be a partial tree-decomposition of G_t constructed by the inefficient-DP from a partial decomposition $(T_{t'}, B_{t'})$ of $G_{t'}$. We first analyse how these decompositions and their compressed cores differ. Later, we use this characterization of the differences to give rules for how to compute all valid compressed cores for a host-node solely from the compressed cores of its child node. The decompositions themselves are not changed by the inefficient-DP at forget-nodes. However, in the compressed cores, the vertex v is not present in the restricted bags of (T_t, B_t). As a consequence, additional leaves may become removable. In particular, the t-core of (T_t, B_t) is a subtree of the t'-core of $(T_{t'}, B_{t'})$. Moreover, nodes along paths of nodes of degree 2 may merge. In such cases, their integer sequences are concatenated and subsequently reduced to their typical sequence. The weights remain unchanged throughout this process.

Our computation of the compressed t-cores emulates these changes. We update the restricted bags, keep the weights, and remove all removable leaves. Whenever two nodes merge, we concatenate their typical sequences and subsequently compute the typical sequence of the result. Correctness of this procedure for computing typical sequences follows from Lemma 8.16, which ensures that the order in which reductions are applied does not affect the resulting typical sequence. By following the differences between the compressed t'-core and the compressed t-core described above, we ensure that the algorithm is correct for this node-type.

Using a naive implementation, this computation can be performed in time polynomial in the number of compressed cores. More precisely, we inspect every node and, for each node, remove the forgotten vertex v from its bag, test whether the resulting bag can be removed, and check whether it can be merged with a neighbouring bag. To compute the typical sequences, we iteratively test for every subsequence whether it can be removed.

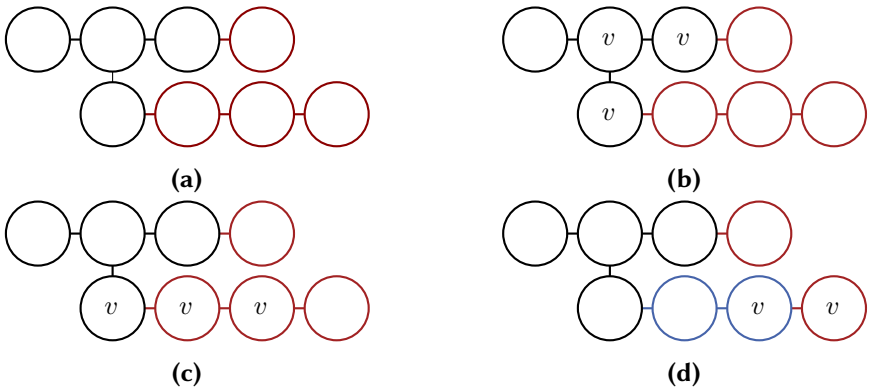


Figure 8.7: The figure illustrates the different cases that arise when adding a vertex v to a partial tree-decomposition. The core is shown in black, while removed nodes are shown in red. Part (a) depicts the decomposition before the introduction of v. Part (b) illustrates the case in which v is added only to nodes of the core. Part (c) shows the case in which v is added to at least one node of the core. Part (d) depicts the case in which v is added only to nodes outside the core. In this situation, the closest node to the core that receives v together with the path connecting it to the core is added to the core. These nodes are highlighted in blue.

Introduce-node. Let t be an introduce-node with child t' and introduced vertex v. Also let (T_t, B_t) be a partial tree-decomposition of G_t constructed by the inefficient-DP from a partial decomposition $(T_{t'}, B_{t'})$ of $G_{t'}$. We first analyse how these decompositions and their compressed cores differ. The tree-decompositions differ only in that the vertex v is added to some subtree. We examine how this affects the compressed cores and distinguish two cases which are illustrated in Figure 8.7. First, consider the case where v is only added to nodes represented in the core. In this case, the changes can be analysed at the level of individual nodes of the compressed core. Let t_c be a node in the compact representation of the t'-core of $(T_{t'}, B_{t'})$. By Lemma 8.6, v is added to the underlying path represented by t_c up to a node corresponding to an entry of its typical sequence. If this node is not the last node represented by t_c, then t_c and its typical sequence are split at this point, since the restricted bags of the nodes that receive v differ from those that do not. By Lemma 8.6, we may assume that $(T_{t'}, B_{t'})$ contains two consecutive nodes with identical bags at the split point and that v is added up to the first of these, since otherwise the nodes directly before and directly after the split point form a violating sequence. Consequently, the typical sequence is split such that both resulting subsequences contain the value at the split point. The second subsequence remains unchanged, while the weights in the first subsequence are increased uniformly. By superset monotonicity (R2), which ensures that adding a vertex preserves weight inequalities, the relative order of values is maintained. Thus, the entries of resulting typical sequence represent the same nodes as before.

However, it is also possible that v is added to nodes of $(T_{t'}, B_{t'})$ not represented in its t'-core. If v is added to at least one node of the core, then any node that was removed during the construction of the t'-core can still be removed during the construction of the t-core, and the topology of the core remains unchanged. This is because for every leaf outside of the core that v is added to, its parent also receives v, and thus the removal rules apply as before. As a result, this case reduces to the previous case in which v is only added to nodes of the core. It therefore remains to consider the case where v is added to a child-node outside the core but not to its parent. In particular, v is not added to any node of the core. In this case, let ℓ be the

node closest to the core among those nodes to which v is added. This node cannot be removed when constructing the t-core of (T_t, B_t), since it is added v but its parent does not receive v. However, all other nodes receiving v can still be removed, as argued above. Furthermore, all nodes on the path from ℓ to the next node represented in the core are also preserved, since they only become leaves through the removal of ℓ. Along this path, the restricted bags form a chain of nested subsets, since these nodes would be removed from the core if not for ℓ. Furthermore, by Corollary 8.5, these nodes contain no forgotten vertices and hence the weight of each full bag equals the weight of its restricted bag. In summary, the compressed core changes in one of two ways: Either some of the nodes of the core additionally contain v or v appears in exactly one node not previously in the core, which is connected to the core by a path of nodes with nested restricted bags.

Our computation of the compressed t-cores emulates these changes. We construct all tree-decompositions obtainable from the following options. First, we may choose a subtree of the core to which the vertex v is added. We must verify that this subtree is valid, that is, a subtree whose bags contain all neighbours of v in G_t. We then update the weights of all affected nodes according to the assumptions of vertex addition computability (R3). This may also require an update of the maximum weight. Additionally, we may split nodes of the core that correspond to leaves of the chosen subtree at an entry of the typical sequence. In this case, we duplicate the entry at the split point, increase the integers in the first subsequence of the typical sequence as described above and keep the integers in the second subsequence of the typical sequence unchanged. Second, alternatively, we may attach a path to some node of the core such that the restricted bags along this path form a chain of strictly nested subsets of the bag of the node to which the path is attached. We then add v to the endpoint of this path. For each node of this attached path, we assign a typical sequence consisting of a single value equal to the weight of the corresponding subset. Again, the maximum bag weight is updated if necessary. By following the differences between the compressed t'-core and the compressed t-core described above, we ensure that the algorithm is correct for this node-type.

For the running time, two aspects are relevant. First, we must verify that the chosen subtree of the core satisfies the property that v appears together with each of its neighbours in G_t in some bag. Since all neighbours of v in G_t are contained in $B(t)$ and thus in the restricted bag, this check can be performed in the required time using only the compressed core. Second, we must ensure that the weight changes can be computed efficiently. By vertex addition computability (R3) this can be done in time $2^{\mathcal{O}(|N(v)\cap X|)}$, where X denotes the non-restricted bag. Since all relevant neighbours are contained in the restricted bag, $|N(v) \cap X|$ is at most the size of the restricted bag. Therefore, all necessary computations can be carried out in time polynomial in the number of compressed cores.

Join-node. Let t be an join-node with child-nodes t_1 and t_2. Also let (T_t, B_t) be a partial tree-decomposition of G_t constructed by the inefficient-DP from a partial decomposition (T_{t_1}, B_{t_1}) of G_{t_1} and a partial decomposition (T_{t_2}, B_{t_2}) of G_{t_2}. We first analyse how these decompositions and their compressed cores differ. For join-nodes, the trees T_t, T_{t_1} and T_{t_2} have the same topology and each pair of corresponding bags in (T_{t_1}, B_{t_1}) and (T_{t_2}, B_{t_2}) has the same intersection of $B(t)$. For every node a of T_t, the bag $B_t(a)$ is given by the union of $B_{t_1}(a)$ and $B_{t_2}(a)$. For the compressed cores, both the restricted bags and the topology are identical. It therefore remains to analyse how the weights are affected. By union computability (R4), the weight of a node in the compressed core for t is equal to the sum its weights in the compressed cores for t_1 and t_2, minus a correction term that depends only on the restricted

bag. Since all entries of an integer sequence correspond to nodes with the same restricted bag, this correction term is identical for every element of the sequence. Subtracting the same value from all entries does not affect which elements appear in the corresponding typical sequence. Hence, it suffices to consider the element-wise sum of the sequences. By Lemma 8.21, each entry of the typical sequence of the join-node is obtained by adding two entries of the child typical sequences. Since typical sequences are compressed versions of longer integer sequences, multiple alignments of entries may be possible. In particular, entries may need to be duplicated to realize a consistent element-wise correspondence between the sequences. For a given alignment, the corresponding typical sequences of X_{t_1} and X_{t_2}, including any duplicated entries, are added element-wise. The correction term is then subtracted from each element and entries are removed whenever possible. The resulting sequence is the corresponding typical sequence in the compressed core of X_t.

Our computation of the compressed t-cores emulates these changes. We require that the child-cores have the same topology and corresponding nodes have identical restricted bags. We then construct the t-core by using the same topology and the same restricted bags as the child-cores. Moreover, we set the maximum weight to be the maximum of the maxima associated with the two child-cores and the weights of the bags in the core. To determine the sequence of weights at a node, we combine the two corresponding typical sequences from the child-cores and then subtract the correction term from each entry. Given two typical sequences $(a_1, \ldots, a_q)$ and $(b_1, \ldots, b_\ell)$, we compute all typical sequences that can be obtained by extending both sequences to equal length through the duplication of nodes and then adding them element-wise. For each resulting typical sequence, we construct a separate compressed core. To enumerate these sequences, we construct a directed auxiliary graph for the two input sequences. The graph contains a vertex $v_{i,j}$ for each index pair (i, j) with $1 \leq i \leq q$ and $1 \leq j \leq \ell$. We assign weight $a_i + b_j$ to vertex $v_{i,j}$. We add a directed edge from $v_{i,j}$ to a vertex $v_{i',j'} \neq v_{i,j}$ whenever $i' \in \{i, i+1\}$ and $j' \in \{j, j+1\}$. An example of this graph is shown in Figure 8.8. Observe, that each directed path from $v_{1,1}$ to $v_{q,\ell}$ in the auxiliary graph corresponds to one possible extension and element-wise summation of the sequences. For every such path, we compute the typical sequence of the weights encountered along the path. The set of all such typical sequences coincides exactly with the set of all typical sequences obtainable by extending and summing A and B. A formal justification of this correspondence is straightforward and can be found in Section 3.5 of the original source [AZ21]. Consequently, we can compute all relevant typical sequences by constructing the auxiliary graph, computing all directed paths from $v_{1,1}$ to $v_{q,\ell}$ and computing the typical sequence of each sequence of weights along a path. By following the differences between the compressed t'-core and the compressed t-core described above, we ensure that the algorithm is correct for this node-type.

For the running time we argue that all weights can be computed efficiently. By union computability (R3), the correction term can be computed in time $2^{\mathcal{O}(|X \cap Y|)}$, where X and Y denote the two non-restricted bags. Since corresponding bags of the two child-decompositions only intersect in vertices of the host-bag $B(t)$, this computation can be performed in time polynomial in the number of compressed cores. It therefore remains to show that computing all extensions and sums of typical sequences is polynomial in the number of compressed cores. First, observe that the auxiliary graph contains $q \cdot \ell$ vertices and can thus be constructed within the desired time bound. Next, we show that the number of directed paths from $v_{1,1}$ to $v_{q,\ell}$ is polynomial in the number of compressed cores. This follows from the fact that, each vertex of the auxiliary graph has at most three outgoing edges, namely those to vertices with neighbouring non-decreased indices, and every such path has length at most $q + \ell$. Hence,

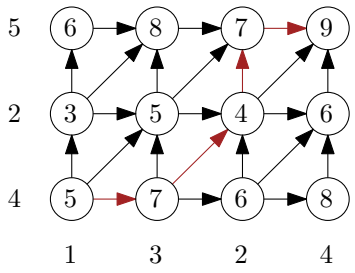


Figure 8.8: An example of the auxiliary graph constructed for join-nodes. Here, the sequences $A = (1, 3, 2, 4)$ and $B = (4, 2, 5)$ are combined. The path highlighted in red corresponds to the extensions $(1, 3, 2, 2, 4)$ of A and $(4, 4, 2, 5, 5)$ of B to the same length.

the number of such paths is bounded by $3^{q+\ell} \in 2^{\mathcal{O}(q+\ell)}$ which is exponential in $q + \ell$ but polynomial in the number of compressed cores. Finally, all such paths can be enumerated in time polynomial in their number, which completes the argument.

Analysis of the FPT-DP. It remains to analyse the resulting FPT-DP. For the running time, recall that the number of compressed cores per host-node and hence the number of partial solutions is bounded by $2^{\mathcal{O}(f(\ell)^3+k\cdot f(\ell)^2)}$, where $\ell \in \mathcal{O}(k)$ is the weight of the host-decomposition and f is the function introduced in the requirement boundedness (R5). Each transition can be computed in time polynomial in this number, yielding the same bound for the time required by the FPT-DP at a single host-node. Since a nice host-tree-decomposition of $f(\ell) \cdot |V(G)|$ nodes and the same weight can be computed in time $f(\ell)^2 \cdot |V(G)|$ from a given tree-decomposition (see Lemma 3.9 and boundedness (R5)), the total running time of the dynamic program is $2^{\mathcal{O}(f(\ell)^3+k\cdot f(\ell)^2)} \cdot |V(G)|$.

To reconstruct a representative tree-decomposition, we trace a compressed core C enumerated for the host-root back through the recursive computations of the FPT-DP, selecting, for each host-node, a compressed core that is compatible with C. We then apply Lemma 8.23. The algorithm of that lemma constructs a tree-decomposition realizing this compressed core in time $s(C)^{\mathcal{O}(1)} \cdot |V(G)|$, where s denotes the recursive size of a valid compressed core for the host-root. Since the size of a compressed core is polynomial in the number of compressed cores, this reconstruction step does not affect the overall running time.

We have therefore established Theorem 8.3 as stated above.

Theorem 8.3: *Let $G = (V, E)$ be a graph and let k be an integer. Let (T, B) be a given tree-decomposition of G with weight $\ell \in \mathcal{O}(k)$. Consider a parameter of $\mathcal{ECP}$ with weight function w. Then, there exists an algorithm that decides whether G has w-treewidth at most k in time $2^{\mathcal{O}(f(\ell)^3+k\cdot f(\ell)^2)} \cdot |V|$.*

8.3 Globally-partitioned treewidth is an efficiently computable parameter

In the previous section, we introduced a class of efficiently computable parameters. We now demonstrate that gp-treewidth belongs to this class. To this end, we first consider the setting in which a clique-partition of the input graph is given with the input and thus is fixed beforehand. In this scenario, we show that, for each graph, the corresponding variant of gp-treewidth is induced by a weight function in the sense defined earlier. We explicitly formalize the weight

functions on bags that induce these variants of gp-treewidth and show that they satisfy all required properties (R1) to (R5). Hence, in this restricted setting, the parameter belongs to the class $\mathcal{ECP}$ and can be computed in FPT-time by the general algorithm developed above. In a second step, we extend this result to the general case in which the clique-partition is not given as part of the input and must be computed as well.

8.3.1 Assuming that a clique-partition is provided with the input

We first show that, under the assumption that a clique-partition is fixed in advance, gp-treewidth belongs to the class $\mathcal{ECP}$. To this end, we verify that, for each graph with a fixed clique-partition, gp-treewidth is a tree-decomposition-based parameter induced by a weight function satisfying (R1) to (R5). By extension, this ensures that the fixed-partition variant of gp-treewidth is efficiently computable for all graphs.

We begin by formally introducing a weight function for each graph. Let $\mathcal{P}$ be a clique-partition of the input graph $G = (V, E)$. We define the weight function $w_{\mathcal{P}}$ to denote the weight under $\mathcal{P}$. Recall, that this was defined as usual by summing the logarithmic weights of cliques. As a consequence of this definition, gp-treewidth with respect to a fixed clique-partition $\mathcal{P}$ is a tree-decomposition-based parameter induced by the weight function $w_{\mathcal{P}}$. In particular, any algorithm for computing the fixed-partition variant of gp-treewidth only needs to compute a suitable tree-decomposition. Indeed, since $\mathcal{P}$ is fixed, both the clique-partition restricted to a bag and the weight of that bag are completely determined by the graph induced by this bag. Furthermore, note that the weight function depends on the clique-partition, hence for each graph and each clique-partition we can get a different weight function. We show that each such parameter is efficiently computable and hence the fixed-partition variant gp-treewidth as the generalization of those parameters also is.

Observe, that the weights defined by $w_{\mathcal{P}}$ are not necessarily an integer. Here, we employ the two different formulations of weight introduced in the preliminaries. Using the fact that the sum of logarithms equals the logarithm of a product, we obtain an equivalent definition in which the weight is given by the product of clique sizes (plus 1), without applying the logarithm. Hence, we may switch between these two formulations. In general, we use the logarithmic formulation. Whenever integer weights are required, we switch to the multiplicative formulation, at the cost of exponentiating the parameter. This alternative formulation is only needed when bounding the number of distinct values in an integer sequence. We consider the effects on the running time later on.

Therefore, it remains to verify that the weight function $w_{\mathcal{P}}$ satisfies the requirements (R1) to (R5) introduced in Section 8.1.

Lemma 8.25: *Let $G = (V, E)$ be a graph and let $\mathcal{P}$ be a clique-partition of G. Then, the weight function $w_{\mathcal{P}}$ satisfies the requirements (R1) to (R5).*

Proof. We verify the five requirements one by one. Recall, that, for a vertex set A, $\mathcal{P}[A]$ denotes the restriction of $\mathcal{P}$ to A.

(R1) *Subset monotonicity*: Let X be a vertex set and let $S \subseteq X$ be a subset of X. Then, $w_{\mathcal{P}}(S) \leq w_{\mathcal{P}}(X)$.

Observe, that for every clique C^* of $\mathcal{P}[S]$, there exists a clique of $\mathcal{P}[X]$ that is a superset of C^*. Since the logarithm is monotone, it follows that

$$\begin{aligned} w_{\mathcal{P}}(S) &= \sum_{C \in \mathcal{P}[S]} w_{\mathcal{P}}(C) \\ &\leq \sum_{C' \in \mathcal{P}[X]} w_{\mathcal{P}}(C') \\ &= w_{\mathcal{P}}(X). \end{aligned}$$

(R2) *Superset monotonicity*: Let X, Y be two vertex sets and let $v \notin X \cup Y$ be a vertex such that $N(v) \cap X = N(v) \cap Y$. If $w_{\mathcal{P}}(X) \leq w_{\mathcal{P}}(Y)$, then $w_{\mathcal{P}}(X + v) \leq w_{\mathcal{P}}(Y + v)$.

Let C^* be the clique of $\mathcal{P}$ containing v. Then, $w_{\mathcal{P}}(X+v) = w_{\mathcal{P}}(X)+w_{\mathcal{P}}(C^* \cap (X+v))-w_{\mathcal{P}}(C^* \cap X)$, since $\mathcal{P}[X]$ and $\mathcal{P}[X + v]$ differ only in the restriction of C^* and, by Observation 3.3, the weights of disjoint sets add under union. Analogously, we have $w_{\mathcal{P}}(Y + v) = w_{\mathcal{P}}(Y) + w_{\mathcal{P}}(C^* \cap (Y + v)) - w_{\mathcal{P}}(C^* \cap Y)$, since v belongs to a restriction of the same clique C^* in both $X + v$ and $Y + v$. Due to $N(v) \cap X = N(v) \cap Y$, the restrictions of C^* to X and to Y coincide. Consequently, we have $w_{\mathcal{P}}(C^* \cap X) = w_{\mathcal{P}}(C^* \cap Y)$ and $w_{\mathcal{P}}(C^* \cap (X + v)) = w_{\mathcal{P}}(C^* \cap (Y + v))$. Together with the assumption $w_{\mathcal{P}}(X) \leq w_{\mathcal{P}}(Y)$, this implies the claim.

(R3) *Vertex addition computability*: Let X be a vertex set and let $v \notin X$ be a vertex. Then, $w_{\mathcal{P}}(X + v) = w_{\mathcal{P}}(X) + g(N(v) \cap X)$ where $g(x)$ can be computed in time $2^{\mathcal{O}(|N(v) \cap X|)}$.

As observed above, it suffices to compute the weight difference $w_{\mathcal{P}}(C^* \cap (X+v))-w_{\mathcal{P}}(C^* \cap X)$. Since every vertex of C^* is adjacent to v, this term is a function of $N(v) \cap X$. Because the clique-partition $\mathcal{P}$ is fixed, the value can be computed in the required time.

(R4) *Union computability*: Let X, Y be two vertex sets such that no vertex of $X - Y$ is adjacent to a vertex of $Y - X$. Then, $w_{\mathcal{P}}(X \cup Y) = w_{\mathcal{P}}(X) + w_{\mathcal{P}}(Y) - h(X \cap Y)$ where $h(x)$ can be computed in time $2^{\mathcal{O}(|X \cap Y|)}$.

Consider $w_{\mathcal{P}}(X \cup Y)$ and compare it clique-by-clique with $w_{\mathcal{P}}(X) + w_{\mathcal{P}}(Y)$ to determine the necessary correction term. To this end, note that for a vertex set Z we have $w_{\mathcal{P}}(Z) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap Z)$. Let C^* be a clique of $\mathcal{P}[X \cup Y]$. We distinguish the cases $C^* \subseteq X - Y, C^* \subseteq Y - X$, and $C^* \cap (X \cap Y) \neq \emptyset$. If $C^* \subseteq X - Y$, then $w_{\mathcal{P}}(C^* \cap X) = w_{\mathcal{P}}(C^* \cap (X \cup Y))$ and this contribution appears exactly once in both $w_{\mathcal{P}}(X \cup Y) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap (X \cup Y))$ and $w_{\mathcal{P}}(X) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap X)$. However, no contribution for C^* occurs in $w_{\mathcal{P}}(Y) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap Y)$ and hence no correction term is needed. The case $C^* \subseteq Y - X$ is analogous. Now suppose that $C^* \cap (X \cap Y) \neq \emptyset$. Then, restrictions of C^* to the respective sets contribute to both $w_{\mathcal{P}}(X) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap X)$ and $w_{\mathcal{P}}(Y) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap Y)$. Since no vertex of $X - Y$ is adjacent to a vertex of $Y - X$, either $C^* \subseteq X$ or $C^* \subseteq Y$ must hold. W.l.o.g. assume $C^* \subseteq X$. In this case, we have $w_{\mathcal{P}}(C^* \cap (X \cup Y)) = w_{\mathcal{P}}(C^* \cap X)$. This is exactly the contribution of C^* to $w_{\mathcal{P}}(X) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap X)$. Hence, the contribution of C^* arising from $w_{\mathcal{P}}(Y) = \sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap Y)$ has to be subtracted from the sum of the individual weights. Due to $C^* \cap (Y - X) = \emptyset$, this is $w_{\mathcal{P}}(C^* \cap Y) = w_{\mathcal{P}}(C^* \cap (X \cap Y))$. Repeating this correction for every such clique yields exactly $\sum_{C \in \mathcal{P}} w_{\mathcal{P}}(C \cap (X \cap Y)) = w_{\mathcal{P}}(X \cap Y)$ as the necessary correction term. Since the clique-partition $\mathcal{P}$ is fixed, $w_{\mathcal{P}}(X \cap Y)$ is a function of $X \cap Y$ and can be computed in the required time.

(R5) *Boundedness*: There exists a computable function f such that $|X| \leq f(w_{\mathcal{P}}(X))$ for all vertex sets X.

By Lemma 4.10 the size of X is bounded by $2^{w_{\mathcal{P}}(X)}$. Thus, we may choose $f(x) = 2^x$.

Hence, all requirements (R1) to (R5) are satisfied. ∎

Therefore, gp-treewidth with a fixed clique-partition belongs to the class $\mathcal{ECP}$. Consequently, by Theorem 8.3, we obtain that gp-treewidth with respect to a fixed clique-partition can be computed in FPT-time parameterized by its value.

Theorem 8.26: *Let $G = (V, E)$ be a graph, let $\mathcal{P}$ be a clique-partition of G and let k be a real number. Let (T, B) be a given tree-decomposition of G with weight $\mathcal{O}(k)$. Then, there exists an algorithm that decides whether G has gp-treewidth at most k under the fixed clique-partition $\mathcal{P}$ in time $2^{2^{\mathcal{O}(k)}} \cdot |V|$.*

Proof. By Theorem 8.3 the running time of the algorithm is at most $2^{\mathcal{O}(f(\ell)^3 + (2^k+1)\cdot f(\ell)^2)} \cdot |V|$, where f is the function introduced by the requirement boundedness (R5) and $\ell \in \mathcal{O}(k)$ is the weight of the host-decomposition. We therefore choose $f(x) = 2^x$ as done in our proof of (R5) for this variant in Lemma 8.25. Thus, we obtain a running time of

$$\begin{aligned} &2^{\mathcal{O}\left(\left(2^{\mathcal{O}(k)}\right)^3 + \left(2^k+1\right)\cdot\left(2^{\mathcal{O}(k)}\right)^2\right)} \cdot |V| \\ &= 2^{2^{3\mathcal{O}(k)} + \left(2^k+1\right)\cdot 2^{2\mathcal{O}(k)}} \cdot |V| \\ &\in 2^{2^{\mathcal{O}(k)}} \cdot |V|. \end{aligned}$$

Note that, in order to bound the number of distinct weight values, we used the multiplicative formulation of the weight. This incurs a factor of $2^k + 1$ instead of k in the analysis of the generalized algorithm. ∎

So far, we have assumed that we are given a tree-decomposition of linear weight. Similar to Theorem 5.9 we can use iterative compression to obtain such a decomposition at the cost of a factor of $|V|$.

Corollary 8.27: *Let $G = (V, E)$ be a graph, let $\mathcal{P}$ be a clique-partition of G and let k be real number. Then, there exists an algorithm that decides whether G has gp-treewidth at most k under the fixed clique-partition $\mathcal{P}$ in time $2^{2^{\mathcal{O}(k)}} \cdot |V|^2$.*

8.3.2 The assumption of a given clique-partition can be satisfied

In the last subsection, we assumed that a clique-partition is fixed in advance when computing the tree-decomposition. We now satisfy this assumption by computing the clique-partition simultaneously within the dynamic program. This extension is based on the observation that, in the algorithm for computing optimal tree-decompositions, only the vertices of the current host-bag of the given host-tree-decomposition are stored explicitly. Hence, in order to evaluate the weight of any subset of these vertices, it suffices to know the restriction of the global clique-partition to the current host-bag. In particular, we may work with individual clique-partitions for each host-bag, provided that these local partitions are consistent and together form a full clique-partition of the graph. By enumerating all such collections of partial partitions and applying the previous algorithm to each of them, we can compute the gp-treewidth.

We first describe an algorithm that enumerates all collections of partial clique-partitions such that the clique-partitions of each collection are restrictions of the same global clique-partition of G. As before, we employ a dynamic programming approach on a given host-tree-decomposition (T, B). A partial solution at a node $t \in V(T)$ consists of the restriction of a clique-partition of the graph G_t visited so far to the bag $B(t)$. The recursion proceeds as follows.

For a **leave-node**, we use the empty partition.

For a **forget-node** with forgotten vertex v, we remove v from the restricted partition.

For an **introduce-node** t with child t' and introduced vertex v, we generate, from each partial solution at t', all partitions obtained by inserting v into one of the existing cliques of the partition of $B(t')$, if possible, or alternatively by placing v into a new clique.

For a **join-node** t with children t_1 and t_2, we pair partial solutions that have the same restricted partition and keep this common partition.

At each step, this procedure computes precisely the set of all clique-partitions of the current host-bag that can be extended to a clique-partition of the subgraph processed so far. Let the weight of the host-decomposition be $\mathcal{O}(k)$. Observe, that each clique-partition of a set X can be seen as a function assigning each of the $|X|$ vertices to one of the at most $|X|$ cliques. There are at most $|X|^{|X|}$ such functions and hence at most $|X|^{|X|}$ clique-partitions of X. Due to this fact and since each host-bag has size at most $2^{\mathcal{O}(k)}$ (see Lemma 4.10), we store at most $2^{\mathcal{O}(k)\cdot 2^{\mathcal{O}(k)}} = 2^{2^{\mathcal{O}(k)}}$ partial solutions per node.

We now combine this enumeration of partial clique-partitions with the dynamic program for computing optimal tree-decompositions. At each node of the host-decomposition, we first compute the collection of all clique-partitions that are a restriction of a clique-partition of G_t to the current host-bag and then, for each such partition, enumerate all compatible compressed cores of partial tree-decompositions of the vertices processed so far. Formally, a partial solution consists of a pair comprising a restricted clique-partition and a compressed core. Informally, a partial solution represents a partial solution of the FPT-DP together with the restriction of some clique-partition of the graph visited so far to the current host-bag. Observe, that along any fixed branch of the dynamic program, once a vertex is assigned to a clique, this assignment never changes. Hence, all clique-partitions of this branch are a restriction of the same clique-partition of G, and the combined algorithm is correct. The number of partial solutions is given by the product of the number of partial clique-partitions and the number of compressed cores. This results in a running time of

$$\begin{aligned} & 2^{2^{\mathcal{O}(k)}} \cdot 2^{2^{\mathcal{O}(k)}} \cdot |V|^2 \\ &= 2^{2^{\mathcal{O}(k)} + 2^{\mathcal{O}(k)}} \cdot |V|^2 \\ &= 2^{2^{\mathcal{O}(k)}} \cdot |V|^2. \end{aligned}$$

We thus have shown the following theorem.

Theorem 8.28: *Let $G = (V, E)$ be a graph and let k be a real number. Then, there exists an algorithm that decides whether G has gp-treewidth at most k in time $2^{2^{\mathcal{O}(k)}} \cdot |V|^2$.*

8.4 Some closing remarks

We conclude with some remarks. First, as mentioned earlier, we adapted the original algorithm for enumerating tree-decompositions to gp-treewidth and not to cp-treewidth. In particular, we relied on a global clique-partition rather than allowing an individual clique-partition for

Figure 8.9: An example for which cp-treewidth violates superset monotonicity (R2). Part (a) depicts the vertex set X (black) together with the vertex v (purple). An optimal clique-partition of X and $X + v$ is shown through the colouring. Part (b) depicts the vertex set Y (black) together with the vertex v (purple). An optimal clique-partition of Y and $Y + v$ is shown through the colouring.

each bag. This is necessary because cp-treewidth violates superset monotonicity (R2). While it might still be possible to adapt the algorithm by different means, our approach fundamentally fails in this setting. To see that cp-treewidth violates (R2), consider the following example visualized in Figure 8.9. Let X be a clique of size 4 and let Y be a path of length 3. Assume that X and Y intersect in a single vertex w, which is an endpoint of the path. Using the multiplicative formulation of the weight, the set X has weight 5, whereas Y has weight $2 \cdot 3 = 6$. Now, add a vertex v to both sets such that v is adjacent only to the shared vertex w. Then, $X + v$ has weight $5 \cdot 2 = 10$, while $Y + v$ has weight $3 \cdot 3 = 9$. Hence, adding v reverses the inequality between the weights, and (R2) is violated.

The underlying reason is that the two local clique-partitions differ on whether the shared neighbour w of v is placed in the same clique as v. Since, in the cp-treewidth setting, each set may choose its clique-partition independently, the vertex w may belong to different cliques in X and Y.

Second, recall that we proved in Theorem 5.13 that no XP-algorithm for Tree-clique-cover number can exist, unless $\mathcal{P} = \mathcal{NP}$. In particular, this result implies that the tree-clique-cover number does not belong to the class $\mathcal{ECP}$ of efficiently computable parameters, unless $\mathcal{P} = \mathcal{NP}$. This can also be seen by observing that the tree-clique-cover number violates the requirement of superset monotonicity (R2). To see this, consider two adjacent vertices u and w. Let $X = \{u, w\}$ and $Y = \{u\}$. Then, both X and Y can be covered with one clique. However, if we add a vertex v adjacent only to u, then the set $X + v$ requires two cliques in any cover, while $Y + v$ still admits a clique-cover of size one.

Furthermore, boundedness (R5) is violated as witnessed by Lemma 4.15. Recall, that requirement (R5) was used only in the analysis of the running time. Hence, instead of relying on (R5), one could bound the running time in terms of the width of the host-decomposition and design an FPT-algorithm parameterized by treewidth rather than by the tree-clique-cover number. Note, that this approach resolves only the violation of requirement (R5), but not the violation of requirement (R2).

9 Conclusion

In this work, we have presented three different approaches for computing parameters based on partitioning the bags of a tree-decomposition into cliques. First, we showed how to compute a $4k + 3$-approximation of cp-treewidth in FPT-time. If an exact algorithm is required, the XP-algorithm presented in this thesis can be used instead. Furthermore, we developed an FPT-algorithm for the larger parameter gp-treewidth by adapting the traditional algorithm of Bodlaender and Kloks [BK96] for computing treewidth. In addition to algorithmic results, we established lower bounds for both cp-Treewidth and gp-Treewidth via a reduction from Pathwidth. This $\mathcal{NP}$-hardness result helps in placing gp-Treewidth within the landscape of complexity classes. In particular, we have shown that gp-Treewidth belongs to the classes FPT and $\mathcal{NP}$, but not to $\mathcal{P}$ unless $\mathcal{P} = \mathcal{NP}$.

Since cp-treewidth and gp-treewidth are closely related, they can be used almost interchangeably in algorithmic applications. Thus, an user may choose between approximating cp-treewidth and computing gp-treewidth exactly depending on the given input. In particular, all currently known algorithms for problems parameterized by one of these parameters can be transferred to the other without increasing the parameter dependence in the running time. Such a transfer fails only when optimality with respect to one parameter is required by the application or when the additional property of gp-treewidth, namely that all local partitions extend to a global partition of the input graph, is explicitly used. To illustrate this transfer, consider two examples. The Independent Set problem can be solved in time $2^{\mathcal{O}(k)} \cdot |V|^{\mathcal{O}(1)}$ when parameterized by cp-treewidth [BKW23 | Gei23]. Applying the same dynamic program to a given gp-tree-decomposition instead of a given cp-tree-decomposition yields an algorithm with the same running time for Independent Set now parameterized by gp-treewidth. As a second example, consider the computation of gp-treewidth using the algorithm of Theorem 8.28. The above mentioned transfer yields an algorithm for gp-Treewidth with running time $2^{2^{\mathcal{O}(k)}} \cdot |V|^2$ now parameterized by cp-treewidth. Note that this does not yield an algorithm for cp-Treewidth, as the computed object remains gp-treewidth. Moreover, the algorithm relies only on the global consistency of constructed decompositions, not of the given input decomposition.

From a broader perspective, we considered how cp-treewidth and gp-treewidth relate to other variants of treewidth with respect to the three key aspects *size*, *usefulness* and *computability* of a parameter. An overview is given in Figure 9.1 and Table 9.1. Both parameters are smaller than treewidth, though larger than the tree-clique-cover number and the tree-independence number. However, cp-treewidth and gp-treewidth retain the full algorithmic applicability of treewidth, unlike some smaller parameters such as the tree-clique-cover number and the tree-independence number. In terms of computability, as summarized above, some basic FPT-time approximation results for treewidth carry over to cp-treewidth, while certain exact FPT-algorithms translate to gp-treewidth. In contrast, the tree-independence number appears to require at least XP-time for comparable computations. Altogether, the two clique-partition-based parameters provide an improvement in terms of parameter size while incurring only limited losses with respect to usefulness and computability when compared

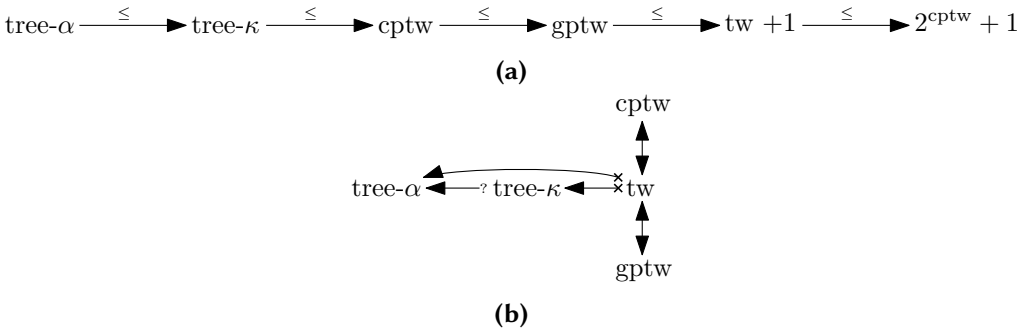


Figure 9.1: This figure visualises two hierarchies formed by the parameters. Part (a) shows the size hierarchy of Theorem 4.1, while part (b) illustrates the hierarchy of algorithmic potential introduced in Section 4.2. In the size-hierarchy, an arrow indicates that one parameter is smaller than another with respect to decomposition-wise comparisons. In the hierarchy of algorithmic potential, arrows represent FPT-reductions from one parameter to another. An $\times$ or ? indicates that the reverse direction is ruled out (assuming $\mathcal{P} \neq \mathcal{NP}$) or unknown, respectively. Note, that parameterized hardness results propagate along these arrows, while algorithms transfer in the opposite direction.

to treewidth. In comparison to the tree-independence number, they admit more efficient algorithms for their computation and support a broader range of fixed-parameter tractable algorithms parameterized by them, at the cost of a larger parameter value.

Future work and open problems. Despite these advances, several open questions remain. Most notably, there still is a gap between the best known algorithms and lower bounds. In particular, it is open whether the cp-treewidth can be computed in FPT-time, and thus whether the algorithm for gp-Treewidth can be adapted to this setting using some modification of our approach. Apart from the containment in some complexity class, it is natural to ask whether single-exponential algorithms exist for these parameters and what the optimal bases and exponents are. Similarly, the approximability of these parameters remains unclear. Current lower bounds do not exclude polynomial-time constant-factor approximation algorithms. For treewidth, it is known that such approximations are unlikely under the Small Set Expansion (SSE) conjecture [WAPL14], while without SSE only hardness results with factors close to 1 are known [Bon25]. It remains an open question whether these results can be adapted to our parameters and whether alternative, easier to adapt, approaches for treewidth can be developed. For gp-Treewidth in particular, developing approximation algorithms remains an open challenge, especially since such algorithms would need to outperform both the approximation algorithm for cp-Treewidth and the exact algorithm for gp-Treewidth.

More broadly, the framework of cp-treewidth discovered so far provides a solid foundation which can now be used for further research. In particular, it would be interesting to develop more efficient algorithms for a wider range of problems, and to investigate whether cp-treewidth can be used to improve algorithms for problems already known to be in $\mathcal{P}$. In their paper, Abboud et al. [AVW16] showed that treewidth can be used to design more efficient algorithms for computing the diameter of graphs. It would be interesting to investigate whether cp-treewidth can be leveraged to further improve this and other similar algorithms.

It thus remains to explore how far cp-treewidth can take us.

Table 9.1: This table compares upper and lower bounds for computing different parameters that refine treewidth. We consider three types of results: Exact algorithms, constant-factor approximation algorithms, and lower bounds for the exact computation in the form of coarse-grained hardness results. In addition to the results presented in this thesis (highlighted in red), which are currently the only known ones for these parameters, we include selected results for treewidth and the tree-independence number. For each parameter, we report the most efficient known algorithm, preferring better dependence on the parameter k over dependence on the input size n and using the approximation ratio as a final tie-breaker.
For the tree-independence number and the tree-clique-cover number, no exact algorithms are known beyond the naive enumeration of all tree-decompositions and all independent sets or clique-covers. Note, that for the parameter of $\mathcal{P}$-flattened treewidth, only an algorithm for some intersection graphs is known (Theorem 12 in [Ber+18]). We therefore do not include this parameter in our comparison, even though it is closely related to the others.

Parameter	Exact	Approximation		Hardness
		Time	Ratio	
treewidth	$2^{\mathcal{O}(k^3)} \cdot n$ [BK96 | AZ21]	$2^{\mathcal{O}(k)} \cdot n$ [Kor21 | Bod24]	$2k+1$	$\mathcal{NP}$-hard [Bon25]
clique-partitioned treewidth	$2^{2^{\mathcal{O}(k)}} \cdot n^{2^k+3}$ [Theorem 5.3]	$2^{2^{\mathcal{O}(k)}} \cdot n^3$ [Theorem 5.9]	$4k+3$	$\mathcal{NP}$-hard [Theorem 7.25]
globally-partitioned treewidth	$2^{2^{\mathcal{O}(k)}} \cdot n^2$ [Theorem 8.28]	-		$\mathcal{NP}$-hard [Theorem 7.28]
tree-independence number	$2^{n^{\mathcal{O}(1)}}$	$2^{\mathcal{O}(k^2)} \cdot n^{\mathcal{O}(k)}$ [Dal+26]	$8k$	no XP-time unless $\mathcal{P} = \mathcal{NP}$ [Dal+26]
tree-clique-cover number	$2^{n^{\mathcal{O}(1)}}$	-		no XP-time unless $\mathcal{P} = \mathcal{NP}$ [Theorem 5.13]

Bibliography

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. "Complexity of Finding Embeddings in a k-Tree". In: *SIAM Journal on Algebraic Discrete Methods* Volume 8 (1987), pp. 277–284. eprint: *https://doi.org/10.1137/0608024*.

[Aro21] Chris Aronis. "The Algorithmic Complexity of Tree-Clique Width". In: *CoRR* Volume abs/2111.02200 (2021). arXiv: *2111.02200*.

[AVW16] Amir Abboud, Virginia Vassilevska Williams, and Joshua R. Wang. "Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter in Sparse Graphs". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. Edited by Robert Krauthgamer. SIAM, 2016, pp. 377–391. DOI: *10.1137/1.9781611974331.CH28*.

[AZ21] Ernst Althaus and Sarah Ziegler. "Optimal Tree Decompositions Revisited: A Simpler Linear-Time FPT Algorithm". In: *Graphs and Combinatorial Optimization: from Theory to Applications: CTW2020 Proceedings*. Edited by Claudio Gentile, Giuseppe Stecca, and Paolo Ventura. Cham: Springer International Publishing, 2021, pp. 67–78. ISBN: 978-3-030-63072-0. DOI: *10.1007/978-3-030-63072-0_6*.

[BB73] Umberto Bertelè and Francesco Brioschi. "On Non-serial Dynamic Programming". In: *J. Comb. Theory A* Volume 14 (1973), pp. 137–148. DOI: *10.1016/0097-3165(73)90016-2*.

[BB86] Alan A. Bertossi and Maurizio A. Bonuccelli. "Hamiltonian Circuits in Interval Graph Generalizations". In: *Inf. Process. Lett.* Volume 23 (1986), pp. 195–200. DOI: *10.1016/0020-0190(86)90135-3*.

[BCKN15] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. "Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth". In: *Inf. Comput.* Volume 243 (2015), pp. 86–111. DOI: *10.1016/J.IC.2014.12.008*.

[Ber+18] Mark de Berg, Hans L. Bodlaender, Sándor Kisfaludi-Bak, Dániel Marx, and Tom C. van der Zanden. "A framework for ETH-tight algorithms and lower bounds in geometric intersection graphs". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*. Edited by Ilias Diakonikolas, David Kempe, and Monika Henzinger. ACM, 2018, pp. 574–586. DOI: *10.1145/3188745.3188854*.

[BGKH91] Hans L. Bodlaender, John R. Gilbert, Ton Kloks, and Hjálmtyr Hafsteinsson. "Approximating Treewidth, Pathwidth, and Minimum Elimination Tree Height". In: *17th International Workshop, WG '91, Fischbachau, Germany, June 17-19, 1991, Proceedings*. Edited by Gunther Schmidt and Rudolf Berghammer. Springer, 1991, pp. 1–12. DOI: *10.1007/3-540-55121-2_1*.

[BJ00] Hans L. Bodlaender and Klaus Jansen. "On the Complexity of the Maximum Cut Problem". In: *Nord. J. Comput.* Volume 7 (2000), pp. 14–31.

[BJ82] Kellogg S. Booth and J. Howard Johnson. "Dominating Sets in Chordal Graphs". In: *SIAM J. Comput.* Volume 11 (1982), pp. 191–199. DOI: *10.1137/0211015*.

[BK96] Hans L. Bodlaender and Ton Kloks. "Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs". In: *J. Algorithms* Volume 21 (1996), pp. 358–402. DOI: *10.1006/JAGM.1996.0049*.

[BKW23] Thomas Bläsius, Maximilian Katzmann, and Marcus Wilhelm. "Partitioning the Bags of a Tree Decomposition into Cliques". In: *21st International Symposium on Experimental Algorithms, SEA 2023, Barcelona, Spain, July 24-26, 2023*. Edited by Loukas Georgiadis. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 3:1–3:19. DOI: *10.4230/LIPICS.SEA.2023.3*.

[Bod+23] Hans L. Bodlaender, Édouard Bonnet, Lars Jaffke, Dusan Knop, Paloma T. Lima, Martin Milanic, Sebastian Ordyniak, Sukanya Pandey, and Ondrej Suchý. "Treewidth is NP-Complete on Cubic Graphs (and related results)". In: *CoRR* Volume abs/2301.10031 (2023). arXiv: *2301.10031*.

[Bod24] Hans L. Bodlaender. "Approximation Algorithms for Treewidth, Pathwidth, and Treedepth - A Short Survey". In: *Graph-Theoretic Concepts in Computer Science - 50th International Workshop, WG 2024, Gozd Martuljek, Slovenia, June 19-21, 2024, Revised Selected Papers*. Edited by Daniel Král and Martin Milanic. Springer, 2024, pp. 3–18. DOI: *10.1007/978-3-031-75409-8*.

[Bod98] Hans L. Bodlaender. "A Partial k-Arboretum of Graphs with Bounded Treewidth". In: *Theor. Comput. Sci.* Volume 209 (1998), pp. 1–45. DOI: *10.1016/S0304-3975(97)00228-4*.

[Bon25] Édouard Bonnet. "Treewidth Inapproximability and Tight ETH Lower Bound". In: *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*. Edited by Michal Koucký and Nikhil Bansal. ACM, 2025, pp. 2130–2135. DOI: *10.1145/3717823.3718117*.

[CCJ90] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. "Unit disk graphs". In: *Discret. Math.* Volume 86 (1990), pp. 165–177. DOI: *10.1016/0012-365X(90)90358-O*.

[CLTV17] Mathieu Chapelle, Mathieu Liedloff, Ioan Todinca, and Yngve Villanger. "Treewidth and Pathwidth parameterized by the vertex cover number". In: *Discret. Appl. Math.* Volume 216 (2017), pp. 114–129. DOI: *10.1016/J.DAM.2014.12.012*.

[Cyg+15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. ISBN: 978-3-319-21274-6. DOI: *10.1007/978-3-319-21275-3*.

[Dal+26] Clément Dallard, Fedor V. Fomin, Petr A. Golovach, Tuukka Korhonen, and Martin Milanic. "Computing Tree Decompositions with Small Independence Number". In: *ACM Trans. Algorithms* Volume 22 (2026), 12:1–12:25. DOI: *10.1145/3767730*.

[DMS24] Clément Dallard, Martin Milanic, and Kenny Storgel. "Treewidth versus clique number. II. Tree-independence number". In: *J. Comb. Theory B* Volume 164 (2024), pp. 404–442. DOI: *10.1016/J.JCTB.2023.10.006*.

[Dvo15] Zdeněk Dvořák. "Cographs; chordal graphs and tree decompositions". 2015. URL: *https://iuuk.mff.cuni.cz/~rakdver/kgiii/lesson14-2.pdf*.

[Erd59] P. Erdös. “Graph Theory and Probability”. In: *Canadian Journal of Mathematics* Volume 11 (1959), pp. 34–38. DOI: *10.4153/CJM-1959-003-9*.

[Gei23] Sven Geißler. “Exploring Clique-Partitioned Treewidth”. Karlsruher Institut für Technologie, 2023.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[Gol04] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. NLD: North-Holland Publishing Co., 2004. ISBN: 0444515305.

[Hal76] Rudolf Halin. “S-functions for graphs”. In: *Journal of Geometry* Volume 8 (1976), pp. 171–186.

[HMV26] Claire Hilaire, Martin Milanič, and Đorđe Vasić. “Treewidth Versus Clique Number. V. Further Connections With Tree-Independence Number”. In: *Journal of Graph Theory* Volume n/a (2026). eprint: *https://onlinelibrary.wiley.com/doi/pdf/10.1002/jgt.70036*.

[IPS82] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. “Hamilton Paths in Grid Graphs”. In: *SIAM J. Comput.* Volume 11 (1982), pp. 676–686. DOI: *10.1137/0211056*.

[Kar72] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Edited by Raymond E. Miller and James W. Thatcher. Plenum Press, New York, 1972, pp. 85–103. DOI: *10.1007/978-1-4684-2001-2_9*.

[Kor21] Tuukka Korhonen. “A Single-Exponential Time 2-Approximation Algorithm for Treewidth”. In: *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 2021, pp. 184–192. DOI: *10.1109/FOCS52979.2021.00026*.

[Lam+19] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. “Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs”. In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. Edited by Stephen G. Kobourov and Henning Meyerhenke. SIAM, 2019, pp. 144–158. DOI: *10.1137/1.9781611975499.12*.

[LK75] J. K. Lenstra and A. H. G. Rinnooy Kan. “Some Simple Applications of the Travelling Salesman Problem”. In: *Journal of the Operational Research Society* Volume 26 (1975), pp. 717–733. eprint: *https://doi.org/10.1057/jors.1975.151*.

[RS86] Neil Robertson and Paul D. Seymour. “Graph Minors. II. Algorithmic Aspects of Tree-Width”. In: *J. Algorithms* Volume 7 (1986), pp. 309–322. DOI: *10.1016/0196-6774(86)90023-4*.

[RS95] Neil Robertson and Paul D. Seymour. “Graph Minors .XIII. The Disjoint Paths Problem”. In: *J. Comb. Theory B* Volume 63 (1995), pp. 65–110. DOI: *10.1006/JCTB.1995.1006*.

[WAPL14] Yu (Ledell) Wu, Per Austrin, Toniann Pitassi, and David Liu. "Inapproximability of Treewidth and Related Problems". In: *J. Artif. Intell. Res.* Volume 49 (2014), pp. 569–600. DOI: *10.1613/JAIR.4030*.

[Yol18] Nikola Yolov. "Minor-matching hypertree width". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Edited by Artur Czumaj. SIAM, 2018, pp. 219–233. DOI: *10.1137/1.9781611975031.16*.

[Zuc07] David Zuckerman. "Linear Degree Extractors and the Inapproximability of Max Clique and Chromatic Number". In: *Theory Comput.* Volume 3 (2007), pp. 103–128. DOI: *10.4086/TOC.2007.V003A006*.