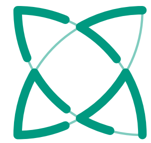
Karlsruhe Institute of Technology



Exploring Clique-Partitioned Treewidth

Bachelor's Thesis of

Sven Geißler

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

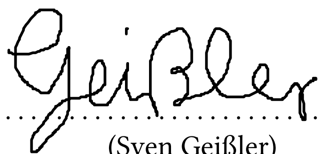
Reviewer: T.T.-Prof. Dr. Thomas Bläsius
Second reviewer: PD Dr. Torsten Ueckerdt
Advisors: Marcus Wilhelm

18th May 2023 – 18th September 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie.*

Karlsruhe, 18th September 2023



. . . .

(Sven Geißler)

Abstract

Clique-partitioned treewidth is a parameter for graphs that is similar to treewidth but additionally considers the structure of each bag. This is achieved by partitioning the vertices of each bag into cliques and assigning each bag a weight based on the logarithmic sizes of its cliques.

In this thesis, we use this parameter to design parametrized algorithms for many $\mathcal{NP}$-complete problems and give a general framework for such algorithms. For designing those algorithms, we adapt already known normalizations of tree decompositions to clique-partitioned tree decompositions and introduce a new type of normalization called smooth clique-partitioned tree decomposition. Furthermore, we prove that FPT-algorithms for problems parametrized by clique-partitioned treewidth can be found if and only if such algorithms can be found for treewidth. Building on this, we prove for a large number of problems that giving algorithms with clique-partitioned treewidth that are substantially faster than their direct treewidth-counterparts is impossible under the Exponential-Time Hypothesis. Here, we prove that those problems admit single exponential algorithms when parametrizing by treewidth but only double exponential algorithms for clique-partitioned treewidth. We also show that introducing an additional parameter, like the newly defined clique-degree, can allow us to describe parametrized algorithms for those problems. To better understand the parameter, we compare it to related parameters and study its behaviour on some graph families.

Zusammenfassung

Die cliquen-partitionierte Baumweite ist ein Parameter für Graphen, der der Baumweite ähnelt, aber zusätzlich die Struktur jedes Bags betrachtet. Dies wird erreicht, indem die Knoten eines jeden Bags in Cliquen partitioniert werden und jedem Bag ein Gewicht basierend auf den logarithmischen Größen seiner Cliquen zu gewiesen wird.

In dieser Thesis nutzen wir den Parameter, um parametrisierte Algorithmen für viele $\mathcal{NP}$-vollständige Probleme zu entwerfen und geben ein generelles Framework für solche Algorithmen an. Um diese Algorithmen zu entwerfen, passen wir bereits bekannte Normalisierungen von Baumzerlegungen auf cliquen-partitionierte Baumzerlegungen an und führen eine neue Art der Normalisierung, welche wir geglättete cliquen-partitionierte Baumzerlegung nennen, ein. Des Weiteren beweisen wir, dass FPT-Algorithmen für Probleme parametrisiert durch die clique-partitionierte Baumweite genau dann gefunden werden können, wenn solche Algorithmen für Baumweite gefunden werden können. Darauf aufbauend beweisen wir für eine große Anzahl an Problemen, dass es unter Annahme der Exponetial-Time Hypothese unmöglich ist einen Algorithmus mit cliquen-partitionierter Baumweite anzugeben, der substantiell schneller ist als sein direkter Baumweiten-Gegenüber. Hierfür beweisen wir, dass es für diese Probleme einen einfach exponentiellen Algorithmus für Baumweite, aber nur doppelt exponentielle Algorithmen für cliquen-partitionierte Baumweite gibt. Außerdem zeigen wir, dass die Einführung eines zusätzlichen Parameters, wie des neu definierten Cliquen-Grades, es uns erlaubt parametrisierte Algorithmen für diese Probleme anzugeben. Um den Parameter besser zu verstehen, vergleichen wir diesen mit verwandten Parametern und untersuchen sein Verhalten für einige Graphfamilien.

Contents

1. Introduction **1**
1.1. Related Work . . . 2
1.1.1. Computation of clique-partitioned treewidth . . . 3
1.1.2. Related parameters . . . 3
1.2. Outline . . . 4

2. Preliminaries **7**
2.1. Notation . . . 7
2.1.1. General mathematical notation . . . 7
2.1.2. Graphs . . . 7
2.1.3. Further notation . . . 8
2.2. The rank-based-approach . . . 9
2.3. Exponential-Time Hypothesis and Strong Exponential Time Hypothesis . . . 11
2.4. Defining treewidth, BBKMZ-treewidth and clique-partitioned treewidth . . . 11

3. Comparison with other parameters **15**
3.1. Treewidth . . . 15
3.1.1. Bounds . . . 15
3.1.2. Sharpness of bounds . . . 17
3.2. Number of cliques . . . 18
3.2.1. Bounds . . . 18
3.2.2. Sharpness of bounds . . . 19
3.3. BBKMZ-treewidth . . . 19
3.3.1. Bounds . . . 19
3.3.2. Sharpness of bounds . . . 20
3.4. Tree-clique width . . . 20
3.4.1. Bounds . . . 21
3.4.2. Sharpness of bounds . . . 21
3.5. Tree-independence number . . . 23
3.5.1. Bounds . . . 23
3.5.2. Sharpness of bounds . . . 24

4. Algorithmic potential and limitations **25**
4.1. A Criterion for the existence of FPT-algorithms parametrized by clique-partitioned treewidth . . . 25
4.2. A first general lower bound using the Exponential-Time Hypothesis . . . 26
4.3. Separating treewidth and clique-partitioned treewidth with 3-Clique Cover . 30

5. Building algorithms: First Approach **37**
5.1. Nice cp-tree-decompositions . . . 37
5.2. Extended cp-tree-decompositions . . . 39
5.3. Independent Set and Vertex Cover . . . 41

5.4. Maximum Induced Forest and Feedback Vertex Set 44

6. The limits of this approach 47

6.1. Finding problems where clique-partitioned treewidth is weaker than treewidth with the use of chordal graphs 47

6.2. Finding problems where clique-partitioned treewidth is weaker than treewidth with the use of co-planar graphs 50

7. Building algorithms with an additional parameter 55

7.1. Attributes of clique-partitioned tree decompositions 55

7.1.1. Star-graphs 57

7.1.2. Normalizations and additional parameters 58

7.2. Dominating Set 59

7.3. Steiner Tree 64

7.4. Clique 66

7.5. Lower bounds for the number of vertices picked in each clique 66

8. Building algorithms by tracing cliques 69

8.1. Smooth cp-tree-decompositions 69

8.2. Hamilton Cycle and Hamilton Path 73

8.3. A framework using clique-partitioned treewidth 80

8.4. Applying the framework 84

8.4.1. Vertex Adjacent Feedback Edge Set 84

8.4.2. Colour 84

8.4.3. Max Cut 85

8.5. How clique-partitioned treewidth and BBKMZ-treewidth algorithms relate to each other 86

9. Conclusion 89

9.1. Future Work 89

Bibliography 93

A. Problem definitions 99

1. Introduction

In 1936 Alan M. Turing published *On computable numbers with an application to the Entscheidungsproblem* [Tur37] and with this paper and his later work on the Ace [HH00] heralded in a new age of solving problems. While using implementations of Turing's machines to solve different problems without the need of changing the hardware for each problem, it was soon discovered that some problems, like the Halting problem, cannot be solved by any means [Tur37] and other problems, that can be solved, take a long time to be solved with the currently known algorithms. The last observations led to the introduction of the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ as well as the concept of $\mathcal{NP}$-completeness by Cook and Levin in 1971 [Coo71]. Informally speaking, $\mathcal{P}$ includes all problems that can be solved in polynomial time and $\mathcal{NP}$ includes all problems verifiable in polynomial time. Since $\mathcal{P}$ obviously is a subset of $\mathcal{NP}$, this leads to the question, which was posed by Cook in 1971 [Coo71], whether $\mathcal{P}$ and $\mathcal{NP}$ are the same and all problems that can be verified efficiently can also be solved efficiently. While so far nobody was able to answer this question, researchers have tried to better understand this topic by finding problems in $\mathcal{NP}$ that are especially hard, called $\mathcal{NP}$-complete problems, and thus are candidates for separating $\mathcal{P}$ and $\mathcal{NP}$.

While proving $\mathcal{NP}$-completeness might seem like a crushing result for the unknowing reader, many alternative approaches have been found that still allow us to solve $\mathcal{NP}$-complete problems in reasonable time. One idea is to limit ourselves to such instances that are easier to solve. This idea is backed by the observation that for many problems a large number of instances appearing in practical problems can be solved rather fast[1]. This leads us to the field of parametrized algorithms. Here, the idea is to introduce a parameter that measures some property of the instance and then extract the non-polynomial complexity into this parameter. We then can solve all instances where this parameter is small in sufficiently low time. Since many $\mathcal{NP}$-complete problems have a notion of solution size, a first idea could be to use this as parameter. This leads to some successes, like for Vertex Cover, and some intractability results, like for Clique, [Cyg+15].

Thus, it might be of interest to analyse other parameters. For graph problems we can use the parameter treewidth introduced by Robertson and Seymour in 1986 [RS86]. They defined the tree decomposition of a graph as so called bags structured in a tree. Those bags contain a subset of the vertices of the original graph and form a recursive hierarchy of separators of it. The width of a tree decomposition is then given by its largest bag and the treewidth by the smallest width of any tree decomposition of the graph. Informally, this parameter describes how treelike the graph is and thus a low parameter should lead to fast algorithms since many problems can be solved efficiently on trees. Due to the fact that the bags form separators, the parameter can be used to design dynamic programs (DP) by solving the remaining parts with only minor dependencies on the bags. Since its introduction, many algorithms have been found using this framework (e. g. [Cyg+15], [BCKN15]) and this parameter has come to be quite well understood. While treewidth has allowed many algorithms to be found, it struggles with one major problem. This parameter only considers the number of vertices in

[1] See [HW12] for a discussion of heuristic approaches.

each bag and thus large cliques in the graph increase it linear to their size [BM90]. Since cliques are rather common in real life graphs, especially social networks, [Jai20] this means that treewidth can be high in practical inputs [MSJ19]. Since all algorithms depend on this parameter being low, this has major impacts on its usefulness.

Due to this, many different ideas have been proposed to decrease this parameter while preserving its usefulness (see Section 1.1). In 2023 Bläsius et al. [BKW23] introduced clique-partitioned treewidth, which is an adaptation of treewidth that decreases the influence of cliques on the size of the parameter. Here, they adapt the definition of tree decompositions by additionally considering the structure of each bag. This is done by partitioning each bag into cliques and letting each clique only increase the parameter logarithmically in its size. In their work, the authors show promising results for their parameter by discovering through practical examination that it can be far smaller than treewidth and proving that it is smaller than the related BBKMZ-treewidth. Furthermore, they introduce and examine different heuristic approaches for computing clique-partitioned treewidth and make a first step towards showing the parameter's algorithmic usefulness by giving one simple algorithm that solves Independent Set. While the work of Bläsius et al. [BKW23] gives some first promising results, it mainly focusses on the computation of their parameter. In particular, not too much is known about the algorithmic potential of the parameter. It is yet unclear whether algorithms solving other problems using this parameter can be found and how fast they can be. The question whether we can always find faster algorithms using clique-partitioned treewidth than using treewidth or if there are problems where this is not possible is also left open by the authors. Finally, it is yet unknown how this parameter compares to other related ones.

This thesis sets out to explore the parameter and answer these open questions. We first compare the parameter, in addition to treewidth, to the parameters BBKMZ-treewidth, tree-clique width and tree-independence number. This helps us in understanding the size of the parameter. The next step is, to find algorithms using this parameter. We first prove that parametrized algorithms for clique-partitioned treewidth can be found, when it is possible to find algorithms with treewidth. Then, we give fast algorithms for many problems using the normalizations of nice, extended and smooth clique-partitioned tree decompositions. Here, the first two are adapted from normalizations of traditional tree decompositions, while the last one allows us to trace cliques throughout the decomposition. We also introduce two secondary parameters, called clique-degree and neighbourhood index, that allow us to find efficient algorithms for problems where this is impossible without those parameters. Apart from proving for some problems that using clique-partitioned treewidth increases the dependency on the parameter from single to double exponential, we also show lower bounds for the problems considered. Our final contribution is a framework that allows us to describe algorithms using the parameter and that also can be used to understand how algorithms based on BBKMZ-treewidth and algorithms using clique-partitioned treewidth relate to each other. All these results allow us to understand the size of the parameter and its algorithmic usefulness and thus we show that clique-partitioned treewidth is a parameter worth using and looking into.

1.1. Related Work

In the field of parametrized algorithms in general and treewidth-based parameters in particular there have been many results and discoveries. For this thesis the book of Cygan et al. [Cyg+15], which summarizes many ideas and algorithms on this topic, and the work of de Berg et al.

[Ber+18], who introduced many algorithms for their parameter BBKMZ-treewidth, for example algorithms solving Independent Set, Dominating Set, Steiner Tree, Maximum Induced Forest and Hamilton Cycle, and supplied many basic ideas on which a large number of the algorithms given in this work are based on, were instrumental. Most important for our work was the introduction of clique-partitioned treewidth by Bläsius et al. [BKW23].

In this section, we give short introductions to related research that is useful for better understanding clique-partitioned treewidth. We first summarize how clique-partitioned treewidth can be computed [BKW23] and then introduce some related parameters. Those are the BBKMZ-treewidth [Ber+18], which uses global clique-partitions, that means of the entire graph, instead of local ones, that means of each bag, the tree-clique width [Aro21], which is based on edge clique covers, and the tree-independence number [DMŠ22], which considers the size of a largest independent set.

Here, the BBKMZ-treewidth, as the first of those parameters to be introduced, is hugely influential. Bläsius et al. [BKW23] use this parameter as basis for their definition of clique-partitioned treewidth. Their aim is to decrease the parameter and ease the computation while preserving its usefulness. While the authors prove the first two aspects, this thesis sets out to achieve the later. Due to the fact that the clique-partitioned treewidth is a modification of BBKMZ-treewith that uses local instead of global partitions, those parameters are closely related.

1.1.1. Computation of clique-partitioned treewidth

For the computation of clique-partitioned treewidth we refer to Bläsius et al. [BKW23] and only give a short summary of the author's results. They calculate upper bounds for the clique-partitioned treewidth in a two step procedure by first calculating a minimum width tree decomposition of the graph using the state of the art algorithms and then partitioning each bag into cliques. They show that the task of finding the clique-partition is $\mathcal{NP}$-complete. The authors then give an exact branch-and-bound algorithm as well as different heuristic approximations. It is important to note the fact that optimality of the clique-partitioned tree decomposition is already lost by dividing into two phases. Since all heuristics and the branch-and-bound algorithm begin with enumerating all maximal cliques, of whom exponentially many can exist [MM65], all currently known methods of computing an approximation of clique-partitioned treewidth run in exponential worst time.

In a practical examination of their results, the authors noticed, especially in highly clustered graphs, a large reduction of the parameter compared to traditional treewidth. Additionally, it was noticed that some of the heuristics are considerably faster than the branch-and-bound algorithm, while still providing close to optimal solutions (at least in the second phase). The authors also found that all heuristics were sufficiently fast to be practicably usable. Thus, the branch-and-bound algorithm can be used as a reference to estimate the quality of approximated solutions.

1.1.2. Related parameters

As mentioned before, other authors have adapted treewidth to improve on its flaws and design new parameters. All those parameters introduce additional structural elements to the tree decomposition. Below we list some closely related ones.

BBKMZ-treewidth. A parameter closely related to clique-partitioned treewidth is the parameter defined by de Berg et al. [Ber+18] which we call BBKMZ-treewidth following the use of Bläsius et al. [BKW23]. This parameter differs from clique-partitioned treewidth in the main aspect that it uses a global partition instead of a local partition of each bag. Thus, we can express the BBKMZ-treewidth using the framework of clique-partitioned treewidth as calculating a clique-partitioned tree decomposition, where each local partition of a single bag is induced by the global clique-partition of the entire graph [BKW23]. The authors use this parameter to give algorithms for, among others, Independent Set, Dominating Set, Steiner Tree and Hamilton Cycle. They then proceed by proving lower bounds. The authors only show these results for geometric intersection graphs, but those could be adapted to general graphs.

Tree-clique width. Next, we want to introduce tree-clique width and begin by considering how clique-partitioned treewidth and tree-clique width relate to each other. We can see clique-partitioned treewidth as an adaptation of tree-clique width, which was introduced by Chris Aronis [Aro21], where the logarithmically weighted sum of clique sizes of a clique-partition are considered instead of the size of an edge clique cover [BKW23]. Apart from proving the hardness of computing the tree-clique width of a graph, Aronis adapts many approaches used for computing treewidth to his parameter, but does not use his parameter in any algorithms.

Tree-independence number Another related parameter is the tree-independence number, which is an adaptation of independence numbers of graphs to tree decompositions. This concept was introduced by Dallard et al. [DMŠ22]. Here, the main idea is to define the parameter in relation to independent sets instead of cliques. Since independent sets and cliques are closely related, we can also hope for relations between the parameters. The authors give an algorithm for Weighted Independent Set and show some theoretical results.

1.2. Outline

In this thesis, we want to explore clique-partitioned treewidth. This means that we want to understand the capabilities of this newly introduced parameter. Our contribution contains comparisons with other parameters, a large number of algorithms using this parameter, lower bounds that prove some of our algorithms optimal and a framework for designing algorithms with clique-partitioned treewidth. Our work is structured in the following way. After introducing the necessary notation and giving some basic concepts in Chapter 2, we compare our parameter to other related parameters and values to give a basic understanding of how it fits into the already existing landscape in Chapter 3. Before we give any algorithms, we establish in Chapter 4 when it is possible to find algorithms based on clique-partitioned treewidth, give a general algorithm for all problems where this is possible and prove a generalized lower bound. We also establish a separation between treewidth and clique-partitioned treewidth with the use of 3-Clique Cover. Here, we prove that, under the Exponential Time Hypothesis, clique-partitioned treewidth is weaker by $f(x) = 2^{\left(x^{1/(1+\varepsilon)}\right)}$ than treewidth for the problem. This means that we can find $2^{\mathcal{O}(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithms, while this is not possible for clique-partitioned treewidth.

The next step is to start building algorithms in Chapter 5. Here, we begin by giving our first and second normalization steps, called nice and extended cp-tree-decompositions and giving algorithms for Independent Set and Maximum Induced Forest. The second normalization step is needed, since some algorithms need another node-type. When trying to use those techniques on other problems we find some difficulties and thus prove that clique-partitioned treewidth is weaker than treewidth for a number of problems with the use of chordal and co-planar graphs. This is done in Chapter 6. After this result, in Chapter 7 we adapt our approach by using a secondary parameter and due to this can describe algorithms for Dominating Set, Steiner Tree and Clique. We introduce two such parameters called clique-degree and neighbourhood index. In this chapter, we also prove that the secondary parameter is necessary when using our approach.

Finally, we introduce a third normalization step in Chapter 8, which we call smooth and extended smooth cp-tree-decompositions. Here, one is based on the nice and the other on the extended cp-tree-decomposition. The smooth cp-tree-decomposition allows us to trace cliques throughout the clique-partitioned tree decompositions. This lets us describe algorithms for Hamilton Cycle, Vertex Adjacent Feedback Edge Set, Colour and Max Cut in Chapter 8. Additionally, we use the techniques introduced in this chapter to describe a framework for algorithms using clique-partitioned treewidth and a discussion of the relation to algorithms using BBKMZ-treewidth. We end with a conclusion and some ideas for further research in Chapter 9. In Appendix A we give the formal definitions of the problems used.

2. Preliminaries

In this chapter we introduce all necessary concepts needed to understand this work. For this, we explain the notation used and then introduce the rank-based approach, which we use later on to reduce the number of partial solutions to consider. Next, we summarize the Exponential-Time Hypothesis and its strengthened version the Strong Exponential-Time Hypothesis. Those two are generally used to show lower bounds for algorithms and are applied by us in the same way. Finally, we give the definitions of treewidth, BBKMZ-treewidth and clique-partitioned treewidth which are the concepts fundamental for this thesis. For our definitions and notation we use the paper of Bläsius et al. [BKW23] as basis and add further details and new concepts when needed. Our notation of functions is drawn from Cygan et al. [Cyg+15], but adapted at some points to fit our purpose.

2.1. Notation

In this section we introduce how we denote different terms and which abbreviations we use in our mathematical language.

2.1.1. General mathematical notation

We abbreviate the first n natural numbers using $[n] = \{1, \dots, n\}$ and assume two to be the standard base for logarithms. Additionally, for sets A and B, function $f : A \to B$ and value $\beta \in B$ we define the new function

$$f_{a \to \beta} = \begin{cases} f(x) & x \neq a \\ \beta & x = a \end{cases}.$$

Here, the value of the function at position a is changed to β. Furthermore, we denote the restriction of f to $Y \subseteq A$ as $f|_Y$. We use f^{-1} to talk about the inverse of the function f.

2.1.2. Graphs

In this thesis we assume all graphs to be simple and undirected and use uv for edge $\{u, v\}$. In such a graph G we use $V(G)$ for the vertices and $E(G)$ for the edges of G, where we choose $n = |V(G)|$ and $m = |E(G)|$. We then refer to the subgraph of G induced by the set $X \subseteq V(G)$ as $G[X]$ and use respectively $G - X$ or $G - v$ to refer to the subgraph gained by removing the vertex set X or the vertex v from $V(G)$ (and its incident edges from $E(G)$). In similar terms we denote the insertion and removal of elements from a set by + and -.

For some problems we need weights and for this we want to extend a weight function to sets. Given a weight function $g : V(G) \to M$ of co-domain M for a graph G we define the extension to vertex sets as follows: $g(X) = \sum_{x \in X} g(x)$ for a subset $X \subseteq V(G)$.

For a graph $G = (V, E)$ we call $S \subseteq V$ a separator if its removal disconnects the graph, in more formal terms this means that $V - S$ can be split into two disconnected vertex sets V_1 and V_2. Here we have $V_1 \cup V_2 = V - S$ and $V_1 \cap V_2 = \emptyset$. For a Graph $G = (V, E)$ we call $S \subseteq V$ a 3-separator if its removal disconnects the graph into three components, in more formal terms this means that $V - S$ can be split into three disconnected vertex sets V_1, V_2 and V_3. Here, we have $V_1 \cup V_2 \cup V_3 = V - S$ and $V_i \cap V_j = \emptyset$ for $i, j \in [3]$. We say that a (3-)separator is minimal if no subset of it is also a separator.

Finally, we want to introduce a common graph family; for this, we call the graph that consists of one large clique of size $n \in \mathbb{N}_+$ a n-clique and use K_n for it. As a subgraph, a clique means the complete graph with $\binom{n}{2}$ edges.

2.1.3. Further notation

We introduce some additional terms used in this work.

Partitions. A partition assigns elements of a set into partition classes such that each element appears in exactly one partition class. Thus, a set of subsets of the base set is called a partition if the union of all subsets equals the base set and all subsets are pairwise disjoint. For a partition $\mathcal{P}$ of a set X we also use $\mathcal{P} - x$ to remove element x from the partition and the set. Similarly we use $\mathcal{P} + x$ to add x to the set and to the partition as its own partition class. We denote moving element $x \in X$ from partition class p to partition class q by $move_{p \to q}(x, \mathcal{P})$. For this, the element is assigned to partition class p by the partition $\mathcal{P}$ and q either is a partition class of $\mathcal{P}$ or will be created as a new, and before moving x, empty partition class.

Problems. For a problem Π parametrized by parameter k we use Π_k to denote this fact.

Since a clique-partitioned tree decomposition is a useful tool only when our problem contains a graph, we want to restrict ourselves to the subclass of graph problems. We do not give a complete definition of this subclass and instead rely on the common understanding of this term and additionally note some important aspects. For a *graph problem* we require the instance of the problem to contain a graph and the question of this problem should relate to the graph.

Comparing running times for two parameters. Since we consider different parameters and the running times of their algorithms for problems, we introduce a notation that allows us to compare two parameters in respect to their best possible algorithms for a problem. Let k and l be two parameters, $f(x) \in \omega(x)$ and g be computable functions and Π be a problem. We then say that *l is weaker by f than k for* Π, when there is a $g(k) \cdot n^{\mathcal{O}(1)}$ but no $g(o(f(l))) \cdot n^{\mathcal{O}(1)}$ algorithm for Π. If l is weaker by f than k for Π using any f, we generalize this and say that *l is weaker than k for* Π.

Notation in the rank-based-approach. This notation is the same as the one used by Bodlaender et al. [BCKN15] in their work and only adapted to use our symbols in the explanation.

The authors assume a base set U and using this give their definitions. For them $\Pi(X)$ denotes the set of all partitions of a set X. For $U \subseteq X$ the authors use $p_{\uparrow X} \in \Pi(X)$ for the equivalence relation obtained by adding singletons for every element in $X - U$ to p. For $X \subseteq U$ the authors use $p_{\downarrow X} \in \Pi(X)$ for the equivalence relation obtained by removing all elements

not in X from p. The authors use the notation $U[X]$ for a set $X \subseteq U$ to denote the equivalence relation on U where one block is $\{X\}$ and all other blocks are singletons. For $a, b \in U$ the shorthand is $U[ab] = U\{a, b\}$. The authors use $\sqsubseteq$ for the coarsening relation on $\Pi(U)$. Also for two equivalence relations p and q, $p \sqcup q$ is obtained as follows: Let $\sim$ be the relation on the elements with $v \sim w$ if and only if v and w belong to the same set in p or v and w belong to the same set in q. Now, $p \sqcup q$ is the equivalence relation of U using the equivalence classes of the transitive closure of $\sim$.

2.2. The rank-based-approach

We later on find algorithms for connectivity based problems. There, it will be necessary to track which parts of our solution are already connected and which currently are separated. The naive approach can handle this information by considering $2^{\Theta(|S| \cdot \log |S|)}$ equivalence relations per subset S. To improve on this we will use the rank-based-approach of Bodlaeder et al. [BCKN15], which allows us to represent those equivalence relations by only $2^{|S|}$ relations per subset. We give a short introduction to this approach.

For this, note that we use the terminology of de Berg et al. [Ber+18] and use equivalence relations instead of partitions (used in the original work) in order to prevent confusion with clique-partitions.

The authors define a set of operators and show that each operator preserves representation, this means that applying the operator to a set of equivalence relations and its representation ensures that the modified representation still represents the modified set. Here, a set of equivalence relations can informally speaking be said to represent another set, if it keeps the optimum[1] the same. Those cases where the representation is substantially smaller than the original set are of interest.

For a base set U and $A \subseteq \Pi(U) \times \mathbb{N}$ they define the following operators with $\text{rmc}(A) = \{(p, w) \in A | \nexists (p, w') \in A \wedge w' < w\}$, which removes non-minimal weight copies:

Union. Union combines two sets of weighted equivalence relations and discards the dominated equivalence relations. For $B \subseteq \Pi(U) \times \mathbb{N}$ the authors define $A \Downarrow B = \text{rmc}(A + B)$.

Insert. Insert adds elements to U and adds them as singletons to each equivalence relation. For $X \cap U = \emptyset$ the authors define $\text{ins}(X, A) = \{p_{\uparrow U+X}, w | (p, w) \in A)\}$.

Shift. Shift increases the weight of each equivalence relation by w'. For $w' \in \mathbb{N}$ they define $\text{shft}(w', A) = \{(p, w + w') | (p, w) \in A\}$.

Glue. For u, v glue combines, in each equivalence relation, the sets containing u and v into one and adds them to the base set if needed. Then, the authors define $\hat{U} = U + u, v$ and $\text{glue}(uv, A) = \text{rmc}(\{(\hat{U}[uv] \sqcup p_{\uparrow \hat{U}}, w) | (p, w) \in A\})$, where glue is a function with co-domain $\Pi(\hat{U}) \times \mathbb{N}$. For $\omega : \hat{U} \times \hat{U} \to \mathbb{N}$ they define $\text{glue}_{\omega}(uv, A) = \text{shft}(\omega(u, v), glue(uv, A))$.

[1]The authors give a formal definition what this optimum is (See Definition 3.4 in [BCKN15]).

Project. Project removes all elements of a set from each equivalence relation and discards equivalence relations where this reduces the number of sets. The authors define for $X \subseteq U$ $\text{proj}(X, A) = \text{rmc}(\{(p_{\downarrow U-X}, w) | (p, w) \in A \wedge \forall e \in X : \exists e' \in U - X : p \sqsubseteq U[ee']\})$, where proj is a function with co-domain $\Pi(U - X) \times \mathbb{N}$.

Join. Join extends all equivalence relations to the same base set and for each pair of equivalence relations returns the outcome of the join operation $\sqcup$, where the weights are added. They define for $B \subseteq \Pi(U') \times \mathbb{N}$ that $\text{join}(A, B) = \text{rmc}(\{(p_{\uparrow U+U'} \sqcup q_{\uparrow U+U'}, w_1 + w_2) | (P, w_1) \in A \wedge (q, w_2) \in B\})$, where join is a function with co-domain $\Pi(U + U') \times \mathbb{N}$.

The authors then show the following lemma:

Lemma 2.1 (Bodlaeder et al. [BCKN15]): *Each of the operations union, shift, insert, glue, and project can be performed in time $S \cdot |U|^{\mathcal{O}(1)}$ where S is the size of the input of the operation. Given $A, B \subseteq \Pi(U) \times \mathbb{N}$, $\text{join}(A, B)$ can be computed in time $|A| \cdot |B| \cdot |U|^{\mathcal{O}(1)}$.*

Finally, they proof that a reduce algorithm exists that finds a small representation. We again quote the lemma of the authors.

Lemma 2.2 (Bodlaeder et al. [BCKN15]): *There exists an algorithm* reduce *that given a set of weighted equivalence relations $A \subseteq \Pi(U) \times \mathbb{N}$, outputs in time $|A| \cdot 2^{(\omega-1)\cdot|U|} \cdot |U|^{\mathcal{O}(1)}$ a set of weighted equivalence relations $A' \subseteq A$ such that A' represents A and $|A'| \leq 2^{|U|-1}$, where $\omega \leq 2.3727$ denotes the matrix multiplication exponent.*

We now leave the paper by Bodlaender et al. [BCKN15] behind and talk about some implications for our work. When we use the rank-based-approach, it will be necessary to use the above operations to keep track of the equivalence relations for each partial solution. Additionally, we will use reduce after processing each bag to bound the number of equivalence relations. For this, we want to show that this does not increase our running time.

Theorem 2.3: *For partial solutions X using the rank-based approach costs at most $2^{\mathcal{O}(|X|)}$ additional time per bag.*

Proof. It is easy to see that the time spend on the rank-based-approach is dominated by the running time of reduce [BCKN15]. Since our base set are the nodes we picked into the partial solution X and each input set of equivalence relations (above known as A) is a subset of the power set of X and thus contains less than $2^{|X|}$ elements, this together with Lemma 2.2 shows that the time spend on reduce is at most $2^{|X|} \cdot 2^{2\cdot|X|} \cdot |X|^{\mathcal{O}(1)}$.
We can simplify this:

$$2^{|X|} \cdot 2^{2\cdot|X|} \cdot |X|^{\mathcal{O}(1)}$$

By using $|X|^{\mathcal{O}(1)} \in 2^{\mathcal{O}(|X|)}$ we can get:

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

This proves the theorem. ∎

2.3. Exponential-Time Hypothesis and Strong Exponential Time Hypothesis

Throughout this thesis, we want to prove lower bounds for different problems parametrized by clique-partitioned treewidth and thus, in the best case, want to show that our algorithms are optimal. For this, it is necessary to make some assumptions, since the problems we consider are $\mathcal{NP}$-complete. An obvious assumption would be $\mathcal{P} \neq \mathcal{NP}$, but to prove that certain parametrized running times are necessary we need stronger statements. For this, most commonly the Exponential-Time Hypothesis (ETH) and the Strong Exponential-Time Hypothesis (SETH) are used. In this section we introduce those concepts and repeat some of its basic attributes.

ETH was first introduced by Russell Impagliazzo and Ramamohan Paturi [IP99], [IP01]. We summarize their results. They first define δ_q (for $k \geq 3$) as the infimum of all $\delta \in \mathbb{R}$ for whom an $\mathcal{O}(2^{\delta \cdot n})$-algorithm for solving q-Sat with n variables exist. Then, the Exponential-Time Hypothesis states that for $q \geq 3$ we have $\delta_q > 0$. They also give a few equivalent statements: First that if for some q there is $\delta_q > 0$, then $\delta_3 > 0$ and finally that k-SAT (for $k \geq 3$) does not have a subexponential-time algorithm. They also introduce SETH, which states $\lim_{q \to \infty} \delta_q = 1$[2]. In less formal terms this means that no algorithms solving k-Sat can be better than $(2 - \varepsilon)^n$ [BM19].

Generally ETH is assumed to be true while the situation for SETH is unclear [Blä21b], [Pil17], [LMS18]. We know that SETH implies ETH [IPZ01], that ETH implies FPT $\neq W[1]$ [CEF12][3] and that ETH implies $\mathcal{P} \neq \mathcal{NP}$ [Lam21]. Through different reductions ETH has been used to show lower bounds for many problems. We similarly use ETH by reducing from already proven lower bounds to show that our algorithms are optimal or close to optimal.

2.4. Defining treewidth, BBKMZ-treewidth and clique-partitioned treewidth

Here, we introduce the definitions of tree decomposition and treewidth, as well as their alteration to clique-partitioned tree decomposition and clique-partitioned treewidth. We also introduce the variant of de Berg et al. [Ber+18] on which our parameter is based.

Traditional treewidth. A *tree decomposition* of a graph G is the pair (T, B) of a tree T and a function B, mapping each node of T to a subset of $V(G)$, which we call *bag*. For this, pair we require three properties: (1) every vertex is contained in at least one bag, (2) for each edge one bag exists that contains both endpoints of the edge and (3) for each vertex the bags containing this vertex form a connected subtree of T. For each tree decomposition we define the *width* of the decomposition as the size of the largest bag minus one. To define the *treewidth* $\text{tw}(G)$ of the graph we take the minimum width of all tree decompositions of G. For a node t of the tree decomposition we use V_t to denote the union of the vertices of all bags positioned in the subtree bellow t. This includes $B(t)$. We call a tree decomposition *redundant*, if the bag of a node is a subset of the bag of an adjacent node. Otherwise we call the tree decomposition non-redundant. Since a non-redundant tree decomposition of the same width

[2]The name SETH was not used by the original authors and only added later by Calabro et al. [CIP09].
[3]Here, FPT refers to the class of fixed parameter tractable problems for the specified parameter.

and with at most n nodes exists [AZ19] we assume all non-normalized tree decompositions to be non-redundant.[4] To differentiate between the vertices of T and G we use *node* for the former and *vertex* for the later.

We now introduce two adaptations of treewidth, that consider the structure of the bags.

BBKMZ-treewidth. First, we introduce the parameter of de Berg et al. [Ber+18]. Here, the authors partition the initial graph $G = (V, E)$ into cliques. They then define a *κ-partition* of G as a partition $\mathcal{P} = (V_1, \ldots, V_\kappa)$ of V such that $G[V_i]$ for $i \in [\kappa]$ is the union of at most κ cliques. Next, they define the weight of each partition class V_i for $i \in [\kappa]$ as $\log(|V_i| + 1)$. Finally, each partition class is contracted into a single vertex of the same weight, a tree decomposition is computed and each bag is assigned as weight the sum of the weights of its vertices. We call the maximum weight of a bag the *BBKMZ-weight* of a tree decomposition on the contracted graph. The minimum over all such tree decompositions then is called the *BBKMZ-treewidth* of G. By replacing each contracted vertex in the tree decomposition above with the vertices in the partition class it represents we get the *BBKMZ-tree-decomposition*. Here, the vertices of each bag are a subset of the union of a number of partition classes with total weight of the classes lower than the BBKMZ-treewidth. The authors show that this BBKMZ-tree-decomposition can be normalized into a nice BBKMZ-tree-decomposition.

The clique-partitioned treewidth. We now want to improve onto treewidth by considering the structure of each bag in our definition. These ideas are inspired by the work on BBKMZ-treewidth and have the intent of making the parameter smaller and easier to calculate. We thus introduce the *clique-partitioned tree decomposition* (for short cp-tree-decomposition) based on the definition of a tree decomposition (T, B) of graph $G = (V, E)$. Here, we partition for each $t \in V(T)$ the subgraph $G[B(t)]$ induced by the bag into cliques and call this partition $\mathcal{P}_t$. Thus, a cp-tree-decomposition is a triplet (T, B, P), where T and B are defined as before and P is a function mapping each node of T to its clique-partition $\mathcal{P}_t$. For the *weight* of a clique we use $\log(|C| + 1)$ and using this we give the *weight* of bag $B(t)$ as the sum of the weights of all cliques in the partition. Similarly to the definition of traditional treewidth we define the *weight* of a cp-tree-decomposition as the maximal weight of a bag[5]. Finally, we can define the *clique-partitioned treewith* of a graph G, which we abbreviate with cp-treewidth, as the weight of the clique-partitioned tree decomposition of G that has the lowest weight and call it $\text{cptw}(G)$. Also note that a cp-tree-decomposition is still a tree decomposition and thus all terms, like e.g. width, are still applicable.

Furthermore, we want to talk about the number of cliques in each bag and thus define an additional attribute. We define $p(t)$ as the number of cliques in $\mathcal{P}_t$ for a node t of the cp-tree-decomposition and $p(B, P)$ as the maximum of all $p(t)$.

This definition has direct implications for some graphs, which we state here. It also allows us to establish some first insights into cp-treewidth.

[4]Similarly it is easy to see that there also exists a non-redundant clique-partitioned tree decomposition of the same weight for each redundant one.

[5]Note that the minus one is omitted.

We consider the family of k-cliques, abbreviated K_n for $n \in \mathbb{N}_+$. Remember that these are the k vertex graphs that consist of one clique. For this family we can state the following lemma.

Lemma 2.4 (Bodlaender and Möhring [BM90]): *Let $G = (V, E)$ be a graph that contains K_n as a subgraph with a given tree decomposition (T, B). Then, a node t with $K_n \subseteq B(t)$ exists.*

This has direct implications for the treewidth and cp-treewidth of this family.

Lemma 2.5: *Let $G = (V, E)$ be a n-clique. Then, G has cp-treewidth* $\log(n + 1)$ *and treewidth* $n - 1$.

Proof. We use the cp-tree-decomposition that assigns all vertices to one single bag and partitions this bag into one clique. This gives us the above weight and width. We cannot find a lower weight or width since due to Lemma 2.4 each tree decomposition has one bag that contains G. ■

3. Comparison with other parameters

There are many closely related parameters that also use the structure of a bag to obtain better results than the traditional treewidth. We now want to take a closer look at them and determine how they compare to cp-treewidth. These comparisons will help determining how small our parameter can comparatively be and could also be used to show the existence of some algorithms by adapting from those parameters.

When comparing the related parameters to cp-treewidth one can realize that these parameters can broadly be grouped into two types. On the one side there are those parameters that depend on the number of vertices in each clique (or just the number of vertices if there are no cliques) and on the other side there are those that depend on the number of cliques. To the first group belong cp-treewidth, treewidth and BBKMZ-treewidth, to the second belong the number of cliques, tree-clique width and tree-independence number.

A first observation is that bounding a parameter of the first group by a parameter of the second group is impossible since we can just pick few large cliques (see for example Lemma 3.5, Lemma 3.10 and Lemma 3.12) for our graph. In comparison with the other parameters of its group cp-treewidth is generally the smallest and can even be exponentially lower than the other two.

To compare cp-treewidth to the other group one can make two statements: Those parameters tend to be smaller than cp-treewidth (with the exception of tree-clique width) and a rather crude estimation showing that cp-treewidth is larger by at most a factor logarithmic in the treewidth than most of those parameters exists.

3.1. Treewidth

First, we want to compare our parameter to treewidth, from which all related parameters originate. For this, we first prove a helpful lemma and then show bounds in both directions between treewidth and cp-treewidth.

3.1.1. Bounds

We begin with a lemma comparing sums and products. This states that the sum of a set of numbers is lower than their product, if each number is large enough.

Lemma 3.1: *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,* $\sum_{i=1}^{k} x_i \leq \prod_{i=1}^{k} x_i$.

Proof. We proof this lemma by induction on k.
Base case: If $k = 1$, then we have $\sum_{i=1}^{k} x_i = x_1 \leq x_1 = \prod_{i=1}^{k} x_i$.
Induction hypothesis: For every $k \in \mathbb{N}_0 : \sum_{i=1}^{k} x_i \leq \prod_{i=1}^{k} x_i$.
Induction step: We make the step from k to $k + 1$.

$$\sum_{i=1}^{k+1} x_i = x_{k+1} + \sum_{i=1}^{k} x_i$$

Using the induction hypothesis we get:

$$\leq x_{k+1} + \prod_{i=1}^{k} x_i$$

Multiplying two numbers greater than 2 with each other corresponds to repeated adding of the bigger one and thus is greater than (or equal to) a single addition, so we have:

$$\leq x_{k+1} \cdot \prod_{i=1}^{k} x_i$$
$$= \prod_{i=1}^{k+1} x_i.$$

This proofs the above lemma. ∎

We can now prove our main results and begin by comparing cp-treewidth to treewidth:

Lemma 3.2: *Let $G = (V, E)$ be a graph with a given tree decomposition (T, B) of width* tw*. Then, a cp-tree-decomposition of width* cptw $\leq$ tw $+1$ *can be found in polynomial time.*

Proof. When given a tree decomposition of G, we can compute a cp-tree-decomposition by assigning each vertex to its own clique. Using the weight of the cp-tree-decomposition cptw we can make the following estimates.

For this, we look at a node t:

$$\text{cptw} \leq \sum_{C \in \mathcal{P}_t} \log(|C| + 1)$$

We use that in the worst case each clique consits of only one vertex to get:

$$\leq \sum_{C \in \mathcal{P}_t} \log(2)$$
$$= \sum_{C \in \mathcal{P}_t} 1$$

Since each vertex is its own clique, we can transform from summing over all cliques to summing over all vertices.

$$= \sum_{v \in B(t)} 1$$

And finally, we use the definition of treewidth to get:

$$\leq \operatorname{tw} + 1.$$

Since we can use these estimations for every bag, the one with the highest weight will be bounded by its width and thus the width of the tree decomposition. ∎

The next step is to find a bound for the other direction.

Lemma 3.3: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw. *Then, the cp-tree-decomposition has width* tw *of at most* 2^{cptw}.

Proof. For the given graph we can use the following estimations to get our result. We take a look at some node t:

$$\begin{aligned} \operatorname{tw} &\leq \left(\sum_{C \in \mathcal{P}_t} |C| \right) - 1 \\ &\leq \sum_{C \in \mathcal{P}_t} (|C| + 1) \end{aligned}$$

Since each clique has size at least 1 and thus $|C| + 1 \geq 2$ we can use Lemma 3.1 to get:

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

Since we can use these estimations for every bag, the one with the highest width will be bounded by its weight and thus the weight of the cp-tree-decomposition. ∎

3.1.2. Sharpness of bounds

We now want to show that our bounds for treewidth are sharp. For this, we discuss some graph families.

Sharpness of Lemma 3.2. For the family of n vertex graphs, that contain no edges, the optimal tree decomposition and cp-tree-decomposition is assigning each vertex to its own bag. A graph of this family thus has cp-treewidth $\log(1 + 1) = 1$ and treewidth $1 - 1 = 0$. Thus, our inequality of $\text{cptw} \leq \operatorname{tw} + 1$ is sharp.

For connected graphs this example is not possible. While it is obvious that we can then prove a lower bound[1], it might be of interest how much lower this bound could even be. For this, consider the family of trees. They have treewidth one and thus cp-treewdith $\log(3)$. Thus the best possible bound for connected graphs is lower by a factor of $\log(3)$.

Sharpness of Lemma 3.3. Here, we use the family of K_n for $n \in \mathbb{N}_+$. Due to Lemma 2.4 the optimal tree decomposition and cp-tree-decomposition must contain all vertices in one bag. We can calculate the cp-treewidth to be $\log(n + 1)$ and the treewidth to be $n - 1$. This proves that treewidth can grow exponentially in cp-treewidth. Since $2^{\log(n+1)} = n + 1$, our bound is only two higher in this case.

It is easy to see that our bound will never be sharp since we ignore an added constant in the first estimation, but in order to obtain a simpler bound we chose to ignore those constants.

[1] One edge forces both vertices to be in a bag and then we get treewidth one and cp-treewidth $\log(3)$.

3.2. Number of cliques

In the preliminaries we defined $p(B, P)$ as the maximum number of cliques in each bag of a cp-tree-decomposition (T, B, P). Although we do not use it as an parameter in our algorithms, $p(B, P)$ still is used in different settings. Due to this it is of interest to compare it to the cp-treewidth.

3.2.1. Bounds

We begin by finding a bound that depends on the number of cliques and on the treewidth.

Lemma 3.4: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of width* tw *and at most $p(B, P)$ cliques in each bag, with width of at least one. Then, the cp-tree-decomposition has weight* cptw *of at most $p(B, P) \cdot (\log(\mathrm{tw}) + 2)$.*

Proof. We then can estimate in a node t:

$$\mathrm{cptw} \leq \sum_{C \in \mathcal{P}_t} \log(|C| + 1)$$

We know that each clique contains at most tw +1 vertices and thus can simplify to:

$$\leq \sum_{C \in \mathcal{P}_t} \log(\mathrm{tw} + 2)$$

Since tw ≥ 1 we can replace the added two by a factor of four:

$$\begin{aligned} &\leq \sum_{C \in \mathcal{P}_t} \log(4 \cdot \mathrm{tw}) \\ &= \sum_{C \in \mathcal{P}_t} \log(4) + \log(\mathrm{tw}) \end{aligned}$$

Since our summands no longer depend on the cliques themselves we can replace the sum over all cliques by multiplying with the number of cliques $p(B, P)$ and get:

$$\begin{aligned} &\leq p(B, P) \cdot (\log(4) + \log(\mathrm{tw})) \\ &= p(B, P) \cdot (\log(\mathrm{tw}) + 2) \end{aligned}$$

This concludes the proof. ∎

We can now show that finding a bound that solely depends on the number of cliques is impossible. For this, we give a family with a constant number of cliques and increasing cp-treewidth

Lemma 3.5: *There exists an infinite family of graphs with a cp-tree-decomposition (T, B, P) of optimal weight* cptw*, that has a constant number of cliques $p(B, P)$ in each bag and* cptw *growing logarithmic with increasing n.*

Proof. Here, we pick the family of K_n for $n \in \mathbb{N}_+$. It is easy to see that the cp-tree-decomposition with all vertices in one bag is a valid one (Lemma 2.4) and has $p(B_n, P) = 1$ and $\mathrm{cptw}(G_n) = \log(n + 1)$. ∎

Lemma 3.6: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and at most $p(B, P)$ cliques in each bag. Then, $p(B, P) \leq$ cptw is fulfilled.*

Proof. This can be observed by noticing that each clique consist of atleast one vertex and thus has weight of $\log(1 + 1) = \log(2) = 1$ or higher. ∎

3.2.2. Sharpness of bounds

Again, we want to show how good our bounds are by discussing graph families.

Sharpness of Lemma 3.4. Here, we again use the family of K_n. Remember that the whole clique is contained in one bag (Lemma 2.4). These graphs have cp-treewidth of $\log(n + 1)$ and treewidth of $n - 1$ (Lemma 2.5) and consist of only one clique. Both sides of the inequality are only separated by additive constants that result from creating simpler bounds.

Sharpness of Lemma 3.6. Again, we consider the family of n vertex graphs with no edges. We can use the same cp-tree-decomposition as before and get a cp-treewidth of $\log(1 + 1) = 1$ and, since every vertex forms its own bag, we get $p(B, P) = 1$.

If we limit ourselves to connected graphs, we can use the family of trees which have $p(B, P) = 1$ and cp-treewidth $\log(3)$ to get a limit of how much better a bound could be.

3.3. BBKMZ-treewidth

In this chapter we want to compare cp-treewidth to BBKMZ-treewidth, which is the parameter related closest to cp-treewidth. For this, we prove bounds for how large each parameter can be compared to the other. Since we already introduced the parameter in the preliminaries, we begin by proving bounds.

3.3.1. Bounds

Again, we first bound the BBKMZ-treewidth by cp-treewidth. Here, the idea is to use the global partition as a local one.

Lemma 3.7 (Bläsius et al. [BKW23])**:** *Let $G = (V, E)$ be a graph with a given BBKMZ-tree-decomposition of BBKMZ-weight* bbkmz*. Then, G has a cp-tree-decomposition with weight* cptw *of at most* bbkmz*.*

After this we can show a bound for the other direction.

Lemma 3.8: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*. Then, G has a BBKMZ-tree-decomposition with BBKMZ-weight* bbkmz *of at most* $\text{tw} + 1 \leq 2^{\text{cptw}} + 1$.

Proof. For this, we consider the worst possible κ-partition, by assigning each node its own partition class. Then, we get in the tree decomposition of the contracted graph (here, partition class V_i is contracted to vertex v_i, who has weight $\text{weight}(v_i) = \log(|V_i|+1)$), when considering a node t:

$$\begin{aligned} \text{bbkmz} &\leq \sum_{v_i \in B(t)} \text{weight}(v_i) \\ &= \sum_{v_i \in B(t)} \log(|V_i| + 1) \end{aligned}$$

We use that in the case of highest weight each partition class consists of only one vertex to get:

$$\leq \sum_{v_i \in B(t)} \log(2)$$
$$= \sum_{v_i \in B(t)} 1$$
$$= \left(\sum_{v \in B(t)} 1 \right) - 1 + 1$$

And finally, we use the definition of treewidth to get:

$$\leq \operatorname{tw} + 1.$$

Since we can use these estimations for every bag, the one with the highest BBKMZ-weight will be bounded by its width and thus the width of the cp-tree-decomposition. By using Lemma 3.3 to estimate $\operatorname{tw} \leq 2^{\operatorname{cptw}}$ we find $\operatorname{bbkmz} \leq \operatorname{tw} + 1 \leq 2^{\operatorname{cptw}} + 1$. ∎

3.3.2. Sharpness of bounds

Again, we show how good our bounds are by discussing some families.

Sharpness of Lemma 3.7. This bound is sharp in so far, that there are graphs where the best global partition is also the best local partition. You could for example consider the infinite family of n node graphs without edges.

Sharpness of Lemma 3.8. Bläsius et al. [BKW23] prove that there is an infinite family of graphs whose cp-treewidth is exponentially lower than their BBKMZ-treewidth.

Again, similar to our considerations on the sharpness of Lemma 3.3, this bound is only sharp up to constants.

3.4. Tree-clique width

We now want to show how tree-clique width and cp-treewidth are related. Here, we begin with the definition.

Definition. We summarize the definition of tree-clique width provided by its creator[2] [Aro21]. The author defines a *tcl-tree-decomposition* as a tree decomposition, where each node t is assigned a set of cliques C_t. Here, each clique in C_t only uses vertices of bag $B(t)$. This set of cliques C_t needs to meet two conditions: (1) the union of all cliques in C_t needs to equal $B(t)$ and (2) for each edge in between two vertices in $B(t)$ one clique needs to exist that contains both of its incident vertices. This means that those cliques need to cover all vertices and all edges. For each bag the *tcl-weight* of a bag is minimal number of cliques needed for this. Then, for a tcl-tree-decomposition we call the maximal tcl-weight of a bag its *tcl-weight* and and the *tree-clique width* is the minimum tcl-weight of all tcl-tree-decompositions.

[2] In order to avoid confusion with other parameters and their decompositions, we have renamed all terms to use tcl (which is the abbreviation used by the original author for tree-clique width) in it and thus clearly show to which parameter they belong.

3.4.1. Bounds

We begin by proving a bound on cp-treewidth that additionally depends on treewith and then we show that finding a bound that depends solely on tree-clique width is impossible.

Lemma 3.9: *Let $G = (V, E)$ be a graph with a given tcl-tree-decomposition (T, B, P) of tcl-weight* tcl *and width* tw. *Then, G has a cp-tree-decomposition with weight* cptw *of at most* $\text{tcl} \cdot (\log(\text{tw}) + 2)$.

Proof. We start by finding a clique-partition $\mathcal{P}_t$ for each bag of our tcl-tree-decomposition. First note that $p(B, P) \leq \text{tcl}$ since otherwise we could find a better clique-partition by adapting the set of cliques from the edge clique cover and assigning duplicate vertices to one clique and then try again. We then use Lemma 3.4 and the inequality above to get the desired result. ∎

We now give a family with constant tree-clique width and growing cp-treewidth and thus prove bounding the latter by the former impossible.

Lemma 3.10: *There exists an infinite family of graphs with a cp-tree-decomposition (T, B, P) of optimal weight* cptw*, that has constant tree-clique width and* cptw *growing logarithmic with increasing n.*

Proof. Here, we pick the family of K_n for $n \in \mathbb{N}_+$. It is easy to see that the cp- and tcl-tree-decomposition with all vertices in one bag is a valid one (Lemma 2.4) and has $\text{tcl}(G_n) = 1$ and $\text{cptw}(G_n) = \log(n + 1)$. Here, we used that all edges and vertices are covered by one clique. ∎

Finally, we give a bound for cp-treewidth using tree-clique width.

Lemma 3.11: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *Then, G has a tcl-tree-decomposition with tcl-weight* tcl *at most* $2 \cdot \text{cptw}^2 \cdot 2^{\text{cptw}} + \text{cptw}$.

Proof. For this, we need to show how we can cover all vertices and all edges of each bag using the already existing clique-partitions as a starting point. By using the cliques of the partition we can cover all vertices and all edges inside those cliques. Note that it is important to cover all vertices since otherwise there could be a vertex of degree zero. Thus, only the edges between two cliques remain. Let us consider two cliques and consider all edges between the two. For one vertex in the first clique there can be one edge to each vertex of the second clique. All those edges can be covered with one clique of our edge clique cover since the vertices of the second clique are all connected. The same holds for each vertex of the second clique and thus we need at most $2 \cdot |C|$ cliques to cover the edges, where $|C|$ is the size of the larger clique. Since we have at most $p(B, P)^2 \leq \text{cptw}^2$ (Lemma 3.6) pairs of cliques and each clique has size smaller than tw, we need at most $2 \cdot \text{cptw}^2 \cdot \text{tw}$ cliques to cover the inter-clique-edges. We then use Lemma 3.3 and add the cliques of the original partition to get the above bound. Together with the cliques of the partition those cliques fulfil the requirements of the tcl-tree-decomposition. ∎

3.4.2. Sharpness of bounds

Again, we consider how good our bounds are.

Sharpness of Lemma 3.9. Here, we again use the family of K_n. Remember that the whole clique is contained in one bag (Lemma 2.4). These graphs have cp-treewidth of $\log(n+1)$ treewidth of $n-1$ (Lemma 2.5) and tree-clique width of 1, since the clique covers all edges and vertices. Both sides of the inequality are only separated by additive constants which result from the creation of simpler bounds.

Sharpness of Lemma 3.11. We again only prove the asymptotic case, meaning we give a graph family where the tree-clique width is exponential in the cp-treewidth. For this, we use a well known graph family called the *Turán graphs*. This family is a central part of Turáns theorem, which gives an upper bound for how many edges a graph without a k-clique can contain.

We first introduce Turán graphs and show some attributes of interest. Afterwards, we introduce a modification. All information on Turán graphs and Turáns theorem is drawn from Martin Aigner [Aig95].

First the vertex set of our graph $G = (V, E)$ is divided into $k-1$ pairwise disjoint subsets $V = V_1 \cup \cdots \cup V_{k-1}$ and two vertices are joined by an edge if and only if they lie in different subsets V_i and V_j. We call this graph a Turán graph and use $\mathcal{T}(l,k)$ for it, if the vertices are distributed evenly. Formally, this means for all $i, j \in [k-1]$ we have $||V_i| - |V_j|| \leq 1$ and the graph has l vertices. For simplicity sake consider l to be a multiple of $k-1$. The authors prove that the Turán graph $\mathcal{T}(l,k)$ has $\frac{k-2}{k-1} \cdot \frac{l^2}{2}$ edges. Furthermore, the graph does by definition not contain K_k.

The next step is to prove some attributes of Turán graphs. We now need to find a tree decomposition for the graph $\mathcal{T}(l,k)$. Since each bag must be a separator of the graph, there must be at least one bag that contains a smaller Turán graph $\mathcal{T}(l',k') = \mathcal{T}(l - \frac{l}{k-1}, k-1)$. The reason for this is, that it is only possible to separate two vertices if they lie in the same partition class since otherwise they are connected by an edge. And thus, we can in the best case extract one partition class out of a bag.

If we now assume $k = 4$, we get $k' = 3$. Thus, we know that each tree decomposition of an unmodified graph, and resulting from this each tcl-tree-decomposition and each cp-tree-decomposition, has one bag that contains a Turán graph $\mathcal{T}(l', 3)$. We now can calculate an upper bound for the cp-treewidth of the unmodified graph. By assigning each vertex of the graph its own clique we get that the cp-treewidth of the original graph is at most $\frac{3}{2} \cdot l'$ and due to this linear in l'. For this, we solve $l' = l - \frac{l}{k-1} = l - \frac{l}{3} = \frac{2 \cdot l}{3}$ for l and use that the graph has at most l vertices. After this, we can compute the tcl-treewidth. Due to the fact that, $\mathcal{T}(l', 3)$ contains no 3-cliques the best possible edge clique cover is the one using one clique per edge. Since the original Turán graph $\mathcal{T}(l', 3)$ has $\frac{1}{2} \cdot \frac{l'^2}{2} = \frac{l'^2}{4}$ edges the tcl-weight is at least the same and thus quadratic in n' and the cp-treewidth.

Next, we introduce a modification of the graph. We now use h copies of the Turán graph, where each vertex v is replaced by the vertices $v_1, \ldots, v_h$ and $v_1, \ldots, v_h$ form a h-clique. Thus v_i and u_i, $i \in [h]$, are connected if v and u were connected in the unmodified graph. We call the resulting graph $\mathcal{T}^h(l,k)$. It is easy to see that the same restrictions on tree decompositions as for normal Turán graphs hold.

For the next step, we fix $l \geq 20$ in addition to $k = 4$, which we already chose and let h be a variable. Due to this, the optimal clique-partition of the bag partitions the vertices of the modified Turán graph into the h-cliques, if $h > l$. Using the unmodified graph, we can find the

cp-treewidth in the following way: If we now consider that each clique has size h instead of the one in the unmodified graph, we get a cp-treewidth of $\frac{3}{2} \cdot l' \cdot \log(h+1)$. Since we assumed h to be variable and l to be constant this is logarithmic in h.

Finally, we calculate the tcl-treewidth of the modified graph using the tcl-treewidth of the unmodified. In the modified variant, we need to consider the edges outside the K_h since the other ones can be covered by l cliques. Here, we have h copies of the graph and thus need $h \cdot \frac{l'^2}{4}$ cliques to cover the remaining edges, as seen above. Resulting from this the tcl-weight is exponential in the cp-treewidth.

3.5. Tree-independence number

The next parameter we want to compare to cp-treewidth is the tree-independence number of Dallard et al. [DMŠ22]. Here, the main idea is to use maximal independent sets in the definition of the parameter. As before, the first step is to quote the definition.

Definition. The authors define, for a graph $G = (V, E)$ and a tree decomposition (T, B), the *independence number* of the tree decomposition as the maximal independence number of a graph $G[B(t)]$ for a node t. This is the graph induced by a bag of the decomposition. For a graph the *independence number* is defined as the size of the largest independent set in the graph. Using this they give the *tree-independence number* of G as the minimum independence number of all possible tree decompositions of G.

3.5.1. Bounds

We first show that no bound that depends only on the tree-independence number can be found an then find a bound in the other direction.

Lemma 3.12: *There exists an infinite family of graphs with a cp-tree-decomposition (T, B, P) of optimal weight* cptw, *that has constant tree-independence number* tree-α *and* cptw *growing logarithmic with increasing n.*

Proof. Here, we pick the family of K_n for $n \in \mathbb{N}_+$. It is easy to see that the (cp-)tree-decomposition with all vertices in one bag is a valid one (Lemma 2.4) and has tree-$\alpha = 1$ and $\text{cptw}(G_n) = \log(n+1)$. Here, we used that the largest independent set in a clique is of size one. ∎

We now bound the tree-independence number by the cp-treewidth.

Lemma 3.13: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw. *Then, the cp-tree-decomposition has tree-independence number* tree-α *of at most* cptw.

Proof. We bound the tree-independence number via the number of cliques in each bag of the cp-tree-decomposition.

$$\text{tree-}\alpha$$

We use that we can pick at most one vertex from each clique into an independent set to get:

$$\leq p(B, P)$$

We use Lemma 3.6 to prove the lemma:

$$\leq \text{cptw}$$

■

3.5.2. Sharpness of bounds

As before, we want to show how good our bound is and give a graph family for this.

Sharpness of Lemma 3.13. One example where the inequality becomes an equality is for the family of n vertex graphs with no edges. Here, each vertex forms its own bag and thus the largest independent set of a bag is of size one and the weight of the bag is $\log(1 + 1) = 1$.

If we limit ourselves to connected graphs, we can use the family of trees in whom the largest independent set of each bag has size one and cp-treewidth $\log(3)$ to get a limit of how much better a bound could be.

4. Algorithmic potential and limitations

Before we begin with designing algorithms for our parameter we want to establish the limits of it. We prove for which problems we can find algorithms and even give a generalized, but slow, algorithm for those problems for which it is possible. Here, the decisive criteria for cp-treewidth-based algorithms is the existence of a treewidth-based FPT-algorithm. Due to this, we can adapt the treewidth algorithm to our problem. Furthermore, we bound the capabilities of cp-treewidth in the other direction and give a first generalized, but not necessarily tight, lower bound. In later sections we can use these discoveries to know how fast our algorithms could be and to know which running time our algorithm needs to beat. For proving these bounds it is necessary to use some assumptions, since otherwise proving that no polynomial algorithm exist, would correspond to proving $\mathcal{N} = \mathcal{NP}$ and finding other algorithms would involve disproving some conjectures believed true, but unproven. Due to this, we usually assume for our lower bound that ETH holds. Finally, we show that we can separate cp-treewidth and treewidth by giving a problem that has an almost exponentially higher dependence on cp-treewidth than treewidth in the running time of any possible algorithm.

4.1. A Criterion for the existence of FPT-algorithms parametrized by clique-partitioned treewidth

We start by proving two reductions which we use later on in the main proof.

We can use the two lemmas we found when comparing treewidth and cp-treewidth in FPT-reductions between the problem parametrized by cp-treewidth Π_{cptw} and the problem parametrized by traditional treewidth Π_{tw}.

Corollary 4.1: *Let* $G = (V, E)$ *be a graph with a given cp-tree-decomposition of weight* cptw *and* (I, cptw) *an instance of a graph problem* Π_{cptw} *with parameter* cptw*. Then, a FPT-reduction from* (I, cptw) *to* (I', tw) *with* $\mathrm{tw} \leq 2^{\mathrm{cptw}}$*, where* (I', tw) *is an instance of* Π_{tw} *with parameter* tw*, exists.*

Proof. Assume we are given an instance of problem Π_{cptw} based on a graph $G = (V, E)$ and a cp-tree-decomposition of cp-treewidth cptw. We can reduce it to the corresponding problem that uses treewidth as parameter by simply ignoring the clique-partitions of each bag and the structure given by them. Thus, we get a tree decomposition. We use Lemma 3.3 to bound the treewidth of the tree decomposition by 2^{cptw} and since we only change the parameter the running time is obviously in FPT-time parametrized by cp-treewidth. Finally, we note the fact that by not changing graph we receive a yes-instance for cp-treewidth if and only if we receive a yes-instance for treewidth. ∎

Corollary 4.2: *Let* $G = (V, E)$ *be a graph with a given tree decomposition of width* tw *and* (I, tw) *an instance of a graph problem* Π_{tw} *with parameter* tw*. Then, a FPT-reduction from* (I, tw) *to* (I', cptw) *with* $\mathrm{cptw} \leq \mathrm{tw} + 1$*, where* (I', cptw) *is an instance of* Π_{cptw} *with parameter* cptw*, exists.*

Proof. Assume we are given an instance of problem Π_{tw} based on a graph $G = (V, E)$ and a tree decomposition of treewidth tw. We can then use Lemma 3.2 to find a cp-tree-decomposition of the weight cptw $\leq$ tw +1. Since this can be done in polynomial time it can be done during our reduction.

Since we again do not change the graph itself, but only change the parameter and add some structure, we keep our solutions equivalent and bounding our parameter is enough to prove the lemma. ∎

Note that it is only necessary to find an upper limit for the parameters since they only influence the running time and not the solution quality and thus only need to be low enough to achieve FPT-time.

We can use these reductions to prove our main theorem.

Theorem 4.3: *Let $t > 0$ be an integer. A graph problem Π_{cptw} parametrized by cp-treewidth is W[t]-complete if and only if the graph problem Π_{tw} parametrized by traditional treewidth is W[t]-complete.*

Proof. This follows directly from the reductions in Corollary 4.1 and Corollary 4.2. ∎

This theorem also has implications for when problems are in FPT, which we state as a corollary, for which we do not give any further proof.

Corollary 4.4: *A graph problem Π_{cptw} is in FPT if and only if the graph problem Π_{tw} is in FPT.*

We then can use Theorem 4.3 and Corollary 4.4 to construct a first algorithm.

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

Proof. Theorem 4.3 and Corollary 4.4 give an algorithm for each problem by following the reduction and then using the traditional algorithm. Since the traditional algorithm has running time $f(\text{tw}) \cdot q(n)$ for computable function f, polynomial q and treewidth tw, this gives us a $f(2^{\text{cptw}}) \cdot q(n)$ algorithm based on cp-treewidth. ∎

The algorithm obtained by Corollary 4.5 will not be of any practical use in our aim of improving onto treewidth since it is by definition the same algorithm and thus as fast as the traditional algorithm it is based on. Even though it is does not help in practice, this result is interesting from theoretical point of view since it shows that an algorithm can be found and asks the question whether and by how much it can be improved.

Finally, observe that this algorithm implies that at worst cp-treewidth is weaker by $f(x) = 2^x$ than treewidth for all problems. We later on find problems where this bound is met.

4.2. A first general lower bound using the Exponential-Time Hypothesis

Similar to Section 4.1 we use the exploit of ignoring the additional structure to gain a first bound. This idea results in the following theorem:

Theorem 4.6: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw *and f be a function fulfilling $n^{O(1)} \in \mathcal{O}\left(2^{o(f(\text{cptw}))} + 2^{o(f(\text{tw}))}\right)$. If a graph problem Π_{tw} parametrized by traditional treewidth* tw *has no $2^{o(f(\text{tw}))} \cdot n^{O(1)}$ algorithm, then the graph problem Π_{cptw} parametrized by cp-treewidth* cptw *has no $2^{o(f(\text{cptw}))} \cdot n^{O(1)}$ algorithm.*

Proof. If we were given an $2^{o(f(\text{cptw}))} \cdot n^{O(1)}$ algorithm for the clique-partitioned problem, we could use the reduction in Corollary 4.2 from the traditional problem to the cp-treewidth variant and then solve it with this algorithm, which would give us an $2^{o(f(\text{tw}))} \cdot n^{O(1)}$ algorithm for the traditional problem. Here, we need f to be sufficiently fast such that the running time is not dominated by the polynomial part. ∎

We can use this theorem to prove lower bounds for the problems considered in this thesis. Here, we prove bounds based on ETH that bound the order of the exponent. We also could use the theorem to adapt bounds from SETH or other conjectures to adapt stronger bounds, e.g. on the base of the exponential term. For some problems this is enough to prove that the algorithm we find in this thesis is optimal, for other problems we find higher lower bounds by proving that, under ETH, cp-treewidth is weaker than treewidth for those problems in later sections. A list of such problems can be found at the end of this section. This theorem also implies that treewidth cannot be weaker than cp-treewidth for any problem. We proceed by considering one problem (and its related problems) after another and giving the needed reductions to apply Theorem 4.6. During this process we always use a graph $G = (V, E)$ with a given cp-tree-decomposition (T, B, P) of weight cptw and width tw.

(Weighted) Independent Set. For Independent Set we cannot find an $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm since no such algorithm for the traditional problems exist assuming ETH [Mar15]. For Weighted Independent Set we can use the reduction from Independent Set to Weighted Independent Set by setting all weights to one to proof the same bound for this problem.

(Connected) Vertex Cover. For Vertex Cover we can use the fact that the complement of an independent set is a vertex cover (see Corollary 5.11 for more details), were we give our algorithm, and get the same result.

The next problem, for which we search for a lower bound, is Connected Vertex Cover. Here, we construct a reduction between two traditional problems and then apply Theorem 4.6. We start with an instance of Vertex Cover and add one vertex v to all bags and add edges from this vertex to all other vertices. Finally, we increase the size of the covers we allow by one. If we have a vertex cover in the original graph, we can add v to the cover and get a connected vertex cover for the modified graph. If we have a connected vertex cover in the modified graph, it has to include v since v has n neighbours and we would need to select all of those if we do not select v. If we now remove v from the vertex cover, we get a vertex cover of the original graph since we already covered all edges of the original graph without v. For the tree decomposition we can add v to all bags and thus get a tree decomposition of the modified graph and increase the width by only one. We thus cannot find a $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm for traditional Connected Vertex Cover and also no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for the cp-treewidth variant. For other connected problems we can use a similar reduction, if our modification keeps the solutions equivalent.

(Connected) Feedback Vertex Set. Since no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for traditional Feedback Vertex Set exists assuming ETH [BST20], we can conclude the same for the cp-variant. We then can use the fact that maximum induced forests and feedback vertex sets are complements (see the reverse of the reduction employed in the proof of Corollary 5.14 for more detail) to get the same result for Maximum Induced Forest. With a similar reduction as above in the paragraph on Connected Vertex Cover we can transition to the connected variant and thus get a bound for Connected Feedback Vertex Set. Here, we can add v to the feedback vertex set and make it connected and for the other direction v needs to be in the set since otherwise all other vertices would need to be disconnected and thus removing v induces a feedback vertex set.

(Connected) (r-)Dominating Set We use the fact that no such algorithm for the traditional problem exists under ETH [Mar15] to rule out any $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for Dominating Set. For r-Dominating Set we can conclude the same since a reduction from Dominating Set to r-Dominating Set is given by setting r to one.

For the connected problem we give a reduction from Connected Vertex Cover parametrized by treewidth to Connected Dominating Set parametrized by treewidth and then use Theorem 4.6. We start with an instance of Connected Vertex Cover (we call the graph $G = (V, E)$), remove all isolated vertices and replace each edge $uv \in E$ with a triangle. For this, we add a new vertex w_{uv} and edges uw_{uv} and vw_{uv}. This is the same modification as in the commonly used polynomial reduction from Vertex Cover to Dominating Set (see [Mou05] and [Bar05] for polynomial reductions). Remember that this reduction keeps solutions equivalent since a vertex cover is a dominating set if no isolated vertices exist and each dominating set of the modified graph needs to contain one of the vertices in the triangle and thus each edge is covered in the original graph. For this, we replace any occurrences of a w_{uv} with either u or v since w_{uv} dominates only vertices of the triangle.

We thus need to prove that it also transforms connected vertex covers correctly. We are given a connected vertex cover, since it also forms the dominating set and we make no changes, the resulting dominating set is also connected. If we are given a connected dominating set, the resulting vertex cover is again connected since the only modification we make to the set is replacing a vertex w_{uv} with u or v. Since we had a connected dominating set v or u needs to be in the dominating set already since it is connected.

We can find a tree decomposition for the modified graph by using a nice tree decomposition for the original graph and adding a new bag for each vertex w_{uv}. Since this is a tree decomposition there is at least one bag that contains u and v. Out of those bags we pick the one closest to the root and call it t', meaning the bag directly after it, called t, forgets either u or v. Then, we add a bag between t' and t that contains the same vertices as t' and additionally contains w_{uv}. This increases the width by at most one.

This reduction means that no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists for Connected Dominating Set parametrized by treewidth. Due to this no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm can be found for Connected Dominating Set parametrized by cp-treewidth.

(Weighted) Steiner Tree Again, we cannot get an $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for Steiner Tree since no $2^{o(n)}$ algorithm [Mar20b] and thus no algorithm for the problem parametrized by treewidth exists under ETH. The same holds for Weighted Steiner Tree since we can find a reduction from the unweighted to the weighted problem by setting all weights to one.

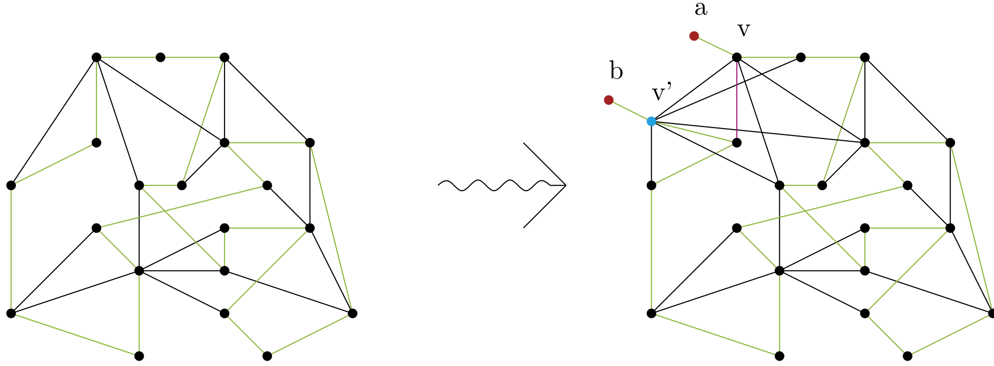


Figure 4.1.: The reduction applied to an example graph and a possible Hamilton Cycle of the graph in green. The endnodes are coloured red and the added vertex blue. The edge we removed to split the cycle is coloured purple.

Clique For Clique we know that under ETH no $2^{o(n)}$ algorithm [Mar15] and thus no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists. By applying Theorem 4.6 we can conclude that no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists.

Hamilton Path and Hamilton Cycle. Since no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for traditional Hamilton Cycle exists under ETH [Mar15], we can conclude the same for the cp-variant.

For Hamilton Path we construct a reduction from traditional Hamilton Cycle to traditional Hamilton Path and then apply Theorem 4.6. For this, we use the construction of Rotenberg [htt][1].

We begin with an Instance of Hamilton Cycle. We choose one vertex $v \in G$ and add a copy v' of it, that is connected to the same vertices as v. After that we add vertices a and b connected to v and v' respectively. A more formal definition is given by transforming $G = (V, E)$ to graph $G' = (V', E')$ with $V' = V + v' + a + b$[2] and $E' = E + av + bv' + vv' \cup \{xv' | xv \in E\}$. We pick a and b as the special vertices for our path.

By adding a, b and v' to every bag as size one cliques we increase the cp-treewidth by at most 3. The additional factor gets lost in $\mathcal{O}$-Notation. For a yes-instance we can split the cycle between v and one of its neighbours and thus get a path from a to v that then follows the cycle through all notes to the neighbour and the goes from v' to b.

If we are given a transformed instance with a hamilton path from a to b and thus a path from v to v' since a and b have no other neighbours, we can then glue together the path to a cycle by connecting v to the vertex trough which v' is accessed from in the path. Note that v and v' have the same neighbourhood except a and b.

This proves our reduction valid and it is easy to see that the reduction does not take to much time. Similarly to the problems before, we thus can not find a $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm for traditional Hamilton Path and also none for the cp-treewidth variant. An example for this reduction can be seen in Figure 4.1.

[1] A simplified construction can be found in the work of Kleinberg and Tardos [KT06].
[2] we have $a, b, v' \notin V$

Vertex Adjacent Feedback Edge Set For the specific version of VERTEX ADJACENT FEEDBACK EDGE SET we used, we could not find any bounds assuming ETH. If any bounds for treewidth exist we can use Theorem 4.6 to get a bound for cp-treewidth. Based on known results a bound of no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm existing, assuming ETH, is likely.

(k-)Colour We know that no $2^{o(n)}$ algorithm exists under ETH for 3-COLOUR [Cyg+15]. Thus, no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ and no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ (Theorem 4.6) algorithm exists, if ETH holds. We can use the reduction gained by adding one vertex that is connected to all old vertices on the traditional problem and then apply Theorem 4.6 to prove the same for k-COLOUR. Since, under ETH, no $2^{o(\text{tw} \cdot \log(\text{tw}))} \cdot n^{\mathcal{O}(1)}$ algorithm for COLOUR can exist [Mar15], we know that no $2^{o(\text{cptw} \cdot \log(\text{cptw}))} \cdot n^{\mathcal{O}(1)}$ algorithm for COLOUR can exist by using Theorem 4.6.

Using SETH we get that no $(k-\varepsilon)^{\text{tw}} \cdot n^{\mathcal{O}(1)}$ [LMS18] and thus no $2^{o(\text{tw} \cdot \log(k))} \cdot n^{\mathcal{O}(1)}$ and $2^{o(\text{cptw} \cdot \log(k))} \cdot n^{\mathcal{O}(1)}$ (Theorem 4.6) algorithms exist for k-COLOUR. We can use this bound to get that no $2^{o(\text{cptw} \cdot \log(k))} \cdot n^{\mathcal{O}(1)}$ exists for k-(COLOUR). Note that our algorithm does not break this bound since $\delta(G, T, B, P)$ can be far larger than $\log(\text{tw})$.

(Weighted) Max Cut For MAX CUT we can not find an $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm since no such algorithm for the traditional problems exist assuming ETH. This can be seen by considering Karps [Kar10] reductions from 3-COLOUR to MAX CUT. Those reductions are linear and no $2^{o(n)}$ algorithm [Cyg+15] and thus no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists for 3-Colour. As seen before, we can set weights to one and get that no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists for WEIGHTED MAX CUT under ETH.

We summarize these results in the following theorem.

Theorem 4.7: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *If ETH holds for the following problems no $2^{o(\text{cptw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists: INDEPENDENT SET, WEIGHTED INDEPENDENT SET, VERTEX COVER, CONNECTED VERTEX COVER, MAXIMUM INDUCED FOREST, FEEDBACK VERTEX SET,CONNECTED FEEDBACK VERTEX SET, DOMINATING SET. r-DOMINATING SET, CONNECTED DOMINATING SET, STEINER TREE, WEIGHTED STEINER TREE, CLIQUE, HAMILTON CYCLE, HAMILTON PATH, COLOUR, k-COLOUR, MAX CUT and WEIGHTED MAX CUT.*

In the next chapters we attempt to build algorithms that match these lower bounds. For (WEIGHTED) INDEPENDENT SET, (CONNECTED) VERTEX COVER, MAXIMUM INDUCED FOREST and (CONNECTED) FEEDBACK VERTEX SET we succeed and thus get ETH tight algorithms. For some of the other problems we show, along the way, stronger lower bounds.

4.3. Separating treewidth and clique-partitioned treewidth with 3-Clique Cover

In this section we want to separate treewidth and cp-treewidth by showing that there is a problem that has a higher bound when parametrizing by cp-treewidth than when parametrizing by treewidth. For this, we consider 3-CLIQUE-COVER and first prove this problem to be in FPT when parametrizing by treewidth or cp-treewidth and then use a reduction to show that, under ETH, cp-treewidth is weaker than by $f(x) = 2^{\left(x^{1/(1+\varepsilon)}\right)}$ treewidth for 3-CLIQUE COVER.

We can rather easily observe that the problem is in FPT, since we can use the monadic second-order (MSO_2) formulas $\text{clique}(X) = \forall_{u,v \in X} \text{adj}(u, v)$ and $\exists_{A \subseteq V} \exists_{B \subseteq V} \exists_{C \subseteq V} \text{clique}(A) \wedge \text{clique}(B) \wedge \text{clique}(C) \wedge (\forall_{v \in V} v \in A \vee v \in B \vee v \in C)$ in Courcelles theorem [Cou90] [3]. Since the algorithms gained by Courcelles theorem are slow and we want to adapt the fastest possible algorithm for treewidth to gain a fast algorithm for cp-treewidth, we give a faster algorithm.

We begin by giving the algorithm and then prove this algorithm to be correct and show its running time. After this, we can summarize our results in Corollary 4.10.

Algorithm for 3-Clique Cover Let $G = (V, E)$ be a graph with a given tree decomposition (T, B) of width tw. We give a dynamic program on a nice tree decomposition. For this, we first compute such a nice tree decomposition of width tw [Cyg+15]. For our partial solutions we assign each vertex of the bag to one of the three sets A, B and C. Those sets will form our cliques when the algorithm is finished. Additionally, we remember for each of those sets whether they are dead or alive. The idea here is, that it is possible to assign further vertices to alive sets and impossible to dead ones.

We can find a 3-clique cover, if we can generate a valid partial solution using the following recursive formulas. We will explain how we create new partial solutions for each node type. For leaf-nodes t we create the base case and thus begin with a partial solution that has all three sets alive and no vertices assigned yet.
For introduce-node t with child t' and introduced vertex v we can create a new partial solution for t that has v in set X from each partial solution for t', if X is still alive and v is adjacent to all vertices of X. If this is true for multiple $X \in \{A, B, C\}$, we create multiple new partial solutions. If the vertex cannot be added to any clique, we cannot find a valid solution.
For forget-node t with child t' and forgotten vertex v we create a new partial solution for t from each partial solution of t' by killing the set to which v was assigned in the partial solution of t', if it was alive.
For join-node t with children t_1 and t_2 we can combine the two partial solutions of t_1 and t_2 to get a partial solution of t, if each vertex of $B(t)$ is assigned to the same set by both children. Furthermore, the two partial solutions can only be combined, if each set is alive in at least one partial solution. If the a set is dead in at least one partial solution, this set is also dead in the resulting partial solution. The input instance is considered a yes-instance, if the algorithm generates a valid solution for the root.

For this algorithm, and especially the formula for join-nodes, it is not obvious that it is correct and thus we proceed by proving the correctness of the algorithm.

Lemma 4.8: *Let $G = (V, E)$ be a graph with a given tree-decomposition (T, B) of width* tw*. Then, the above algorithm solves 3-Clique Cover.*

Proof. We first prove that this algorithm finds all 3-clique-covers of the graph and then show that each solution found by the algorithm is a 3-clique-cover.

If there is a 3-clique-cover, it is always possible to add the introduced vertex into a set when processing introduce-nodes, since the algorithm tests all possible assignments. In the assignment of vertices to sets that corresponds to the 3-clique-cover no vertex is introduced after one vertex of the clique is forgotten. This is since those two vertices share an edge (see

[3]This theorem was introduced by Courcelle [Cou90] and we use the summary of Downey and Fellow [DF13].

requirement (2) and (3) of tree decompositions). Finally, one of the two instances of the set is alive when two branches are joined, since all vertices are connected. Thus, the algorithm finds the 3-clique-cover.

If the algorithm finds a partial solution for the root, then all vertices have been assigned to the three sets. This, is since the algorithm would have stopped, if any vertex could not be assigned. We thus need to prove that each set is a clique. Here, introduce-nodes only add vertices when they connect to all vertices of the set and forget-nodes mark sets with lost vertices as dead. Since join-nodes only join two instances of a set if atleast one of the two is alive and both are assigned the same vertices, it is ensured that all vertices are still connected after joining. To have two disconnected vertices, those two need to be forgotten in two different branches, since otherwise introduce-nodes ensure connectedness. This is ruled out since at least one set is alive. ∎

Finally, we want to analyse the running time of the algorithm.

Lemma 4.9: *Let $G = (V, E)$ be a graph with a given tree-decomposition (T, B) of width* tw*. Then, the above algorithm runs in time $\mathcal{O}\left(3^{\text{tw}} \cdot \text{tw} \cdot n^3\right)$.*

Proof. For this, we note that we can find 3^{tw} different ways of assigning tw vertices to three sets. We additionally store for each set whether it is dead or alive and thus need a factor of $2^3 = 8$, which we hide in the $\mathcal{O}$-notation. We have at most $\mathcal{O}(\text{tw} \cdot n)$ nodes in the nice tree decomposition and can calculate the necessary data for the recursion in $\mathcal{O}(n^2)$. For this, we need to be able to compare two sets of tw vertices each and check adjacency of one vertex to tw others. Thus, the total running time of the algorithm is in $\mathcal{O}\left(3^{\text{tw}} \cdot \text{tw} \cdot n^3\right)$. ∎

Corollary 4.10: *Let $G = (V, E)$ be a graph with a given tree decomposition (T, B) of width* tw*. Then, 3-Clique Cover can be solved in $\mathcal{O}\left(3^{\text{tw}} \cdot \text{tw} \cdot n^3\right)$.*

We then can use Corollary 4.5 to get an algorithm for cp-treewidth that is as fast as the algorithm of Corollary 4.10, but has an exponentially higher dependence on the parameter. The next step is to prove that the algorithm of Corollary 4.10 is optimal.

Lemma 4.11: *Let $G = (V, E)$ be a graph with a given tree decomposition (T, B) of width* tw*. If ETH holds, then no algorithm solving 3-Clique Cover parametrized by treewidth* tw *in time $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. We use proof by contradiction. Assume that an algorithm solving 3-Clique Cover in time $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ exists, then we can get an $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm by using $\text{tw} \leq n + 1$. Since 3-Colour is the complement of 3-Clique Cover [Kar10], we can use this to solve 3-Colour in time $2^{o(n)} \cdot n^{\mathcal{O}(1)}$. This is a contradiction to the fact that, assuming ETH, no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm exists for 3-Colour [Cyg+15]. Thus, no $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm for 3-Clique Cover exists. ∎

This means that the algorithm of Corollary 4.10 is optimal up to constant factors in the exponent. In the rest of this section we prove that the algorithm gained by adapting the optimal algorithm for treewidth using Corollary 4.5 leaves almost no room for improvement.

Before we can give the reduction necessary to prove our claim, we give a definition and introduce an algorithm which we use later.

We say that an algorithm decides between a k- and a l-clique cover of a graph for $k \leq l$ if it satisfies the two following conditions. If the input graph can be covered with k cliques, then the algorithm covers the input graph with l cliques. If the input graph is not coverable with k cliques, then the algorithm either returns that the graph is not coverable with k cliques or still gives a l-clique cover.

Lemma 4.12: *Let $G = (V, E)$ be a graph and let $\gamma : \mathbb{R} \to \mathbb{R}$ be a function fulfilling $\gamma \in \omega(1)$ and $\gamma(n) \leq n$. Then, an algorithm that decides between a 3- and a $\mathcal{O}(\gamma(n))$-clique cover in $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. We begin by partitioning V into $\gamma(n)$ sets of size $\frac{n}{\gamma(n)}$ [4]. We then can test for each set whether it is coverable with 3 cliques by searching all $3^{\frac{n}{\gamma(n)}}$ possible assignments into 3 sets. If any set is not coverable with 3 cliques, we can return that the whole graph is not coverable with 3 cliques. By using different cliques for each of the $\gamma(n)$ sets, we can cover the graph with $3 \cdot \gamma(n) \in \mathcal{O}(\gamma(n))$ cliques. Since we required $\gamma \in \omega(1)$, we have that $\frac{n}{\gamma(n)} \in o(n)$ and thus we have $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ such possible assignments into 3 sets. Finally, note that $\gamma(n) \leq n$ and we thus do not divide into more than n sets. ∎

To make the cp-treewidth algorithm as slow as possible compared to the treewidth algorithm we are interested in good approximations and thus in small γ [5]. One example function we could choose for γ is the log-function or even the nested log-function $\log(\log(\ldots \log(n)))$ for any constant number of nesting. We may also use the inverse of the Ackermann function. For the following reduction fix γ as any function of your choice that fulfils the requirements of Lemma 4.12.

We now give a reduction from 3-Clique-Cover parametrized by treewidth to 3-Clique-Cover parametrized by cp-treewidth that decreases the instance size. This reduction will be useful in our proof.

Reduction. We begin with an instance of 3-Clique-Cover parametrized by treewidth. Here, we are given a tree decomposition of possibly optimal width tw for the graph G. The idea of this reduction is to decrease the parameter almost logarithmicaly by transitioning from treewidth to cp-treewidth and use time $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$. For this, we filter out obvious no-instances and then are left with instances that have low cp-treewidth. First, notice that each bag can be covered with three cliques, if the whole graph can be covered with three cliques since each bag is a subset of all vertices and thus we can pick subsets of our three cliques to cover the bag. Due to this, as a contraposition the graph is not coverable by three cliques if any bag is not coverable with three cliques. We can use this to only test the bags and thus filter out no-instances. We first test if the graph induced by each bag can be covered with $\gamma(\text{tw})$ cliques for the γ we chose above. For this, we can use the algorithm of Lemma 4.12 and need $2^{o(\text{tw})} \cdot n^{\mathcal{O}(1)}$ time. If this is not the case we can return a trivial no-instance. In the next step we only have instances whose bags are coverable with $\gamma(\text{tw})$ cliques. This means

[4]For this algorithm we ignore rounding issues when the sets do not have exactly the same size but are at most one larger or smaller. This case can be handled by rounding the different values or increasing the input size to the next larger number that is properly divisible.

[5]Since the complement of a graph that can be covered with three cliques can be coloured with three colours, we can take a look at results for approximating colour in polynomial time to show that our algorithm is roughly in line with current research. It is $\mathcal{NP}$-hard to colour a 3-colourable graph with five colours [KLS00] ([KT17] claim this paper proves it for five colours). Under a conjecture Dinur et al. [DMR05] prove that colouring a 3-colourable graph with constantly many colours is hard.

that the graphs we did not filter out have weight almost logarithmic in their width since each bag consist of few cliques. To be more formal we can cover each bag with $\gamma(\text{tw})$ cliques and thus have weight of at most $\gamma(\text{tw}) \cdot \log(\text{tw})$. We can use the found clique covers to get our clique-partition. Also note that we did only discard obvious no-instances and thus our reduction produced equivalent instances.

Optimality It might also be of interest how optimal decompositions behave. For this, observe that, if our tree decomposition was optimal, the weight is almost logarithmic in it and thus the cp-treewidth, which is the minimal weight, is as well.

We have now introduced all necessary tools to prove the final theorem of this section and show that each algorithm using cp-treewidth as parameter has an almost exponentially higher dependency on the parameter than the treewidth-based algorithm we gave above. For this, observe that the algorithm itself is not slower since we only compared the two parameters. The difference is solely based on cp-treewidth being a smaller parameter. This result separates between treewidth and cp-treewidth and shows that there are problems for which we need a substantially larger dependence on the parameter.

Theorem 4.13: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *Let $\gamma : \mathbb{R} \to \mathbb{R}$ be a function fulfilling $\gamma \in \omega(1)$ and $\gamma(n) \leq n$. If ETH holds, then no algorithm solving 3-Clique Cover parametrized by cp-treewidth* cptw *in time $2^{o(g(\text{cptw}))} \cdot n^{O(1)}$ exists. Here, $g(x)$ is the inverse function of $\gamma(x) \cdot \log(x)$.*

Proof. We prove this by contradiction. Assume we are given an $2^{o(g(\text{cptw}))} \cdot n^{O(1)}$ algorithm for the clique-partitioned problem. We then could use the reduction from the traditional problem to the cp-treewidth variant given above and reduce from tw to $\text{cptw} \in \mathcal{O}(\gamma(\text{cptw}) \cdot \log(\text{tw}))$. Afterwards, we can solve the reduced instance with the given algorithm, which would give us an $2^{o(\text{tw})} \cdot n^{O(1)}$ algorithm for the traditional problem. This is a contradiction, as seen in Lemma 4.11. ∎

In order to better understand this result, we want to give a simple example for γ and thus for the inverse function of $\gamma(x) \cdot \log(x)$. While it generally is not obvious how this inverse can be found, we use a case in which finding this inverse is easy. If we set $\gamma = \log$, we can simplify this result to:

Corollary 4.14: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *If ETH holds, then no algorithm solving 3-Clique Cover parametrized by cp-treewidth* cptw *in time $2^{o\left(2^{\sqrt{\text{cptw}}}\right)} \cdot n^{O(1)}$ exists.*

Proof. We can use that the inverse of $\log(x) \cdot \log(x)$ is $2^{\sqrt{x}}$ and then apply Theorem 4.13. ∎

We can generalize this result using arbitrary powers and get the following lemma.

Corollary 4.15: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *For a given $\varepsilon > 0$ no algorithm solving 3-Clique Cover parametrized by cp-treewidth* cptw *in time $2^{o\left(2^{\left(\text{cptw}^{1/(1+\varepsilon)}\right)}\right)} \cdot n^{O(1)}$ exists, if ETH holds.*

Proof. We can use that the inverse of $\log^{\varepsilon}(x) \cdot \log(x) = \log^{1+\varepsilon}(x)$ is $2^{\left(\text{cptw}^{1/(1+\varepsilon)}\right)}$ and then apply Theorem 4.13. ∎

We might want to find similar results with lower dependencies, since we can obviously choose smaller, but growing, functions. This again is not trivial, since we need to find the inverse of the resulting term. As an upper limit of how close we can get, consider the following idea. Keep in mind this is no full proof but a concept that needs to be considered in any proof.

We argue that proving $2^{o\left(2^{(c)\cdot \mathrm{cptw}}\right)}$ is not possible with our construction for any constant $c > 1$. For this, we need the inverse of $2^{c\cdot x}$ to be expressible as $\gamma(x, c) \cdot \log(x)$ for $\gamma(x, c) \in \omega(x)$. We then have $2^{c\cdot\gamma(x,c)\cdot\log(x)} = x^{\gamma(x,c)\cdot c} \in \Theta(x)$. This requires $\gamma(x, c) \cdot c \in \Theta(1)$. Since we fix c, it is constant and we have $\gamma(x, c) \in \omega(x)$ which is a contradiction.

We can summarize our results in the following corollary.

Corollary 4.16: *Under ETH, cp-treewidth is weaker by $f(x) = 2^{\left(x^{1/(1+\varepsilon)}\right)}$ than treewidth for 3-Clique Cover, with a given $\varepsilon > 0$.*

Proof. We use Corollary 4.15 together with the traditional algorithms of Corollary 4.10. ∎

Finally, during this section we always considered the tree decomposition to be given in advance. If no tree decomposition is given, we can compute an approximation in polynomial time which has width $\mathrm{tw} \cdot \sqrt{\log(\mathrm{tw})}$ for a graph of treewidth tw [FHL05]. We then can argue in an analogous way to the optimal case and get a result that is almost logarithmic in the approximated width and thus similar to the optimal case and the case where the tree decomposition is given.

5. Building algorithms: First Approach

After characterizing when FPT-algorithms for problems parametrized by cp-treewidth can be found, we want to find such algorithms for different $\mathcal{NP}$-complete problems. Since algorithms for problems parametrized by treewidth use the normalizations of nice and extended[1] tree decompositions, we adapt those to cp-tree-decompositions. For normalizations it is important that they can be computed efficiently and that they keep the number of bags and the cp-treewidth low. The time needed to calculate the normalization does not appear in the asymptotic running time of our algorithms, since it is usually dominated by the time needed to solve the problem.

To generate effective algorithms using these normalizations, our first approach is always to adapt traditional dynamic programming algorithms on traditional treewidth to our new parameter. As done in the traditional algorithm, we first normalize our cp-tree-decomposition. For adapting the algorithms, we use the fact that we have some information on the structure of the bags. Here, we can use the partition into cliques to reduce the number of partial solutions we need to consider. This can be done by bounding the number of vertices of each clique that can be chosen into a valid solution by a constant. This allows us to disregard partial solutions that cannot lead to a valid solution.

5.1. Nice cp-tree-decompositions

This normalization is called nice cp-tree-decomposition and is a close adaptation of nice tree decompositions, which are the commonly used normalizations of tree decompositions. We begin by giving the definition followed by a construction and then show the necessary attributes for our construction. In the end we summarize our results in Theorem 5.4.

Definition. We define a *nice cp-tree-decomposition*, analogous to Cygan et al. [Cyg+15], as a cp-tree-decomposition, fulfilling the following properties: (1) we can choose one node $r \in T$ as *root* with $B(r) = \emptyset$, (2) $B(l) = \emptyset$ holds for every leaf l of T, we thus call l a *leaf-node* and (3) every non-leaf-node has on of the following three types *introduce-node*, *forget-node* or *join-node*.

For this, we define: An introduce-node is a node t, having exactly one child t' with $B(t) = B(t') + v$ for a $v \notin B(t')$. We say that it introduces v. A forget-node is a node t, having exactly one child t' with $B(t) = B(t') - w$ for a $w \in B(t')$. We say that it forgets vertex w. A join-node is a node t, having exactly two children t_1 and t_2 with $B(t) = B(t_1) = B(t_2)$. Similarly, a traditional tree decomposition fulfilling the above conditions, is called nice tree decomposition.

[1] In this section we only use the extended cp-tree-decomposition indirectly, for a more conventional use see Section 7.2.

In our proofs we use the fact that it is possible to find a nice tree decomposition of the same traditional width in $\mathcal{O}\left(\text{tw}^2 \cdot (n+m)\right)$ with at most $\mathcal{O}(\text{tw} \cdot n)$ nodes for a graph with a given tree decomposition of width tw [Cyg+15] [2]. We also use the proof sketch given by the authors to argue that the cp-treewidth does not change either.

Construction. Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight cptw and width tw. We can now explain how to compute a nice cp-tree-decomposition of G. We start with a cp-tree-decomposition of weight cptw and width tw. Remember from Section 2.4 that width corresponds to treewidth and weight corresponds to cp-treewidth. We can now use the traditional construction to get a nice tree decomposition of width at most tw and with no more than $\mathcal{O}(\text{tw} \cdot n)$ nodes in $\mathcal{O}\left(\text{tw}^2 \cdot (n+m)\right)$ time. This nice tree decomposition can be adapted to a nice cp-tree-decomposition by adding a clique-partition for each bag. Thus, we need to define a partition of each bag into cliques. For this, we take a look at the construction used in the traditional proof and then show how we can find a clique-partition for each bag.

The construction used by the authors keeps the original bags of the tree decomposition and adds some intermediate bags. They first add the root (after rooting the tree) and leaf-nodes by simply adding empty bags at each leaf. For node t and its only child t' first all vertices from $B(t') - B(t)$ are forgotten one by one and then the vertices from $B(t) - B(t')$ are introduced one by one. The traditional construction uses introduce- and forget-nodes to turn bags with at least two children into join-nodes. Here, it is necessary to replace nodes with more than two children with binary trees of two-child-nodes.

It can be seen that each bag of the nice cp-tree-decomposition is a subset of a bag of the original cp-tree-decomposition. Thus, we can reuse, possibly restricted to the subsets of the original bags, the clique-partitions of those bags. This completes our construction of a nice cp-tree-decomposition.

After giving the construction, we now want to show that the weight does not increase, this is motivated by the fact that the running time of our algorithms depend on it being low.

Lemma 5.1: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, the nice cp-tree-decomposition constructed above has weight* cptw *and width* tw.

Proof. For this, we need to show that the partitions we did define are good enough to not increase the weight. Since leaf-nodes as well as the original bags and the join-nodes use no non-trivial partitions, we argue only for introduce- and forget-nodes. Here, the added bags have lower weight than the bags whose partition we use, since removing a vertex, while keeping the other clique sizes and the number of cliques the same, only decreases weight.

The bound on width is the same as in the traditional case, since we only add partitions. This proves our lemma. ∎

[2] When looking for a proof of this we found two variants of the statement, one by Kloks [Klo94] and one by Cygan et. al. [Cyg+15]. As far as our search reached, we could not find a full proof for Cygans version, who themselves leave the theorem up to the reader to prove and give only a proof sketch and a reference to Kloks as the original creator of nice tree decompositions. Furthermore, all proof sketches (for the Cygan-variant) are incompatible with Kloks proof. One example proof sketch, which together with the ideas of Cygan can suffice for our needs, is given by Althaus and Ziegler [AZ19]. We thus use the Cygan version and those two proof sketches.

The next step is to show a bound for the number of bags created in our construction.

Lemma 5.2: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of width* tw*. Then, the nice cp-tree-decomposition constructed above has $\mathcal{O}(\text{tw} \cdot n)$ nodes.*

Proof. Since we use the traditional construction and only add partitions to each bag and not increase the number of bags, the traditional bound still holds. Note that we assumed our cp-tree-decomposition to be non-redundant and thus have at most n nodes in the original decomposition (See Section 2.4). ∎

The final attribute we want to show concerns the running time of our construction.

Lemma 5.3: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, the construction above runs in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$.*

Proof. We again only add partitions for each bag and make no further changes to the traditional algorithm and thus do not change the running time, if adding partitions is sufficiently fast. Here, we can note that we find these partitions by using the empty partition, reusing an old partition or removing one vertex from an old partition which all can be done in time $\mathcal{O}(\text{tw})$. By doing this for each of the $\mathcal{O}(\text{tw} \cdot n)$ bags we do not increase the running time. ∎

Theorem 5.4: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, a nice cp-tree-decomposition of G with weight* cptw*, width* tw *and $\mathcal{O}(\text{tw} \cdot n)$ nodes can be computed in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$.*

5.2. Extended cp-tree-decompositions

Our second normalization extends our definition of nice cp-tree-decompositions by a new node type called introduce-edge-node and is an adaptation of the traditional use of this node type. Again, we start by giving the definition followed by a construction and then show the necessary attributes. The last step, as before, is the summary of our results.

Definition. Later on we need an extension of nice cp-tree-decompositions having a further type of nodes called introduce-edge-node. We again use the book of Cygan et al. [Cyg+15] as orientation for our definition of the *extended cp-tree-decomposition*[3]. We first describe the new node type.

A *introduce-edge-node* is a node t, having exactly one child t' with $B(t) = B(t')$ and which is labelled with edge $uv \in E(G)$ for $u, v \in B(t)$. We say that it introduces edge uv. We then define an *extended cp-tree-decomposition* on the basis of the nice cp-tree-decomposition. In an extended cp-tree-decomposition all nodes have one of the node types introduce-, forget-, join, leaf- or introduce-edge-node. Furthermore, we add the following requirements. First, we require each edge to be introduced at least once and at most once on every path from leaf to root. And second, we define that an introduce-node no longer introduces edges, since those are now added explicitly by introduce-edge-nodes.

We can now consider in our algorithms parts of a graph $G_t = (V_t, E_t)$, where E_t analogue to V_t is the set of all edges, which already where introduced in the subtree bellow t.

[3]We differ in one major aspect, since we allow multiple introductions of the same edge. This is necessary to prove Theorem 5.8.

We adapt the construction of Cygan et al. [Cyg+15] for an extended tree decomposition. The change made in comparison to the original work is introducing each edge as early as possible instead of as late as possible since we need these edges to find clique-partitions.

Construction. Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight cptw and width tw. We now explain how to compute an extended cp-tree-decomposition of G. We start with a cp-tree-decomposition of weight cptw and width tw. Remember from Section 2.1 that width corresponds to treewidth and weight corresponds to cp-treewidth. We first compute a nice cp-tree-decomposition of the same weight and same width tw using Theorem 5.4.

The next step is to insert introduce-edge-nodes into the nice cp-tree-decomposition. In our case it is necessary to introduce each edge as soon as both adjacent vertices are in a bag together. Thus, after each introduce-node a series of introduce-edge-nodes (all with the same vertices in each bag) will follow. These introduce all edges from the introduced vertex to all other vertices in the bag. This costs us more additional bags compared to the traditional construction, and for this, we need to relax the requirement of each edge being introduced exactly once to no more than once per path from leaf to root. We introduce each edge at least once (this is since both endpoints must be in one bag due to requirement (2) of tree decompositions) and on the only node type where two different states of how edges were introduced can meet, join-nodes, all edges that are introduced in one child, are also introduced in the other, since we have the same vertices.

The next step is to find cliques partitioning each bag. Here, it is important to remember that in an extended cp-tree-decomposition a introduce-node no longer introduces edges. For the bags in which all edges have been introduced we can use the partitions of the nice cp-tree-decomposition since they have the same edges and vertices. For the other nodes in the series of introduce-edge-nodes, as well as the introduce-node t (introducing v) with child t' at the start of it, we can use the partition of t' and add a new clique for v. When introducing an edge we can check, whether the edge allows us to add v to any clique, and if possible add it to the biggest clique. This will not improve the total weight of the decomposition but will decrease the weight of some bags and thus will decrease running times in practice.

We again need to bound the resulting weight. This is done in the following lemma.

Lemma 5.5: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, the extended cp-tree-decomposition constructed above has weight* cptw +1 *and width* tw*.*

Proof. Our construction added a series of new bags after each introduce-node. Of all those nodes in the series the first one, where no edges have been introduced, is the one with the highest weight, since adding edges and keeping the vertices the same only allows for lighter partitions. For the newly introduced vertex we have to use a new clique since it has no edges and for the other vertices we can use the partition of the child of the bag in question. Thus, we have a partition of weight less than cptw +1 by using that the partition of the child is of weight at most cptw and the single clique has weight $\log(1 + 1) = 1$. It is easy to see that generally a weight of cptw can not be achieved with this construction.

The width stays the same since our construction adds only bags that have the same amount of vertices as the corresponding introduce-node. ∎

The next step is to show a bound for the number of bags created in our construction.

Lemma 5.6: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of width* tw*. Then, the extended cp-tree-decomposition constructed above has $\mathcal{O}(\mathrm{tw} \cdot n \cdot \log(n))$ nodes.*

Proof. To bound the number of nodes we use Lemma 5.2 for the nice cp-treedecomposition we computed in the first step and then bound the number of added introduce-edge-nodes.

Each edge can be introduced at most once per path from a leaf-node to the root, since a vertex cannot be forgotten more than once and thus can not be reintroduced (together possibly with its edges) on a path. Since we only allow for nodes with at most two children, the most paths we can get is a binary tree of join-nodes and thus at most the logarithm of the number of bags. If we use that the nice cp-tree-decomposition has $\mathcal{O}(\mathrm{tw} \cdot n)$ bags, we get for the number of bags added:

$$\begin{aligned}\log(\mathcal{O}(\mathrm{tw} \cdot n)) &= \mathcal{O}(\log(\mathrm{tw} \cdot n)) \\ &= \mathcal{O}(\log(\mathrm{tw}) + \log(n))\end{aligned}$$

Here, we use $\mathrm{tw} \leq n$ to get

$$\begin{aligned}&= \mathcal{O}(2 \cdot \log(n)) \\ &= \mathcal{O}(\log(n)) .\end{aligned}$$

Finally, we need that a graph of treewidth tw has at most $\mathrm{tw} \cdot n$ edges [Cyg+15]. Those three facts can be combined to show that a most $\mathcal{O}(m \cdot \log(n)) \leq \mathcal{O}(\mathrm{tw} \cdot n \cdot \log(n))$ introduce-edge-nodes are necessary. We can add the $\mathcal{O}(\mathrm{tw} \cdot n)$ bags of the nice cp-tree-decomposition to this number and thus have $\mathcal{O}(\mathrm{tw} \cdot n \cdot \log(n))$ bags in total. ∎

The final attribute we want to show concerns the running time of our construction.

Lemma 5.7: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, the construction above runs in time $\mathcal{O}\left(\mathrm{tw}^2 \cdot (n + m) \cdot \log(n)\right)$.*

Proof. We can find a nice cp-tree-decomposition in $\mathcal{O}(\mathrm{tw}^2 \cdot (n + m))$ using Theorem 5.4. For adding the introduce-edge-nodes we need to traverse the tree of the nice cp-tree-decomposition backwards, which can be done in $\mathcal{O}(\mathrm{tw} \cdot n)$ and check for each added vertex which vertices it neighbours, which can naively be done in $\mathcal{O}(\mathrm{tw})$. For finding the partitions we can use the already given partitions and adapt them in $\mathcal{O}(1)$ as described above for each of the $\mathcal{O}(\mathrm{tw} \cdot n \cdot \log(n))$ bags, if we already know to which clique we want to add the newly introduced vertex. This has been calculated by checking which vertices it neighbours. Together we get a construction time of $\mathcal{O}\left(\mathrm{tw}^2 \cdot (n + m) \cdot \log(n)\right)$. This concludes our proof. ∎

Theorem 5.8: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, an extended cp-tree-decomposition of G with weight* cptw +1*, width* tw *and $\mathcal{O}(\mathrm{tw} \cdot n \cdot \log(n))$ nodes can be computed in time $\mathcal{O}\left(\mathrm{tw}^2 \cdot (n + m) \cdot \log(n)\right)$.*

5.3. Independent Set and Vertex Cover

The first problem we want to consider is Independent Set and its related problems. Here, the task is to find a set of vertices with no edges between the two of them. Since independent sets and cliques are closely related, the structure of each bag given by the clique-partition can be used to improve onto traditional treewidth-based algorithms.

Theorem 5.9: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *Then,* Independent Set *can be solved in $\mathcal{O}\left(2^{\text{cptw}} \cdot \text{tw} \cdot n^2\right)$.*

Proof. As a proof we give an FPT-algorithm with the running time from above. For this, we use, as a base, the traditional algorithm of Cygan et al. [Cyg+15] and adapt it to our parameter. During this process our adaptation are similar to de Berg et al. [Ber+18], who adapt the same algorithm to their parameter, the algorithm gained in this proof corresponds to the algorithm in the paper of Bläsius et al. [BKW23]. We still give this algorithm to show our notation on a simple example. This also allows us to build on the ideas introduced in this algorithm in later proofs.
For this proof we first define how we represent partial solutions before giving the recursive formulas needed to compute those. Afterwards we can bound the number of partial solutions we need to consider and thus estimate the running time.
Similarly to the traditional algorithm, we start with a nice cp-tree-decomposition. We also consider possible partial solutions in the current bag, meaning for each bag $B(t)$ we consider the subsets $S \subseteq B(t)$ and compute the value $c[t, S]$ of a maximum independent set $\hat{S} \subseteq V_t$ with $\hat{S} \cap B(t) = S$ for each of these. This is an independent set on the graph we have visited so far and that, in the bag we currently look at, uses exactly the vertices from S. The current optimal value for the subset S in node t is $c[t, S]$. If S is not independent, then this value is $-\infty$. Thus, we get, at least in the other cases, the following recursive formulas:

For leaf-nodes t we have:

$$c[t, \emptyset] = 0$$

For introduce-node t with child t' and introduced vertex v we have:

$$c[t, S] = \begin{cases} c[t', S] & \text{if } v \notin S, \\ c[t', S - v] + 1 & \text{if } v \in S \end{cases}$$

For forget-node t with child t' and forgotten vertex v we have:

$$c[t, S] = \max\{c[t', S], c[t', S + v]\}$$

For join-node t with children t_1 and t_2 we have:

$$c[t, S] = c[t_1, S] + c[t_2, S] - |S|$$

Having given the recursive formulas the next task is to limit how many partial solutions we need. In contrast to the traditional algorithm we can use the known structure of the bags to our advantage and we thus do not need to look at all possible subsets. It is easy to see that only one vertex from each clique can be chosen in an independent set. Thus, we can consider only partial solutions that pick at most one vertex from each clique. To phrase it differently, for each clique we need to consider the partial solutions gained from choosing each vertex one at a time and the partial solution gained from choosing no vertex. We then have to combine those possible solutions with those of other cliques. Thus, we have per Clique C in total $|C| + 1$ options and we have to multiply those for each clique. This allows us to consider only

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

possible solutions per node t. Similarly to the traditional algorithm we can argue for the total running time: For each bag we can calculate all needed values and test the independence of the set in $\mathcal{O}(n)$ [Cyg+15][4]. Furthermore, the cp-tree-decomposition has a maximum of $\mathcal{O}(\text{tw} \cdot n)$ bags. Thus, we gain the total running time of $\mathcal{O}(2^{\text{cptw}} \cdot \text{tw} \cdot n^2)$ and proved our claim. ∎

We use this result to get algorithms for related problems by making some small modifications. First we consider the problem which additionally assigns each vertex a weight and the asks for the highest weight independent set.

Corollary 5.10: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*, and let $g : V \to \mathbb{R}$ be a weight function for G. Then, Weighted Independent Set can be solved in $\mathcal{O}\left(2^{\text{cptw}} \cdot \text{tw} \cdot n^2\right)$.*

Proof. We now need to only adapt the recursive formula for introduce- and join-nodes t in the algorithm above.[5]

$$c[t, S] = \begin{cases} c[t', S] & \text{if } v \notin S, \\ c[t', S - v] + g(v) & \text{if } v \in S \end{cases}$$
$$c[t, S] = c[t_1, S] + c[t_2, S] - g(S)$$

Since every valid partial solution for a bag in Independent Set is a valid solution in Weighted Independent Set, the number of partial solutions we need to test does not change. The running time does not change either, since the needed weights can be found in $\mathcal{O}(n)$. ∎

After this we can use that Vertex Cover and Independent Set are complements to get an algorithm for another problem.

Corollary 5.11: *Let $G = (V, E)$ be a graph wit a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*. Then, Vertex Cover can be solved in $\mathcal{O}\left(2^{\text{cptw}} \cdot \text{tw} \cdot n^2\right)$.*

Proof. A graph has a vertex cover of size at most k if and only if it has an independent set of size at least $n - k$ [Cyg+15]. Thus, we can calculate an independent set S of size $n - k$ with Theorem 5.9 and then choose the complement $V - S$ of the same as vertex cover. A proof of this connection can be found in the work of Ayad [Aya]. ∎

Finally, we want to consider the variant of Vertex Cover in which we are interested in connected solutions. For this problem we can adapt our algorithm and need to track which solutions are connected.

Corollary 5.12: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*. Then, Connected Vertex Cover can be solved in $2^{\mathcal{O}(\text{cptw})} \cdot n^2$.*

Proof. For this, we can use the same algorithm as for Vertex Cover and thus for Independent Set but we need to also track if the components of the complement of the independent set are connected. This can be done by using equivalence relations and the rank-based-approach which we employ later on for our algorithm solving Steiner Tree to remember which vertices are connected and requiring the final solution to be connected [Ber+18].

In our algorithm we need to access partial solutions, add weights, which can be done with *shft*, and take maxima, which can be done with *union*.

[4] Here, we mostly need to consider whether the old partial solutions are independent and the introduced or removed vertex changes the independence.

[5] We use the same adaptation as in the traditional case given by Cygan et al. [Cyg+15]

Thus, we have expressed our algorithm with the operations of Bodlaender et al. [BCKN15] and we can use the rank-based-approach to limit the number of equivalence relations to $2^{\mathcal{O}(p(B,P))}$ since we select one vertex from each clique and have at most $p(B,P)$ cliques per bag and thus our solutions are of size $p(B,P)$ at most. This gives us at most $2^{\mathcal{O}(p(B,P))} \cdot 2^{\text{cptw}} = 2^{\mathcal{O}(\text{cptw})}$, with $p(B,P) \leq \text{cptw}$ (Lemma 3.6), partial solutions. Using Theorem 2.3 with partial solutions of size of at most $p(B,P) \leq \text{cptw}$ (Lemma 3.6) we can show that using the rank-based-approach does not increase the needed time. We can use Lemma 3.3 and Lemma 5.2 to bound then number of bags by $2^{\text{cptw}} \cdot n$ and then lose the factor of 2^{cptw} in the $\mathcal{O}$-Notation. ∎

5.4. Maximum Induced Forest and Feedback Vertex Set

The next problem we want to consider is Maximum Induced Forest. Again, we will use the structure of the bags to find an algorithm and then adapt it to also deal with different variants of the problem.

Theorem 5.13: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*. Then, Maximum Induced Forest can be solved in $2^{\mathcal{O}(\text{cptw})} \cdot n^2$.*

Proof. Again, we do not give a complete algorithm and instead follow de Berg et al. [Ber+18] and show how we can use the rank-based-approach of Bodlaender et al. [BCKN15] for our parameter. Here, we use the authors algorithms expressed via the rank-based-approach. This, algorithm uses the rank-based-approach in an unorthodox fashion, for a more traditional use of this approach see our algorithm for Steiner Tree (Theorem 7.17). Note that the algorithm of de Berg et al. [Ber+18] uses an extended tree decomposition and thus our adaptation uses an extended cp-tree-decomposition.

For this, we first need to introduce a modification to the problem and then bound the number of vertices selected from each clique, which gives us the running time.

We use the same modification as de Berg et al. [Ber+18] and Bodlaender et al. [BCKN15] to go from maximising connectivity with the rank-based-approach to minimizing it for Maximum Induced Forest.

This is necessary since we can only use the rank-based-approach if our aim is to connect as many parts as possible. For Maximum Induced Forest the task is the opposite, here we want to reduce connectivity as much as possible. For this, we need to make a modification that connects all parts of the forest and allows us to search for a connected solution. We add a special universal vertex v_0 with edges to all other vertices to the graph.

In terms of the cp-tree-decomposition this means adding v_0 to all bags and thus increasing the cp-treewidth by at most $\log(1 + 1) = 1$, since we can add v_0 as a separate clique to the partition of each bag.

The new question asked is whether we can delete some of the edges incident to v_0 to create an induced, connected subgraph that includes v_0. We require this graph to have $k + 1$ vertices and k edges. This gives us a maximum induced forest of size k in the original graph. This modification allows us to find a connected solution by arbitrarily glueing together the trees of the forest into one tree.

The next step after modifying the problem is to use the properties of the cp-tree-decomposition and bound the number of partial solutions. From here on we differ from de Berg et al. [Ber+18]. Each partial solution is a subset of vertices to pick into the maximum induced forest for each

bag. For this, we need the observation that we can pick at most two vertices from each clique, since we would otherwise get a cycle. This gives us similarly to the previous algorithms a upper bound on the number of subsets to consider per node t.

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

This means, since we have $p(t)$ cliques in bag $B(t)$, we need to select at most $2\cdot p(t)\leq 2\cdot p(B,P)$ vertices from each bag. We use the rank-based-approach to keep the number of equivalence relations in $2^{\mathcal{O}(p(B,P))}$. By combining equivalence relations and subsets we get with $p(B,P)\leq\mathrm{cptw}$ (Lemma 3.6):

$$2^{2\cdot\mathrm{cptw}}\cdot 2^{\mathcal{O}(p(B,P))} = 2^{\mathcal{O}(\mathrm{cptw})}$$

Again, we gain the total running time by considering $2^{\mathcal{O}(\mathrm{cptw})}$ partial solutions in $\mathcal{O}(2^{\mathrm{cptw}}\cdot n)$ bags and taking $\mathcal{O}(n)$ time each. For this, we did use Lemma 3.3 to modify the bound on the number of nodes given by Lemma 5.6. Using Theorem 2.3 with partial solutions of size of at most $2\cdot\mathrm{cptw}$ we can show that using the rank-based-approach does not increase the needed time. ∎

We can adapt this algorithm to solve the complement of Maximum Induced Forest.

Corollary 5.14: *Let $G=(V,E)$ be a graph with a given cp-tree-decomposition (T,B,P) of weight* cptw*. Then, Feedback Vertex Set can be solved in $2^{\mathcal{O}(\mathrm{cptw})}\cdot n^2$.*

Proof. We note that Feedback Vertex Set is the complement of Maximum Induced Forest and thus we can calculate a solution F to Maximum Induced Forest of size k and get $V-F$ as a solution of size $n-k$ for Feedback Vertex Set. ∎

As done before we consider the connected variant of Feedback Vertex Set.

Corollary 5.15: *Let $G=(V,E)$ be a graph with a given cp-tree-decomposition (T,B,P) of weight* cptw*. Then, Connected Feedback Vertex Set can be solved in $2^{\mathcal{O}(\mathrm{cptw})}\cdot n^2$.*

Proof. For this, we can use the same algorithm as for the not connected variant of Maximum Induced Forest but we need to also track if the components of the complement of the independent forest are connected. This can be done by using the same equivalence relations which keep the solution cycle free and requiring the final solution to be connected [Ber+18].

Here, we can again use the rank-based-approach to limit the number of equivalence relations to $2^{\mathcal{O}(p(B,P))}$ since we select at most two vertices from each clique and have at most $p(B,P)$ cliques per bag and thus our solutions are of size $2\cdot p(B,P)$ at most. This gives us at most $2^{\mathcal{O}(p(B,P))}\cdot 2^{2\cdot\mathrm{cptw}} = 2^{\mathcal{O}(\mathrm{cptw})}$, using $p(B,P)\leq\mathrm{cptw}$ (Lemma 3.6), partial solutions. ∎

6. The limits of this approach

In the last chapter we found algorithms for multiple problems by bounding the number of vertices we can choose from each clique in a valid partial solution. When attempting to apply this approach to other problems like Dominating Set, Clique and Hamilton Cycle we encounter a major problem. For those problems no such bounds have been found. In this section we prove that this is impossible and the first approach cannot be applied to those problems. In order to find algorithms for more problems, we thus need to adapt our approach.

Additionally, we answer a question posed by Section 4.3, that is whether there are more problems, like 3-Clique Cover, for which cp-treewidth is weaker than treewidth under ETH. Apart for the obvious candidates of 4-Clique Cover, 5-Clique Cover and so on, it is easy to see that all problems for which the graph of each yes-instance can be covered with constantly many cliques are also of this type. For this, proceed as before by first filtering out all instances that need far too many cliques and then have a low parameter.

This results in two questions: First, are there any other problems, for which each yes-instance can be covered with only constantly many cliques? And second, are there problems for which cp-treewidth is weaker than treewidth under ETH of a different type?

While we will leave the first question upto anybody that wants to come up with a new problem, we give proofs for some problems that they cannot be solved in time $2^{o(2^{\text{cptw}})} \cdot n^{\mathcal{O}(1)}$ when parametrizing only by cp-treewidth. We prove that, under ETH, cp-treewidth is weaker by $f(x) = 2^{\sqrt{x}}$ than treewidth for Dominating Set, Steiner Tree, Max Cut, Clique and k-Clique Cover. Furthermore, we show that, under ETH, cp-treewidth is weaker by $f(x) = 2^x$ than treewidth for Hamilton Cycle, Hamilton Path and weaker by $f(x) = 2^{(1-\varepsilon)\cdot x}$ for Colour. This can be found in Corollary 6.12 and Corollary 6.19. Thus, we prove in this section that there are more problems that separate cp-treewidth from treewidth and that this is the reason for our struggles in finding more algorithms. The general idea for those problems is showing that for some graph class with cp-treewidth logarithmic in vertex count no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm[1] can be found under ETH and then using this in a proof by contradiction. For this, we consider chordal and co-planar graphs.

6.1. Finding problems where clique-partitioned treewidth is weaker than treewidth with the use of chordal graphs

The first graph family we use to find problems for which cp-treewidth is weaker than treewidth are chordal graphs. We first give their definition and prove some results relating to cp-treewidth before proceeding with our comparison of cp-treewidth and treewidth.

Definition and attributes. We call a graph $G = (V, E)$ chordal if no cycle of length greater than three is induced [AJKL22], this means that each cycle of G with length greater than three contains a chord [BP83]. A chord is a edge not belonging to the cycle but connecting two non-adjacent vertices on the cycle.

[1]The uncommon notation is due to the fact that we will replace the n in the exponent with cptw in the next step.

Chordal graphs are of interest when exploring clique-partitioned treewidth since their minimal separators are cliques [Dir61] and they admit a tree decomposition in which every bag induces a clique [Dvo15].

Using this we can state the following lemmas. We do not give proofs, since it can easily be seen that they hold if one considers the above tree decomposition with a clique-partition in which the whole bag is assigned to one clique.

Lemma 6.1: *Let $G = (V, E)$ be a chordal graph. Then, a cp-tree-decomposition of G with weight* $\log(n + 1)$ *exists.*

Lemma 6.2: *Let $G = (V, E)$ be a chordal graph with the size of the largest clique in G being c. Then, G has cp-treewidth* $\log(c + 1)$.

Lemma 6.3: *Let $G = (V, E)$ be a chordal graph with treewidth* tw. *Then, G has cp-treewidth* $\log(\text{tw} + 2)$.

We can use chordal graphs in proving that cp-treewidth is weaker than treewidth for some problems. We first rule out fast algorithms on chordal graphs for each problem and then adapt this to general graphs for all of the above problems. The first problem we consider is Dominating Set.

Lemma 6.4: *Let $G = (V, E)$ be a chordal graph. If ETH holds, then no algorithm solving Dominating Set in time $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from Vertex Cover on general graphs to Dominating Set on chordal graphs. Here, we can use the already known polynomial reduction [CP84] that consists of finding the incident graph $I = (V \cup E, F)$ and adding edges to ensure V induces a complete graph. This reduction is fast enough and changes the instance size from $n + m$ to $(n + m) + (n^2 + 2 \cdot m) = n^2 + n + 3 \cdot m$, which is quadratic. The resulting graph is a split graph which is a graph that can be partitioned into a clique and an independent set. Split graphs are chordal [CP84]. Since no $2^{o(n+m)} \cdot n^{\mathcal{O}(1)}$ algorithm for Vertex Cover on general graphs can exist under ETH [Cyg+15], we can conclude that no $2^{o(\sqrt{n+m})} \cdot n^{\mathcal{O}(1)}$ algorithm for Dominating Set on chordal graphs can exist under ETH due to the reduction. ∎

The next problem we consider is Steiner Tree.

Lemma 6.5: *Let $G = (V, E)$ be a chordal graph. If ETH holds, then no algorithm solving Steiner Tree in time $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from 3-Colour on general graphs to Steiner Tree on chordal graphs. This reduction will be completed in multiple steps and increases the instance size quadraticly. We can use Karps [Kar10] reduction from 3-Colour to Exact Cover which is linear and then reduce to 3D-Matching in a quadratic reduction [Kar10]. Then, we use a proof by restriction (which is obviously linear) and end at 3-Exact Cover [McC14]. The final step is to reduce to Steiner Tree, here we use the reduction of Illuri et al. [IRS16] which is also linear in terms of size. The resulting graph is in a subclass of split graphs [IRS16] and thus chordal [CP84]. Since no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm for 3-Colour on general graphs can exist under ETH [Cyg+15], we can conclude that no $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ algorithm for Steiner Tree on chordal graphs can exist under ETH due to the reduction. ∎

After this, we consider Hamilton Cycle.

Lemma 6.6: *Let $G = (V, E)$ be a chordal graph. If ETH holds, then no algorithm solving* HAMILTON CYCLE *in time $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from SAT to HAMILTON CYCLE on chordal graphs. Here, we can use the already known polynomial reductions from SAT to a restricted version of SAT [Lic82], then to HAMILTON CYCLE on chordal bipartite graphs[2] [Mül96] and then to HAMILTON CYCLE on strongly chordal split graphs[3] [Mül96]. All those reductions are fast enough and only increase the input size linearly. The resulting graph is chordal [CP84]. Since by definition no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm for SAT with n variables can exist under ETH, we can conclude that no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ algorithm for HAMILTON CYCLE on chordal graphs can exist under ETH due to the reduction. ∎

We can adapt this result to HAMILTON PATH. Here, we can use the reduction by Müller [Mül96] from HAMILTON CYCLE to HAMILTON PATH on strongly chordal split graphs, which is trivially linear, and get the following corollary.

Corollary 6.7: *Let $G = (V, E)$ be a chordal graph. If ETH holds, then no algorithm solving* HAMILTON PATH *in time $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ exists.*

As the last problem for chordal graphs we consider MAX CUT.

Lemma 6.8: *Let $G = (V, E)$ be a chordal graph. If ETH holds, then no algorithm solving* MAX CUT *in time $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from MAX CUT on general graphs to MAX CUT on chordal graphs. Here, we can use the already known polynomial reduction [BJ00] that is fast enough and changes the instance size from $n + m$ to $(n + n^2 - m) + (n^2 + 2 \cdot m) = 2 \cdot n^2 + n + m$, which is quadratic. The resulting graph is a split graph [BJ00] and thus chordal [CP84]. Since no $2^{o(n+m)} \cdot n^{O(1)}$ algorithm for MAX CUT on general graphs can exist under ETH[4], we can conclude that no $2^{o(\sqrt{n+m})} \cdot n^{\mathcal{O}(1)}$ algorithm for MAX CUT on chordal graphs can exist under ETH due to the reduction. ∎

We can use these results to prove that cp-treewidth is weaker than treewidth for DOMINATING SET, STEINER TREE, HAMILTON CYCLE, HAMILTON PATH and MAX CUT. Here, we combine two versions, for general graphs and chordal graphs, into one theorem.

Theorem 6.9: *Let $G = (V, E)$ be a (chordal) graph of cp-treewidth* cptw*. If ETH holds, then for* HAMILTON CYCLE *and* HAMILTON PATH *no algorithm solving those problems parametrized by cp-treewidth* cptw *in time $2^{o\left(2^{\text{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. Since $\text{cptw} \leq \log(n + 1)$ in chordal graphs (Lemma 6.1), we have $n \geq 2^{\text{cptw}} + 1$ and thus Lemma 6.6 and Corollary 6.7 can be rewritten to no $2^{o\left(2^{\text{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm existing for chordal graphs.

We can then generalize this result to all graphs. Due to the fact that, any $2^{o\left(2^{\text{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm for general graphs would also be a $2^{o\left(2^{\text{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm in chordal graphs, which contradicts the above statement, we can rule out such an algorithm. ∎

[2] A graph is chordal bipartite if each of its chordless cycles has length four [Mül96]. Those graphs can be not chordal [Mül96].

[3] A graph is a strongly chordal split graph if it is a split graph and each cycle with even number of vertices has an odd chord, i.e., an edge which connects two vertices having odd distance in the cycle [Mül96]. Those graphs are obviously strict and thus chordal [CP84].

[4] Observe that all of Karps [Kar10] reductions from 3-SAT to MAX CUT which he uses to prove the problem $\mathcal{NP}$-complete increase the size at most linearly.

We can state a similar result for the other problems when considering the square root in each step.

Corollary 6.10: *Let $G = (V, E)$ be a (chordal) graph of cp-treewidth* cptw. *If ETH holds, then for* Dominating Set, Steiner Tree *and* Max Cut *no algorithm solving those problems parametrized by cp-treewidth* cptw *in time* $2^{o\left(2^{\sqrt{\text{cptw}}}\right)} \cdot n^{\mathcal{O}(1)}$ *exists.*

Obviously, these results can be extended to closely related problems like r-Dominating Set or Weighted Max Cut by using the reductions of Section 4.2.

Corollary 6.11: *Let $G = (V, E)$ be a (chordal) graph of cp-treewidth* cptw. *If ETH holds, then for r-*Dominating Set,Connected Dominating Set, Weighted Steiner Tree *and* Weighted Max Cut *no algorithm solving those problems parametrized by cp-treewidth* cptw *in time* $2^{o\left(2^{\sqrt{\text{cptw}}}\right)} \cdot n^{\mathcal{O}(1)}$ *exists.*

A short summary is given by this corollary.

Corollary 6.12: *Under ETH, cp-treewidth is weaker by $f(x) = 2^{\sqrt{x}}$ than treewidth for* Dominating Set, *r-*Dominating Set,Connected Dominating Set, Steiner Tree Weighted Steiner Tree, Max Cut *and* Weighted Max Cut. *Under ETH, cp-treewidth is weaker by $f(x) = 2^x$ than treewidth for* Hamilton Cycle *and* Hamilton Path.

Proof. We use Theorem 6.9, Corollary 6.10 and Corollary 6.11 together with the traditional algorithms of Cygan et al. [Cyg+15], Bodlaender et al. [BCKN15] and Bodlaender and Jansen [BJ94]. ∎

Although some other problems can be solved easily on chordal graphs, we could use the same idea with a different family to prove similar results. This, is done in the next section.

6.2. Finding problems where clique-partitioned treewidth is weaker than treewidth with the use of co-planar graphs

Another problem of interest is Clique, since it could be considered closely related to 3-Clique Cover and thus comes to mind as a candidate for problems where cp-treewidth could be weaker than treewidth. As before we want to establish the fact that certain algorithms can not be found for Clique. Since we can embed a clique into any kind of graph, not all yes-instances can be covered with constantly many cliques. Furthermore, Clique can be solved on chordal graphs by finding the largest bag, if we are given an optimal tree decomposition. This is a polynomial algorithm and finds the largest clique since each bag is a clique [Dvo15] and each clique is contained in at least one bag (Lemma 2.4). Thus, we need to find a different family of graphs.

We want to consider the family of co-planar graphs to prove that cp-treewidth is weaker by $f(x) = 2^{\sqrt{x}}$ than treewidth for Clique and, if possible, other problems. Those are the graphs whose complements are planar, meaning the complements are $K_{3,3}$- and K_5-minor-free (Wagner's Theorem [Wag37]) as well as $K_{3,3}$- and K_5-subdivision-free (Kuratowski's Theorem [Kur30]). Those graphs can be covered with four cliques since their complements can be coloured with four colours due to the four colour theorem [AH76]. Due to this, we can state Lemma 6.13 about their cp-treewidth.

Lemma 6.13: *Let $G = (V, E)$ be a co-planar graph. Then, G has cp-treewidth of at most $\mathcal{O}(\log(n))$.*

Proof. For this, consider the cp-tree-decomposition where every vertex is assigned to one single bag. Then, this bag can be covered with four cliques as seen above and thus the claimed cp-treewidth can be achieved. ∎

Since Clique, Clique Cover and Colour are NP-complete on co-planar graphs[5] this family can be used to show that cp-treewidth is weaker by $f(x) = 2^{(1-\varepsilon)\cdot x}$, respectively $f(x) = 2^{\sqrt{x}}$, than treewidth for Clique, k-Clique Cover and Colour. We proceed, as done for the other problems, by proving that, under ETH, no $2^{o(n)} \cdot n^{\mathcal{O}(1)}$, respectively $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$, algorithm on co-planar graphs exists and then use the known relations between cp-treewidth and number of vertices to prove Theorem 6.17 and Corollary 6.18.

We consider those problems one by one and begin with Clique.

Lemma 6.14: *Let $G = (V, E)$ be a co-planar graph. If ETH holds, then no algorithm solving Clique in time $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this, we give a reduction from Vertex Cover on general graphs to Vertex Cover on planar graphs. Here, we can use the already known polynomial reduction [GJS76] that embeds the graph into the plane by replacing each crossing with a gadget of constant size. Due to the fact that a graph can be drawn with at most m^2 crossings [Fal+21][6], the reduction is quadratic. We then reduce from Vertex Cover to Independent Set as seen before (e. g. in Corollary 5.11). This clearly is linear and keeps the graph planar. By transitioning to the complement graph we get a linear reduction to Clique on co-planar graphs, since those are the complement graphs of planar graphs and Clique is the complement of Independent Set [GJ90]. Since no $2^{o(n+m)} \cdot n^{\mathcal{O}(1)}$ algorithm for Vertex Cover on general graphs can exist under ETH [Cyg+15], we can conclude that no $2^{o(\sqrt{n+m})} \cdot n^{\mathcal{O}(1)}$ algorithm for Clique on co-planar graphs can exist under ETH due to the reductions. ∎

The next problem is Clique Cover.

Lemma 6.15: *Let $G = (V, E)$ be a co-planar graph. If ETH holds, then no algorithm solving Clique Cover in time $2^{o(\sqrt{n})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from 3-Colour on general graphs to 3-Colour on planar graphs. Here, we can use the already known polynomial reduction [Lee14] that embeds the graph into the plane by replacing each crossing with a gadget of constant size. Due to this, the reduction is quadratic[7]. We then reduce from 3-Colour to Colour by a trivial reduction. By transitioning to the complement graph we get a linear reduction to Clique Cover on co-planar graphs, since those are the complement graphs of planar graphs and Clique Cover is the complement of Colour [Kar10]. Since no $2^{o(n+m)} \cdot n^{\mathcal{O}(1)}$ algorithm for 3-Colour on general graphs can exist under ETH [Cyg+15], we can conclude that no $2^{o(\sqrt{n+m})} \cdot n^{\mathcal{O}(1)}$ algorithm for Clique Cover on co-planar graphs can exist under ETH due to the reductions. ∎

[5]Consider the reductions and citations in the proofs below.

[6]For this upper bound the authors require what they call a good drawing. This means that no edge can cross itself, each pair of edges has at most one point in common (including endpoints) and no more than two edges can cross at any point [Fal+21]. Such a drawing can be found by modifying an existing drawing. Here, we remove unnecessary loops from edges that cross them selves, untangle pairs of edges that cross multiple times and remove points where more that two edges cross by moving one edges by an infinitesimal amount.

[7]See the proof of Lemma 6.14 for a more detailed analysis.

And finally, we consider Colour.

Lemma 6.16: *Let $G = (V, E)$ be a co-planar graph. If ETH holds, then no algorithm solving Colour in time $2^{o(n)} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. To prove this we give a reduction from 3,3-Sat to Clique Partition on planar graphs. Here, we can use the already known polynomial reduction [Cer+08] that replaces variables and clauses with a gadgets of constant size. Due to this, the reduction is linear. This reduction uses 3,3-Sat as starting point, which is the restricted version of 3-Sat where each variable appears in at most three clauses and at most once negated. We can find a linear reduction[8] from 3-Sat to this problem [Lyu08][9]. We then reduce from Clique Partition to Clique Cover using the fact that every clique partition is a clique cover. This clearly is linear and keeps the graph planar. By transitioning to the complement graph we get a linear reduction to Colour on co-planar graphs, since those are the complement graphs of planar graphs and Colour is the complement of Clique Cover [Kar10]. Since by definition no $2^{o(n+m)} \cdot n^{\mathcal{O}(1)}$ algorithm for 3-SAT on general graphs can exist under ETH, we can conclude that no $2^{o(n+m)} \cdot n^{\mathcal{O}(1)}$ algorithm for Clique on co-planar graphs can exist under ETH due to the reductions. ∎

As before, we can use this to prove that cp-treewidth is weaker than treewidth for those problems. Here, we combine two versions, for general graphs and co-planar graphs, into one theorem.

Theorem 6.17: *Let $G = (V, E)$ be a (co-planar) graph of cp-treewidth* cptw*. If ETH holds, then no algorithm solving Colour parametrized by cp-treewidth* cptw *in time $2^{o\left(2^{\mathrm{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. Since $\mathrm{cptw} \in \mathcal{O}(\log(n))$ in co-planar graphs (Lemma 6.13), we have $n \in \Omega(2^{\mathrm{cptw}})$ and thus Lemma 6.16 can be rewritten to no $2^{o\left(2^{\mathrm{cptw}}\right)} \cdot n^{O(1)}$ algorithm existing for co-planar graphs.

We can generalize this result to all graphs. Due to the fact that, any $2^{o\left(2^{\mathrm{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm for general graphs would also be a $2^{o\left(2^{\mathrm{cptw}}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm in co-planar graphs, which contradicts the above statement, we can rule out such an algorithm. ∎

Again, we state the same result for those problems for which we only found quadratic reductions (see the Lemma 6.14, Lemma 6.15[10]).

Corollary 6.18: *Let $G = (V, E)$ be a (co-planar) graph of cp-treewidth* cptw*. If ETH holds, then for Clique and k-Clique Cover no algorithm solving those problems parametrized by cp-treewidth* cptw *in time $2^{o\left(2^{\sqrt{\mathrm{cptw}}}\right)} \cdot n^{\mathcal{O}(1)}$ exists.*

Finally, we can extract the desired results as a corollary.

Corollary 6.19: *Under ETH, cp-treewidth is weaker by $f(x) = 2^{\sqrt{x}}$ than treewidth for Clique and k-Clique Cover. Under ETH, cp-treewidth is weaker by $f(x) = 2^{(1-\varepsilon)\cdot x}$ than treewidth for Colour, with a given $\varepsilon > 0$.*

[8] This reduction is linear since there are at most 3 times as many appearances of all variables together as the number of clauses. Thus, at most this number of variables and clauses is added.

[9] This reduction uses a slightly varying definition which instead of limiting the number of negated occurrences to one forbids each literal to appear more than twice. Transitioning from one definition to the other is no problem since we can replace any variables where there are two negated and one non-negated occurrence with its negation.

[10] Since Clique Cover is not yet known to be in FPT, we restrict ourselves to k-Clique Cover, which is in FPT (consider similar algorithms as in Lemma 4.8). The needed reduction is obvious.

Proof. We use Theorem 6.17 and Corollary 6.18 together with the traditional algorithms of Corollary 4.10 and Marx [Mar20a] as well as the trivial algorithm for Clique.

For Colour the traditional algorithm has running time $2^{o(\mathrm{tw}\cdot\log(\mathrm{tw}))}\cdot n^{\mathcal{O}(1)}$ and we ruled out any $2^{o\left(2^{\mathrm{cptw}}\right)}\cdot n^{O(1)}$ algorithms (Theorem 6.17). Thus, we have $g(x)=2^{o(x\cdot\log(x))}$ and we can choose $f(x)=2^{(1-\varepsilon)\cdot x}$ in our definition. This is since $g(o(f(x)))=2^{o\left(2^{(1-\varepsilon)\cdot x}\cdot\log\left(2^{(1-\varepsilon)\cdot x}\right)\right)}=2^{o\left(2^{(1-\varepsilon)\cdot x}\cdot(1-\varepsilon)\cdot x\right)}\in 2^{o(2^x)}$. ∎

Note that this idea can be generalized by finding problems that are $\mathcal{NP}$-complete on some graph class, adapting the known polynomial reductions to show no $2^{o\left(2^{\mathrm{cptw}}\right)}\cdot n^{\mathcal{O}(1)}$ algorithm[11] can exist under ETH and then using this to show that cp-treewidth is weaker than treewidth for this problem. Furthermore, these discoveries support the need for an additional factor in the exponent for those problems. As shown above, if we use no additional factor we cannot find a fast algorithm and thus need some factor that is not bounded by a polynomial of cp-treewidth. In Section 7.5 we show for a selection of problems some evidence supporting the fact that this factor needs to be $\Delta(G,T,P)$ or something similar.

[11] Or something similar, if no linear reduction can be found.

7. Building algorithms with an additional parameter

In the last section we did see that finding algorithms solely based on cp-treewidth is impossible for some problems. In this section we want to find algorithms for some of those problems anyway. To achieve this we need an additional parameter that can be exponential in cp-treewidth but that allows us to find algorithms if both cp-treewidth and this parameter are small. As the next step, we introduce two such parameters and then give those algorithms. Here, we use the extended cp-tree-decomposition, we did already introduce, in a more traditional fashion. The final step is to show for a selection of problems that we cannot use smaller parameters with our current approach.

7.1. Attributes of clique-partitioned tree decompositions

In this section we describe important attributes of cp-tree-decompositions that we then can use in our algorithms as supporting parameters. This means that we parametrize by cp-treewidth as well as one of these parameters. This allows us to find FPT-algorithms for problems for which we can prove that we could not find fast algorithms otherwise. We first introduce those parameters and then analyse a graph family to get a better understanding of these parameters. The final step is to understand how those parameters behave in normalizations.

Clique-degree. We want to find an upper bound on the number of cliques in G that neighbour a given clique. We first give a helpful definition and then define the bound. For a graph $G = (V, E)$ with a given cp-tree-decomposition (T, B, P) we define the *clique-base-set* $\mathcal{C}(G, P)$ as the union of the sets of all cliques in each partition. For a formal definition, we use $\mathcal{C}(G, P) = \bigcup_{t \in V(T)} \mathcal{P}(t)$ and for a clique C we call all vertices in $G - C$ that are adjacent to a vertex in C its *neighbourhood* and use $N(C)$ for it. We then define the *degree* of a clique C as the minimal number of cliques from $\mathcal{C}(G, P)$ needed to cover all vertices in $N(C)$. We next define the *clique-degree* of a node t as the maximal degree of a clique in t. Finally, the *clique-degree* of the cp-tree-decomposition (T, B, P) is the maximal clique-degree of a node in G. We will denote this value by $\Delta(G, T, P)$ for graph G and cp-tree-decomposition (T, B, P).

The clique-degree is used in this work together with the cp-treewidth and thus is defined as a property of a given cp-tree-decomposition instead of a parameter, where we would minimize over all cp-tree-decompositions in the end. When parametrizing with cp-treewidth and clique-degree the aim is to find a cp-tree-decomposition that minimizes both parameters. To be more precise: For the algorithms presented here, the running time is optimal, when the product of the two parameters is minimal.

Computing the clique-degree. Since we introduced the clique-degree in the last paragraph, it is of interest how this value can be computed. Let us first rephrase the problem. We are given the neighbourhood of a clique, which is a set of vertices, as well as the intersections between the cliques of $\mathcal{C}(G, P)$ and this neighbourhood, which is a family of subsets of the

neighbourhood. The task is then to find the minimal number of subsets that cover the neighbourhood. Due to this rephrasing it is obvious that calculating $\Delta(G, T, P)$ for given graph and cp-tree-decomposition is equivalent to solving the Set Cover problem. Since this problem is $\mathcal{NP}$-complete [Kar10], finding $\Delta(G, T, P)$ is as well. To compute $\Delta(G, T, P)$ we can thus use Set Cover-solvers and approximations.

For Set Cover the following results have been found: It can be approximated with a greedy algorithm that always picks the set that covers the most yet uncovered elements [You08]. This greedy algorithm has an approximation factor of $\mathcal{O}(\log(|U|))$ [You08], where U is the universe, which is optimal unless $\mathcal{NP}$ has slightly superpolynomial time algorithms [Fei98]. Further analysis of the algorithm allows us to reduce the approximation factor to $\mathcal{O}(\log(d))$ (mentioned as known algorithm in [GHY93]), where d is the size of the largest set in the family. This is no improvement in the worst case, since the largest set might be almost as large as the universe, but will ease our analysis. This algorithm has a running time in $\mathcal{O}(l \cdot \log(l))$, where l is the number of subsets (mentioned as known algorithm in [GHY93]).

Thus, we can calculate $\Delta(G, T, P)$ either in exponential time or use the greedy algorithm with a $\mathcal{O}(\log(d))$ approximation. Since the size of the largest subset, which is a clique in our problem, is smaller than the treewidth plus one (see Lemma 2.4) and we have $\mathrm{tw} \leq 2^{\mathrm{cptw}}$ (Lemma 3.3), we get an approximation factor of $\mathcal{O}(\log(\mathrm{tw})) \subseteq \mathcal{O}(\log(2^{\mathrm{cptw}})) = \mathcal{O}(\mathrm{cptw})$. Since the family of subsets is the set of all cliques in $\mathcal{C}(G, P)$, this means that the algorithm has running time of $\mathcal{O}(\mathrm{cptw} \cdot n \cdot \log(\mathrm{cptw} \cdot n))$. We can bound the size of $\mathcal{C}(G, P)$ by $p(B, P) \cdot n \leq \mathrm{cptw} \cdot n$ (Lemma 3.6), since we have at most n nodes in a non-redundant cp-tree-decomposition[1] and each partition consists of at most $p(B, P)$ cliques.

We can summarize our results in two lemmas:

Lemma 7.1: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P). Then, calculating $\Delta(G, T, P)$ is $\mathcal{NP}$-complete.*

Lemma 7.2: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P). Then, we can calculate an $\mathcal{O}(\mathrm{cptw})$-approximation for $\Delta(G, T, P)$ in $\mathcal{O}(\mathrm{cptw} \cdot n \cdot \log(\mathrm{cptw} \cdot n))$ time.*

Comparing the clique-degrees. Since de Berg et al. [Ber+18] also defined a degree related parameter, we now give their definition. The authors define the BBKMZ-degree[2] $\Delta_{BBKMZ}(\mathcal{P})$ of a κ-partition $\mathcal{P}$ as the maximum degree of the graph where each partition class of $\mathcal{P}$ is contracted into one vertex. While they assumed their parameter to be constant most of the time due to the specific graph class they focus on and we generally consider it in our analysis, those parameters are used in the same circumstances and same algorithms. It is easy to see that those two are corresponding concepts for the two primary parameters. Thus, we want to compare those two parameters. First, observe that both of them depend on the specific clique-partition in use and thus one parameter can be made smaller compared to the other by choosing favourable and unfavourable partitions. We thus establish some comparisons between the partitions to ensure a fair comparison.

We begin by considering the minimum over all partitions.

[1] If we normalize our cp-tree-decompositions they become redundant and the running time would increase accordingly. We show in Lemma 7.7 and Corollary 8.7 that normalizing does not increase $\Delta(G, T, P)$ and thus we can calculate the value for the original cp-tree-decomposition and use it as an upper bound for the normalized cp-tree-decomposition. This upper bound might not always be tight but generally should be close.

[2] We adapted the name to show its origin.

Lemma 7.3: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of minimal clique-degree $\Delta(G, T, P)$ and a given BBKMZ-tree-decomposition of optimal BBKMZ-degree $\Delta_{BBKMZ}(\mathcal{P})$ with its κ-partition $\mathcal{P}$. Then, $\Delta(G, T, P) \leq \Delta_{BBKMZ}(\mathcal{P})$ for $\kappa = 1$ and $\Delta(G, T, P) \in \mathcal{O}(\Delta_{BBKMZ}(\mathcal{P}))$ for $\kappa > 1$.*

Proof. The BBKMZ-degree also needs to cover all vertices of the neighbourhood due to the fact that every vertex of G is assigned to one clique by the partition. Since we can use the global partition $\mathcal{P}$ as a local partition, we can find a cp-tree-decomposition that has the same cliques available as for the BBKMZ-degree. The minimal value of all clique-degrees is lower than this and thus lower than the minimal BBKMZ-degree for $\kappa = 1$. For $\kappa > 1$, multiple cliques are contracted to one vertex and thus the BBKMZ-degree can become lower but will always be within the constant factor of κ from the version of $\kappa = 1$. Thus, $\Delta(G, T, P) \in \mathcal{O}(\Delta_{BBKMZ}(\mathcal{P}))$ holds. ∎

We can prove in similar fashion that the same holds if we use the global partition as a local one.

Lemma 7.4: *Let $G = (V, E)$ be a graph with a given BBKMZ-tree-decomposition and its κ-partition $\mathcal{P}$ of BBKMZ-degree $\Delta_{BBKMZ}(\mathcal{P})$. Then, let (T, B, P) be the cp-tree-decomposition of clique-degree $\Delta(G, T, P)$ that uses the same bags as the BBKMZ-tree-decomposition and uses the global partition as the partition for each bag. Then, $\Delta(G, T, P) \leq \Delta_{BBKMZ}(\mathcal{P})$ for $\kappa = 1$ and $\Delta(G, T, P) \in \mathcal{O}(\Delta_{BBKMZ}(\mathcal{P}))$ for $\kappa > 1$.*

For some algorithms we need another parameter which we introduce in the next paragraph.

Neighbourhood index. In a graph G with a cp-tree-decomposition (T, B, P) we define the *neighbourhood index* of a clique C in node t as the number vertices in C that have a neighbour in V outside of the clique. Resulting from this we define the *neighbourhood index* of a bag t as the maximum neighbourhood index of a clique in $\mathcal{P}_t$ and the *neighbourhood index* of G and (T, B, P) as the maximal neighbourhood index of a node $t \in V(T)$. We use $\delta(G, T, B, P)$ for the neighbourhood index of G and (T, B, P).

7.1.1. Star-graphs

We now introduce the family of star-shaped graphs and use them to get a better understanding of the clique-degree. Roughly, this family consists of one central and many outer cliques and each outer clique is connected only to the central clique. This is intentionally vague, since we use different variations of such graphs, and is explained in more detail when used. Here, we introduce the basic variant and later on we use some small variations in how the outer cliques are connected to the central clique.

Thus, let us consider a star-shaped graph G of $c + 1$ cliques. Here, one clique of size c is in the center and the other c cliques (of size one) are connected by one edge to one vertex of the central clique each using a different vertex. We call graphs of this basic family c-stars. An example graph for $c = 14$ can be seen in Figure 7.1. This graph family is of interest since it helps us get a clearer understanding of $\Delta(G, T, P)$ and its relation to cp-treewidth. For this, consider the following lemma.

Lemma 7.5: *Let $G = (V, E)$ be a c-star. Then, G has cp-treewidth $\log(c + 1)$ and each cp-tree-decomposition (T, B, P) has clique-degree of c.*

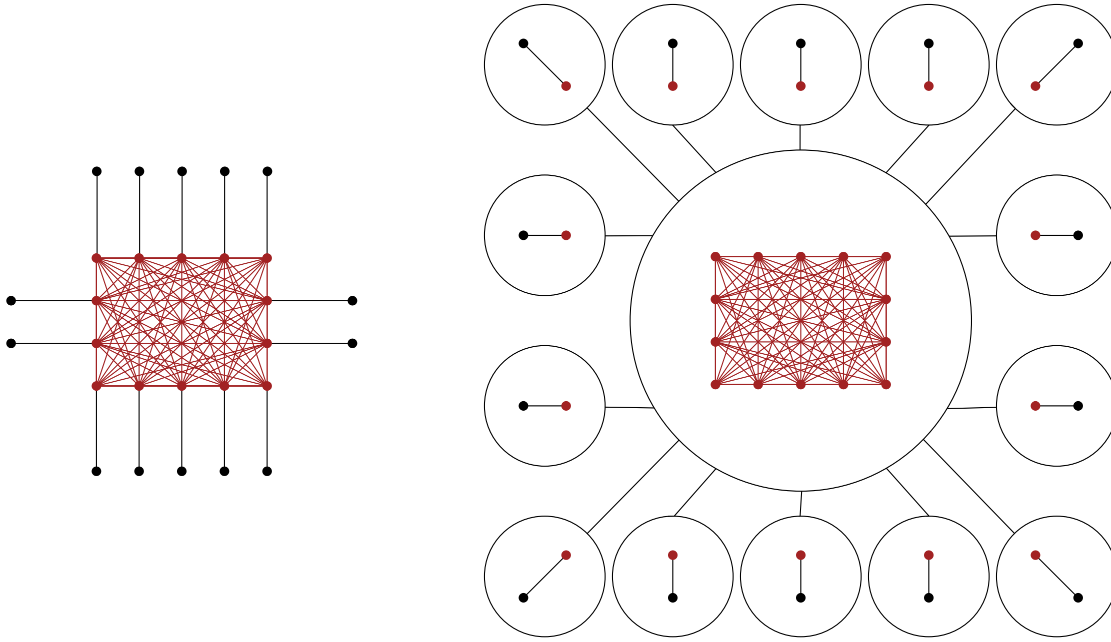


Figure 7.1.: A graph G with a cp-tree-decomposition (T, B, P) of optimal weight, which is given on the right, that has $\Delta(G, T, P)$ larger than its weight. Here, the red vertices form a clique of size $c = 14$ and the c black vertices form c cliques of size one.

Proof. A cp-tree-decomposition of weight $\text{cptw} = \log(c + 1)$ would be to group the center clique into one bag and for each edge between two cliques have a bag with both endnodes. This clearly is optimal due to Lemma 2.4. Since the outer vertices are not connected, the neighbourhood of the central clique cannot be covered with fewer than c cliques for each cp-tree-decomposition. Needing more than c cliques is also obviously impossible, since we search for the minimal number of cliques. Thus, the clique-degree is c. ∎

This lemma shows that there is an infinite family of graphs whose clique-degree is exponentially larger than its cp-treewidth. This means that bounding the clique-degree by a polynomial of cp-treewidth is impossible. It is easy to see that using the same family we can obviously get the same results for the neighbourhood index.

Corollary 7.6: *Let $G = (V, E)$ be a c-star. Then, G has cp-treewidth $\log(c + 1)$ and each cp-tree-decomposition (T, B, P) has neighbourhood index of c.*

7.1.2. Normalizations and additional parameters

To describe our algorithms we use normalized cp-tree-decompositions. We have already proven that the weight stays relatively low when normalizing the cp-tree-decomposition. Since we also use $\Delta(G, T, P)$ in many different algorithms and its definition depends on the used cp-tree-decomposition, we also need to show that it does not increase.

Lemma 7.7: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of clique-degree $\Delta(G, T, P)$. Then, the nice/extended cp-tree-decomposition (T', B', P') of Theorem 5.4/ Theorem 5.8 has clique-degree $\Delta(G, T', P') \leq \Delta(G, T, P)$.*

Proof. For this, observe that each normalization only adds additional bags and keeps the original ones. Due to this, $\mathcal{C}(G, P)$ is a subset of $\mathcal{C}(G, P')$ and we can choose the same cliques as before to cover each neighbourhood. Since some new cliques could be added to the set, it is possible that the value decreases. ∎

We can show the same for $\delta(G, T, B, P)$.

Lemma 7.8: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of neighbourhood index $\delta(G, T, B, P)$. Then, the nice/extended cp-tree-decomposition (T', B', P') of Theorem 5.4/ Theorem 5.8 has neighbourhood index $\delta(G, T', B', P') \leq \delta(G, T, B, P)$.*

Proof. For this, observe that each normalization only adds bags whose cliques are subsets of the original cliques. Due to this, less vertices can have neighbours outside of the clique. ∎

7.2. Dominating Set

For Dominating Set we want to again find a FPT-algorithm by reducing the number of partial solutions. As before we use the traditional algorithm given in the book of Cygan et al. [Cyg+15] as a starting point and adapt it to our parameter. We again have similar adaptations as de Berg et al. [Ber+18]. Since we make some slight changes in how we define the colouring, we first give our definitions and explain our dynamic program, then show in Lemma 7.9 that our adaptations are valid, before proceeding by showing how many partial solutions we need to consider and finally prove our running time.

Again, we we consider a graph $G = (V, E)$ with a given cp-tree-decomposition (T, B, P).

Colouring. We now need more than the two options (in the set or not in the set) we had for each vertex in Independent Set.

Thus, we introduce a colouring $f : B(t) \rightarrow \{\text{black}, \text{gray}, \text{white}\}$ of all vertices in bag $B(t)$. A black vertex is part of the partial solution of G_t. A white vertex is dominated by the partial solution and thus is no part of it. A grey vertex is neither in the partial solution nor dominated by it. For these vertices it is still an open question, whether they will be black or white later on.

To find the above colouring we need to just choose in our partial solutions which vertices we want to colour black, since the white vertices are directly given as the neighbours of the black ones. The remaining vertices can be coloured gray.

With the Definition of our colouring given, we can proceed with our dynamic programming algorithm.

Dynamic program. We start with an extended cp-tree-decomposition, which is a nice cp-tree-decomposition extended by introduce-edge-nodes and we consider partial solutions in a bag.

We use $c[t, f]$ for the size of a minimal set $D \subseteq V_t$ such that $D \cap B(t)$ corresponds to all black vertices in the bag of node t, those vertices are considered our partial solution, and that D dominates all but the grey vertices. If there is no such set for a colouring f and node t we have $c[t, f] = +\infty$.

In all other cases we gain the following recursive formulas:

For leaf-node t we have:

$$c[t, \emptyset] = 0$$

For introduce-node t with child t' and introduced vertex v we have (as a reminder: we introduce no edges with this node):

$$c[t, f] = \begin{cases} +\infty & \text{if } f(v) = \text{white}, \\ c[t', f|_{B(t')}] & \text{if } f(v) = \text{gray}, \\ c[t', f|_{B(t')}] + 1 & \text{if } f(v) = \text{black} \end{cases}$$

For introduce-edge-node t with child t' and introduced edge uv we have:

$$c[t, f] = \begin{cases} \min\{c[t', f_{v\to\text{gray}}], c[t', f]\} & \text{if } f(u) = \text{black} \wedge f(v) = \text{white}, \\ \min\{c[t', f_{u\to\text{gray}}], c[t', f]\} & \text{if } f(u) = \text{white} \wedge f(v) = \text{black}, \\ c[t', f] & \text{otherwise} \end{cases}$$

For forget-node t with child t' and forgotten vertex v we have:

$$c[t, f] = \min\{c[t', f_{v\to\text{black}}], c[t', f_{v\to\text{white}}]\}$$

For join-node t with child t_1 and t_2 we have:

$$c[t, f] = \min_{f_1, f_2}\{c[t_1, f_1] + c[t_2, f_2] - |f^{-1}(\text{black})|\}$$

where we consider all f_1 and f_2 with $(f(v), f_1(v), f_2(v)) \in \{$(black, black, black), (white, white, white), (white, white, gray), (white, gray, white), (gray, gray, gray)$\}$ for all $v \in B(t)$.

When adapting the algorithm of Cygan et al. [Cyg+15] we made some changes to the colouring and thus had to make some changes in our DP. We first explain the differences and the later on argue for each node type that our changes in the DP are enough to counteract our changed Definition.

Differences in the definitions of the colouring. The original source allows grey vertices to be dominated by black vertices, while we on the other hand demand grey vertices to be not dominated. Resulting from this, a vertex could be coloured grey in the original algorithm even though it is dominated and thus could be white.

Individual solution quality. This does not increase the quality of the individual solution further down the road, since we now need to dominate the grey vertex later on, while our algorithm has already dominated it.

Thus, we need to ensure that the recursive formula are still able to access all necessary partial solutions and can explicitly consider white vertices where the old rules only consider grey vertices, should it be needed. We argue that this is fulfilled for all node types.

Introduce-nodes. For introduce-nodes it is easy to see, that the change of definition has no effect, since the new vertex neither dominates nor is dominated.

Introduce-edge-nodes. For introduce-edge-nodes we are only interested in those cases, where the new edge could change the dominance. Thus, one of the vertices u and v has to be coloured black and the other one white or grey (in traditional definition) by f. Thus, lets assume w.l.o.g. u to be black.
When we consider the case where v is white, then v can either be already dominated in the child node (in this case we do not need to do anything) or it gets newly dominated by the introduced edge. Thus, we can have $f(v)$ = white or $f(v)$ = gray following the new definition in the child node and we surely gain the correct value by taking the minimum. The old definition catches the uncertainty by allowing gray vertices to be dominated. If v is gray in the traditional definition, we can, if we need this particular partial solution at a later stage, use the partial solution with v being white instead and gain a better solution as mentioned above.

Forget-nodes. For forget-nodes we need the forgotten vertex to be either black or white already, so the reader can easily convince himself that no change in the rule is necessary.

Join-nodes For join-nodes we need to ensure, that we consider all cases of f_1 and f_2 in correct fashion.

In the case of (black, black, black) there is no change and in the case of (gray, gray, gray) one option is that only one of the vertices in the children is white and thus we land in (white, white, gray) or (white, gray, white). If both vertices are white or we have the case (white, white, gray) or (white, gray, white) we need to additionally consider (white, white, white) as a possibility. This is done by the new definition.

These results can be summarized in the following lemma.

Lemma 7.9: *The restriction of not allowing grey vertices to be dominated, does not change the quality of the solution to Dominating Set, if we adapt the introduce-edge- and join-rules to the ones listed above.*

We now need to analyse this DP and its running time. For this, we first consider the number of partial solutions in each bag. Here, it is necessary to separately prove a bound for join-nodes since we search for the minimum over all colourings with the necessary attributes in our DP.

Lemma 7.10: *Let $G = (V, E)$ be a graph with a given extended cp-tree-decomposition (T, B, P) of weight* cptw. *Let t be a node in T that has another type than join-node. Then, we need to consider at most $2^{\Delta(G,T,P)\cdot \mathrm{cptw}}$ partial solutions per bag.*

Proof. We can use the properties of the cp-treewidth and show that we need to only use some of the partial solutions.

For a dominating set we can choose a maximum of $\Delta(G, T, P)$ vertices from each clique (meaning colouring them black) in node t, because we could choose instead of more than $\Delta(G, T, P)$ vertices from a clique, one vertex from the $\Delta(G, T, P)$ cliques in the neighbourhood and one vertex from the clique we are trying to colour. An illustration of this fact can be seen in Figure 7.2.

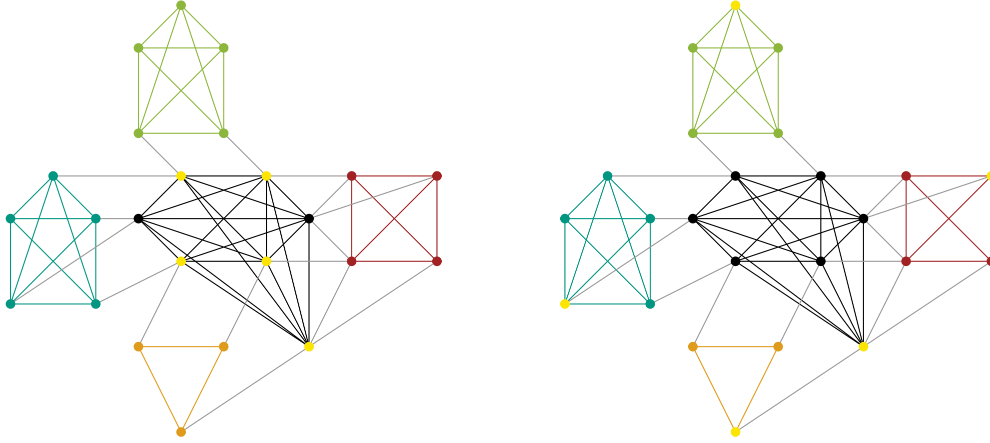


Figure 7.2.: An excerpt of a bag $B(t)$ in a cp-tree-decomposition (T, B, P) containing five different cliques (marked through their colouring) and on the left side a set of more than $\Delta(G, T, P)$ vertices in yellow. On the right side a second set of same size that dominates all vertices dominated by the first set and possibly more.

This means that we can find an optimal solution by only considering such solutions that use less than $\Delta(G, T, P)$ black vertices in each node t. Thus, we can bound the count of possible partial solutions of a node t, meaning the possible variants of choosing less than $\Delta(G, T, P)$ vertices from each clique, by the following calculation.

$$\begin{aligned}\prod_{C\in\mathcal{P}_t} (|C|+1)^{\Delta(G,T,P)} &= \prod_{C\in\mathcal{P}_t} 2^{\log(|C|+1)^{\Delta(G,T,P)}} \\ &= 2^{\sum_{C\in\mathcal{P}_t} \log(|C|+1)^{\Delta(G,T,P)}} \\ &= 2^{\Delta(G,T,P)\cdot \sum_{C\in\mathcal{P}_t} \log(|C|+1)} \\ &\leq 2^{\Delta(G,T,P)\cdot \mathrm{cptw}}\end{aligned}$$

For all node types, except join-nodes, we have limited the arrangements that have to be searched. ∎

Lemma 7.11: *Let $G = (V, E)$ be a graph with a given extended cp-tree-decomposition (T, B, P) of weight* cptw*. Let t be a join-node in T. Then, we need to consider at most $2^{\Delta(G,T,P)\cdot \mathrm{cptw}}$ partial solutions per bag and for this need to search $2^{3\cdot\Delta(G,T,P)\cdot \mathrm{cptw}}$ partial solutions.*

Proof. For join-nodes we can make the same estimations as in Lemma 7.10, but we need to additionally find a upper bound for how many f_1 and f_2 we need to look at. For a join-node t we have to, in the worst case, work through all partial solutions of both children for each partial solution of t to find the minimum. This gives us $2^{\Delta(G,T,P)\cdot \mathrm{cptw}} \cdot 2^{\Delta(G,T,P)\cdot \mathrm{cptw}} \cdot 2^{\Delta(G,T,P)\cdot \mathrm{cptw}} = 2^{3\cdot\Delta(G,T,P)\cdot \mathrm{cptw}}$ partial solutions we need to search. ∎

We can combine these results to obtain our final theorem.

Theorem 7.12: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*. Then,* Dominating Set *can be solved in $\mathcal{O}\left(2^{3\cdot\Delta(G,T,P)\cdot \mathrm{cptw}} \cdot \mathrm{tw} \cdot n^2 \cdot \log(n)\right)$.*

Proof. We first construct an extended cp-tree-decomposition using Theorem 5.8 and note that the weight increases by one. To prove this theorem we can use our colouring in our DP defined above. Using Lemma 7.10 and Lemma 7.11 we need to consider and search at most $2^{3\cdot\Delta(G,T,P)\cdot(\text{cptw}+1)}$ partial solutions. We, thus, can argue for our running time bound in similar fashion as in the traditional algorithm and we thus gain with $\mathcal{O}(\text{tw}\cdot n\cdot\log(n))$ bags and $\mathcal{O}(n)$ additional time the resulting total time. Finally, note that the weight increase of one also gets lost in the $\mathcal{O}$-Notation. ∎

Again, we want to apply this algorithm to similar problems by making some small adaptations. We begin with r-Dominating Set, where a vertex dominates all vertices that have at most distance r to it.

Corollary 7.13: *Let* $G=(V,E)$ *be a graph with a given cp-tree-decomposition* (T,B,P) *of weight* cptw *and width* tw*. Then, r-Dominating Set can be solved in* $\mathcal{O}\left(2^{3\cdot\Delta(G,T,P)\cdot\text{cptw}}\cdot\text{tw}^3\cdot n^2\cdot\log(n)\right)$.

Proof. We again follow de Berg et al. [Ber+18] in adapting Dominating Set to r-Dominating Set. The upper bound of $\Delta(G,T,P)$ vertices per clique stays intact, since the moving of a vertex to a neighbouring clique in the case of more than $\Delta(G,T,P)$ vertices only helps cover more vertices. Thus, the dynamic program has to additionally only remember the distance of each vertex to a vertex of the solution. We thus have to look up to distance r from the black coloured vertex when colouring white vertices. This can be done by a BFS with maximal searchdepth of r. During this process we need to look at less than tw vertices and tw^2 edges. This gives an additional factor. ∎

As before we want to consider a variant of Dominating Set that requires all solutions to be connected.

Corollary 7.14: *Let* $G=(V,E)$ *be a graph with a given cp-tree-decomposition* (T,B,P) *of weight* cptw*. Then, Connected Dominating Set can be solved in* $\mathcal{O}\left(2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}\cdot n^2\cdot\log(n)\right)$.

Proof. For this, we can use the same algorithm as for the not connected variant but we need to also track if the components of the dominating set are connected. This can be done by using equivalence relations and the rank-based-approach, which we employ later on for our algorithm solving Steiner Tree, to remember which vertices are connected and requiring the final solution to be connected [Ber+18].

In our algorithm we need to restrict functions, which can be achieved with *proj*, add weights, which can be done with *shft*, take minima, which can be done with *union*, and recolour vertices, which can be done by switching from the representation of one partial solution to one of another.

Thus, we have expressed our algorithm with the operations of Brodlaender et al. [BCKN15] and we can again use the rank-based-approach to limit the number of equivalence relations to $2^{\mathcal{O}(\Delta(G,T,P)\cdot p(B,P))}$ since we select at most $\Delta(G,T,P)$ vertices from each clique and have at most $p(B,P)$ cliques per bag and thus our solutions are of size $\Delta(G,T,P)\cdot p(B,P)$ at most. This gives us at most $2^{\mathcal{O}(\Delta(G,T,P)\cdot p(B,P))}\cdot 2^{3\cdot\Delta(G,T,P)\cdot\text{cptw}}=2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}$, with $p(B,P)\leq\text{cptw}$ (Lemma 3.6), partial solutions. Using Theorem 2.3 with partial solutions of size of at most $\Delta(G,T,P)\cdot p(B,P)\leq\Delta(G,T,P)\cdot\text{cptw}$ we can show that using the rank-based-approach does not increase the needed time. We can use Lemma 3.3 and Lemma 5.6 to bound then number of bags by $2^{\text{cptw}}\cdot n$ and then lose the factor of 2^{cptw} in the $\mathcal{O}$-Notation. ∎

7.3. Steiner Tree

In this section, we develop an algorithm for STEINER TREE. For this, we will first quote a useful lemma and prove a upper bound on the number of vertices of each clique we can pick into the partial solution and then give our algorithm.

Lemma 7.15: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) that contains the clique C in one of its bags and X a minimal solution to STEINER TREE. We call the set of terminal vertices K. Then, for each vertex $v \in (C \cap X) - K$ a private neighbour $u \in X - C$, such that u and v are adjacent and u is not adjacent to any previously assigned private neighbour, exists.*

Proof. This is shown in the proof of Lemma 18 from de Berg et al. [Ber+18]. ∎

As before we want to use the structure of our bags to reduce the number of partial solutions. For this, we show how many vertices of each clique can be in a partial solution.

Lemma 7.16: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) that contains the clique C in one of its bags and X a minimal solution to STEINER TREE. Then, X uses at most $\Delta(G, T, P)$ vertices from C, that are not also in the set of terminal vertices K.*

Proof. We can use Lemma 7.15 to find a private neighbour for each vertex in X. We know that C is neighboured by at most $\Delta(G, T, P)$ other cliques (remember the fact that $\Delta(G, T, P)$ is maximal degree of a clique). This leads to the number of private neighbours and thus the number of vertices to be chosen in a clique, to which a one-on-one relationship exists, being bounded by $\Delta(G, T, P)$. We cannot have two connections of the Steiner tree in $G(X)$ between two cliques, since we could save one connection by using an edge inside of one of the cliques and thus could gain a solution that uses the same amount or less vertices. ∎

This lemma allows us to describe the algorithm.

Theorem 7.17: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*, width* tw *and clique-degree $\Delta(G, T, P)$. Then, STEINER TREE can be solved in $2^{\mathcal{O}(\Delta(G,T,P)\cdot \mathrm{cptw})} \cdot n^2 \cdot \log(n)$.*

Proof. In this case we do not give a complete algorithm and instead follow de Berg et al. [Ber+18] and show how we can use the rank-based-approach of Bodlaender et al. [BCKN15] for our parameter[3]. In their work on the rank-based-approach they show its application to STEINER TREE, this is the algorithm we adapt. This includes first talking about representation of partial solutions, then finding a ceiling for the number of those and finally using the rank-based-approach to reduce this limit.

The traditional algorithm uses a vertex set, as well as a equivalence relation representing the parts of the tree that are not yet connected, as a partial solution. The rank-based-approach manages to represent all necessary calculations for the recursion with the use of a set of operations and thus can use a function called *reduce* to decrease the size of a solution.

[3] We refer to the book of Cygan et al. [Cyg+15] for a traditional algorithm not using the rank-based approach. This algorithm does run in time $2^{\mathrm{tw}\cdot\log(\mathrm{tw})} \cdot n^{\mathcal{O}(1)}$ but gives some insight into the functioning of dynamic programming algorithms for STEINER TREE.

Like de Berg et al. [Ber+18] we set the weights in the algorithm with the rank-based-approach to one and only use it for the unweighed case. We also represent a partial solution in node t as the subset $S \subseteq B(t)$ which are the vertices of the steiner tree in this bag and a equivalence relation on S that denotes which vertices share a common connected component. As per Lemma 7.16 we need to only choose $\Delta(G,T,P)$ or less vertices from each clique and thus we do not need to consider all partial solutions that select more vertices. This leads us to the following number of partial solutions to consider in node t:

$$\begin{aligned}\prod_{C\in\mathcal{P}_t}(|C|+1)^{\Delta(G,T,P)} &= \prod_{C\in\mathcal{P}_t} 2^{\log(|C|+1)^{\Delta(G,T,P)}}\\ &= 2^{\sum_{C\in\mathcal{P}_t}\log(|C|+1)^{\Delta(G,T,P)}}\\ &= 2^{\Delta(G,T,P)\cdot\sum_{C\in\mathcal{P}_t}\log(|C|+1)}\\ &\leq 2^{\Delta(G,T,P)\cdot \text{cptw}}\end{aligned}$$

For each subset S there are less than $2^{\Theta(|S|\cdot\log|S|)}$ equivalence relations and we can represent those by using the reduce algorithm with at most $2^{|S|}$ equivalence relations. Due to that we can, by using reduce after processing each bag, bound the number of relations to be considered by $2^{\mathcal{O}(|S|)}$. We know that we need to pick at most $\Delta(G,T,P)$ vertices per clique and that we have a maximum of $p(B,P)$ cliques in each bag. This allows us to state about the size of each subset: $|S| \leq p(B,P)\cdot\Delta(G,T,P)$. Due to the rank-based-approach, we need to consider no more than $2^{\mathcal{O}(|S|)} \leq 2^{\mathcal{O}(\Delta(G,T,P)\cdot p(B,P))} \leq 2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}$, using $p(B,P) \leq \text{cptw}$ (Lemma 3.6), representative equivalence classes. All together we get at most

$$2^{\Delta(G,T,P)\cdot\text{cptw}} \cdot 2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})} = 2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}$$

partial solutions by considering for each subset all equivalence relations. Again, we have $\mathcal{O}(2^{\text{cptw}}\cdot n\cdot\log(n))$ bags and can calculate each value in $\mathcal{O}(n)$ resulting in running time $2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}\cdot\mathcal{O}(n^2)$. Note that using the rank-based-approach does not increase our running time (Theorem 2.3). For this, we did use Lemma 3.3 to modify the bound on the number of nodes given by Lemma 5.6. Using Theorem 2.3 with partial solutions of size of at most $\Delta(G,T,P)\cdot\text{cptw}$ we can show that using the rank-based-approach does not increase the needed time. ∎

As done for Independent Set we want to consider the variant of Steiner Tree which uses weights for each vertex.

Corollary 7.18: *Let $G = (V,E)$ be a graph with a given cp-tree-decomposition (T,B,P) of weight* cptw, *width* tw *and clique-degree $\Delta(G,T,P)$, and let $g : V \longrightarrow \mathbb{R}_0$ be a weight function for G. Then, Weighted Steiner Tree can be solved in* $2^{\mathcal{O}(\Delta(G,T,P)\cdot\text{cptw})}\cdot n^2\cdot\log(n)$.

Proof. Since adding weights does not change how many vertices we need to pick and thus does not change the number of partial solutions, we can use the same algorithm as in the unweighted case and only need to consider weight in the underlying algorithm of Bodlaender et al. [BCKN15]. This algorithm can handle the weighted case. ∎

7.4. Clique

The Clique problem is quite an odd case for parametrized algorithms based on cp-treewidth, since we already use cliques in creating our cp-tree-decomposition. First, observe that the clique-partition can split the largest clique of a graph (See the example in the proof of Observation 11 in the work of Bläsius et al. [BKW23]). We thus need to check for cliques that are split between multiple cliques of the partition. We can use an algorithm for treewidth that is trivially given by checking for all subsets of each bag whether they are a clique and sufficiently large. This will find the largest clique, when done for all bags, since each clique in the original Graph G is contained as a complete clique in at least one bag (Lemma 2.4). We give an algorithm parametrized by cp-treewidth.

Theorem 7.19: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*, where the clique-partition is optimal for the given tree decomposition. Assume G has neighbourhood index $\delta(G, T, B, P)$. Then, Clique can be solved in $2^{\mathcal{O}(\delta(G,T,B,P)\cdot \mathrm{cptw})} \cdot n^{\mathcal{O}(1)}$.*

Proof. We first observe that only the vertices with edges that lead outside the clique can be contained in cliques that are split in the partition and thus we can prune the other vertices and are left with at most $\delta(G, T, B, P)$ in each clique. The resulting graph has treewidth of $\delta(G, T, B, P) \cdot \mathrm{cptw}$, since we have at most $p(B, P) \leq \mathrm{cptw}$ (Lemma 3.6) cliques. We can then run the algorithm for treewidth and get our final result. ∎

7.5. Lower bounds for the number of vertices picked in each clique

For the problems of this section we needed an additional factor in the exponent for our algorithms. This factor usually resulted from the bounds we proved on the number of vertices we need to pick from each clique. In this section, we show for some problems that this factor is necessary when the algorithm is designed using this principle. In Chapter 6 we showed that some kind of factor, that grows up to exponentially faster than cp-treewidth, is necessary for fast FPT-algorithms by cp-treewidth. In this section, we show that with the current concept of developing such algorithms we need this factor to be asymptotically as large as the clique-degree.

We begin by showing for Dominating Set, Steiner Tree and Hamilton Cycle that we need to consider partial solutions that use $\Theta(\Delta(G, T, P))$ vertices from each clique. This obviously can be translated to the related problems for whom we derived the algorithms by adapting the algorithms for one of the above problems. By proving these results for Hamilton Cycle we get another requirement for any algorithm we can design for this problem.

Dominating Set For this, consider the graph in Figure 7.3 , which consists of one central clique of size c and $2 \cdot c$ vertices. Here, each vertex of the central clique is connected to two of the outer vertices that are no yet connected to other central vertices.

In this graph we need to choose all vertices of the central clique in order to obtain a minimal dominating set. This is since, in each module of a central vertex and its two adjacent outer vertices we need to choose either the central or both outer vertices. We thus choose $\frac{1}{2} \cdot \Delta(G, T, P) \in \Theta(\Delta(G, T, P))$ vertices from the central clique in the solution for the whole graph and thus need to choose the same amount of vertices from this clique in a partial solution for any bag that contains the central clique. This obviously are more than constantly

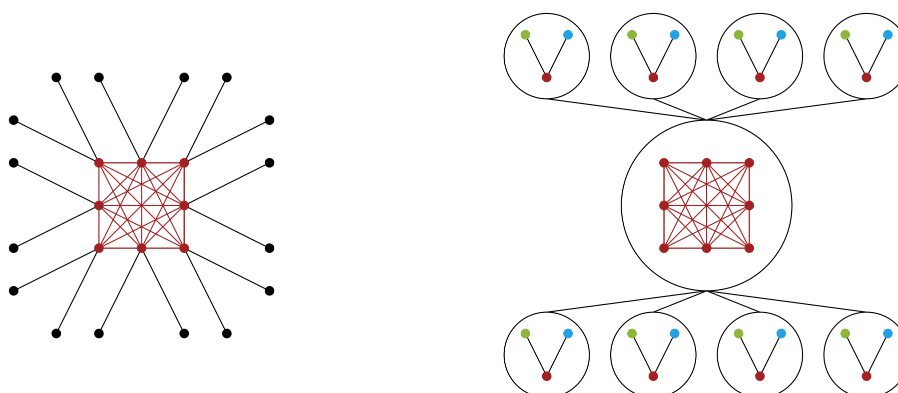


Figure 7.3.: A graph G with a cp-tree-decomposition (T, B, P) where $\Theta(\Delta(G, T, P))$ have to be picked from one clique into a dominating set. This is an example for $c = 8$. On the left the graph is shown and the central clique is marked in red. On the right a cp-tree-decomposition with minimal $p(B, P)$ is depicted. Here, the colours denote the clique-partition.

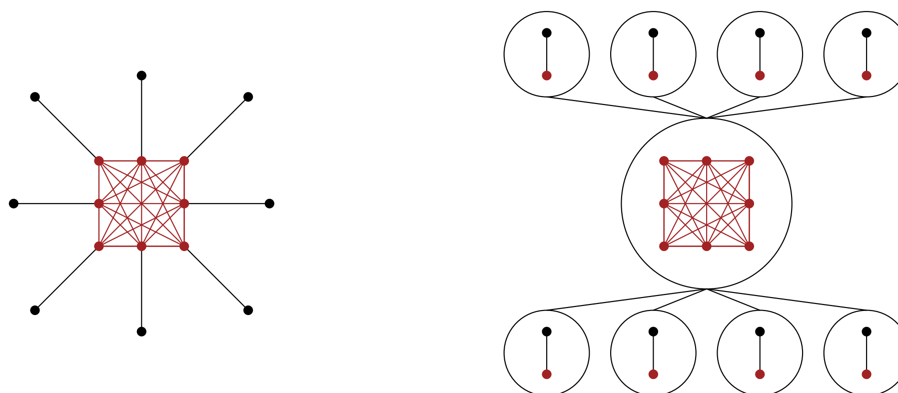


Figure 7.4.: A graph G with a cp-tree-decomposition (T, B, P) where $\Theta(\Delta(G, T, P))$ have to be picked from one clique into a steiner tree. This is an example for $c = 8$. On the left the graph is shown and the central clique is marked in red. On the right a cp-tree-decomposition with minimal $p(B, P)$ is depicted. Here, the colours denote the clique-partition. For the terminal vertices we choose all outer vertices.

many vertices, since $\Delta(G, T, P)$ grows proportional to c. We also need more than $p(B, P)$ vertices. For this, consider the cp-tree decomposition that assigns each module its own bag and connects all these bags to a central bag that contains all vertices of the central clique (see Figure 7.3 for an example). Here, $p(B, P) = 3$ for all c.

Steiner Tree Here, we consider a similar graph. The only difference is that each vertex of the central clique is connected to only one outer vertex. We then pick all outer vertices to be terminal vertices of the Steiner Tree problem. An example graph can be seen in Figure 7.4. It is obvious that we need to choose all vertices of the central clique and thus can get analogue results to above.

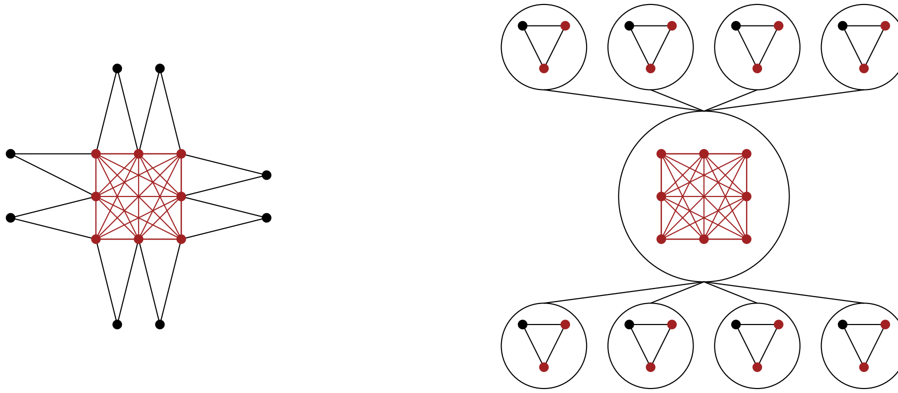


Figure 7.5.: A graph G with a cp-tree-decomposition (T, B, P) where each hamilton cycle contains $\Theta(\Delta(G, T, P))$ edges that lead from a vertex of the central clique to an outer vertex. This is an example for $c = 8$. On the left the graph is shown and the central clique is marked in red. On the right a cp-tree-decomposition with minimal $p(B, P)$ is depicted. Here, the colours denote the clique-partition.

Hamilton Cycle Again, we consider a similar graph and get similar results. Here, each outer vertex is connected to two central ones, that are connected to no other outer vertex. An example graph can be seen in Figure 7.5.

Implications Since we required the algorithms to work for all graphs and all cp-tree-decompositions, any algorithm for Dominating Set and Steiner Tree needs to work on the above graphs and cp-tree-decompositions. Thus, any DP that picks as partial solution for each bag the vertices of the bag that end up in the complete solution needs to consider partial solutions that pick up to $\Theta(\Delta(G, T, P))$ vertices from each clique. Due to the fact that all currently known algorithms for cp-treewidth are of this type and the only other known technique is brute forcing the problem for each bag, this implies that in order to find a substantially faster algorithm than we did, one needs to make major discoveries and invent a new technique.

For Hamilton Cycle the neuralgic points are the edges that connect two cliques. Of those we are mostly interested in edges leading to different cliques. Due to the above graph, that has $\Theta(\Delta(G, T, P))$ of such edges, our algorithm can only be improved using a DP, if a major discovery is made that allows us to discard some of those edges.

These results do not prove for any of those problems that specific factor of $\Delta(G, T, P)$ is necessary, but strongly hint that this is the case and show that we would need a different kind of algorithm to get a faster running time.

8. Building algorithms by tracing cliques

So far, we have seen many results that prove certain requirements that any algorithm solving Hamilton Cycle needs to fulfil. We have shown that it needs an additional parameter, since cp-treewidth is weaker by $f(x) = 2^x$ than treewidth for this problem, and we did also prove that this parameter needs to be asymptotically as large as the clique-degree, if we want to use approaches similar to the ones used so far. While it is possible to prove some attributes of hamilton cycles in cliques, this requires us to keep track of how the cliques change from one bag to another. Currently, the clique-partition of two adjacent bags can have major differences. To handle this, we introduce a third normalization step, called smooth cp-tree-decomposition, that places restrictions on the clique-partitions. The general statements for the use of smooth cp-tree-decompositions in our algorithms are the same as for nice and extended cp-tree-decompositions. With the introduction of smooth cp-tree-decompositions two questions arise. First, is this concept also applicable to other problems than Hamilton Cycle and second, is this concept powerful enough to describe all cp-treewidth-based algorithms or do we need to introduce more steps. To answer the first question, we give, in addition to Hamilton Cycle, algorithms for a number of problems. For the second question, we introduce a framework that summarizes our approaches and give conditions under which this framework can be applied. This framework can be used to describe algorithms for all problems, where the idea of using a dynamic program on cp-tree-decompositions and limiting the number of vertices chosen from each clique can be applied. Thus, to find an algorithm whose basic idea cannot be expressed by this framework, one needs to use another approach than finding a dynamic program on the cp-tree-decomposition and limiting the number of partial solutions based on the number of vertices chosen from each clique. It still might be possible to find other or faster algorithms. Furthermore, algorithms might have lower constant factors and lower constants in the exponent than the framework. Finally, we use this framework to debate the relation between cp-treewidth and BBKMZ-treewidth algorithms.

8.1. Smooth cp-tree-decompositions

The previous two normalizations make no demands on how the clique-partition of each bag compares to the partition of the previous bag. If we need to reference cliques of previous bags in our algorithm it might be helpful to ensure some basic properties for our clique-partitions. For this, we introduce the smooth cp-tree-decomposition.

As done for the other two normalizations, we then need to prove that a smooth cp-tree-decomposition exists and that cp-treewidth and number of bags stay low when constructing the smooth cp-tree-decomposition. The outline is similar to the last two subsections, we first give a construction, then show an upper bound on cp-treewidth, an upper bound on the node number and finally an upper bound on the running time of the construction. We summarize these results in Theorem 8.5. For our proofs we use our results for nice cp-tree-decompositions from Section 5.1, since the smooth cp-tree-decomposition is based on the nice cp-tree-decomposition.

Definition. For some problems it is necessary to trace cliques from one bag to another. For this, we need to ensure that the clique-partitions of the two bags are similar. This leads us to a new type of normalization, the smooth cp-tree-decomposition.

We first introduce a new node type called *partition-change-node*. A partition-change-node is a node t, having exactly one child t' with $B(t) = B(t')$, which is annotated with a vertex $v \in B(t)$. We say that it changes to which clique v is assigned. Here, the changed vertex v is removed from its clique in the clique-partition of t and inserted into another (possibly new) clique. Formally, this is $\mathcal{P}_t = move_{C' \to C}(v, \mathcal{P}_{t'})$, where v is in clique C in node t and in C' in node t'.

We define a *smooth cp-tree-decomposition* on the basis of the nice cp-tree-decomposition. In a smooth cp-tree-decomposition all nodes have one of the node types introduce-, forget-, join-, leaf- or partition-change-node. Furthermore, in a smooth cp-tree-decomposition additionally require that introduce-, forget- and join-nodes do not change the partitions. In particular, we require for introduce-nodes that the partition of the node and its child are the same apart from the introduced vertex which is added as its own clique. Similarly, we require for forget-nodes that the partition of the node and its child are the same apart from the forgotten vertex which is removed from its clique. Again, we require for join-nodes that the partition of the node and its two children are the same. Formally this can be stated as, using node t and child t' or children t_1 and t_2: $\mathcal{P}_t = \mathcal{P}_{t'} + v$ for introduce-nodes, $\mathcal{P}_t = \mathcal{P}_{t'} - v$ for forget-nodes and $\mathcal{P}_t = \mathcal{P}_{t_1} = \mathcal{P}_{t_2}$ for join-nodes. For leaf-nodes we have no added requirements.

Before we can give the construction of a smooth cp-tree-decomposition, it is useful to lay some groundwork. For this, we begin by explaining how we use partition-change-nodes to transition from one partition to another and then prove that this process keeps the weight low. This requires some additional thought, since no direct relation between the cliques of the partition exists. We thus need to establish which clique maps to which and then can use this to move the vertices from one to another.

Construction of a series transitioning between two partitions. Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P). Let t and t' be nodes in T with $B(t) = B(t')$ and let t' be a child of t. We give the construction of a series of partition change nodes, that transitions from $\mathcal{P}_{t'}$ to $\mathcal{P}_t$ and moves each vertex at most once. We begin with the partition of t' and then give a series of move-operations that transforms it into the partition of t. Since each move-operation corresponds to a partition-change-node, this fulfils our requirements. In this process we consider the cliques of $\mathcal{P}_t$ one after another and construct each clique from the cliques of $\mathcal{P}_{t'}$.

In more detail this means that we pick in each step a clique $C \in \mathcal{P}_t$, which we have not considered yet and construct this clique from an empty clique by adding the vertices that are assigned to it in $\mathcal{P}_t$. Next, we need to assign all needed vertices to C by moving them from their cliques in $\mathcal{P}_{t'}$ (called $C_1, \ldots, C_k$ for $k \in \mathbb{N}_+$) to C. Here, the vertices may be moved in any order and after the last vertex is moved we have constructed D. If a clique is empty after this step, we can remove it from the partition. In this transformation each vertex is moved only once and will not be moved by later steps. We can repeat this process until all cliques have been considered and get the required series.

This idea finds its application in our construction of smooth cp-tree-decompositions. But first, we prove that the series does keep the width within a constant factor.

Lemma 8.1: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *Let t and t' be nodes in T with $B(t) = B(t')$ and let t' be a child of t. Then, in the series transitioning from $\mathcal{P}_{t'}$ to $\mathcal{P}_t$ as constructed above each bag of the series has weight of at most* $2 \cdot \text{cptw}$.

Proof. We bound the treewidth by comparing each clique of a partition-change-node in the series to a clique of $\mathcal{P}_{t'}$ or $\mathcal{P}_t$ and proving that it is a subset of such a clique. We use the same naming as in the above construction. For this, observe that in the construction only two types of cliques can appear: First those that formed $\mathcal{P}_{t'}$ (and may have lost some vertices) and second those that will form $\mathcal{P}_t$ (and may still need to get some vertices).

This can be seen in the following way. We start with the cliques of $\mathcal{P}_{t'}$ and then in each step remove vertices from a set of cliques $C_1, \ldots, C_k$ for $k \in \mathbb{N}_+$ and then add them to a new clique, which we call C^* to allow us to distinguish between the state of C when the series is finished and during the series. Here, it is important that vertices are only added to new cliques and not to any cliques already in $\mathcal{P}_{t'}$. If we remove some vertices from $C_1, \ldots, C_k$ the resulting cliques are still subsets of $C_1, \ldots, C_k$ and thus of cliques from $\mathcal{P}_{t'}$. Furthermore, we only add vertices of a clique C from $\mathcal{P}_t$ to C^* and thus C^* is a subset of such a clique.

We can use this to estimate the weight of the nodes during the series. The cliques of the two types are subsets of cliques of $\mathcal{P}_{t'}$ and $\mathcal{P}_t$ respectively and thus their total weight is bounded by the weight t and t' respectively and this means that it is bounded by cptw. For this, it is important that we have only one subset for each clique of $\mathcal{P}_{t'}$ and $\mathcal{P}_t$. This is fulfilled by our construction. Since the total weight of each of the two types is bounded by cptw, the weight of each bag is bounded by $2 \cdot \text{cptw}$.

∎

We can now use these discoveries in our construction.

Construction. Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight cptw and width tw. We now explain how to compute a smooth cp-tree-decomposition of G.

When given a cp-tree-decomposition of weight cptw and width tw, we, as the first step, compute a nice cp-tree-decomposition of weight cptw, width tw and $\mathcal{O}(\text{tw} \cdot n)$ nodes in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$ using Theorem 5.4.

We now need to adapt this nice cp-tree-decomposition to a smooth cp-tree-decomposition. For this, we insert partition-change-nodes and adapt the partitions of each node such that we fulfil the added requirements.

For introduce-/forget-node t with child t' we first add the introduced/remove the forgotten vertex and then adapt the clique-partition from t' to t. In more detail this means adding an introduce-/forget-node after t', called t^*, that fulfils $\mathcal{P}_{t^*} = \mathcal{P}_{t'} \pm v$. We then can use the series of partition-change-nodes constructed above to transition from the partition of t^* to t. Keep in mind that t, or at least the node with the same vertices and partition as t, is the last node in this series and thus no longer an introduce-/forget-node.

For join-node t with children t_1 and t_2 we use the same partition as before. We call the children of t in the smooth cp-tree-decomposition t'_1 and t'_2. Here, we use a series of partition-change-nodes for each child to fulfil $\mathcal{P}_t = \mathcal{P}_{t'_1} = \mathcal{P}_{t'_2}$.

For leaf-nodes there were no new requirements and we do not need to make any adaptations.

By combining these ideas and traversing the nice cp-tree-decomposition from leaf to root we can construct a smooth cp-tree-decomposition.

After giving the construction we now want to show that the weight stays relatively small, since the running time of our algorithms depends on it being low.

Lemma 8.2: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw. *Then, the smooth cp-tree-decomposition computed constructed above has weight of at most* $2 \cdot \text{cptw} + 2$ *and width* tw.

Proof. Since we only add series of partition-change-nodes in between two nodes, we only need to show that these series have small enough weight. For this, we can bound weight of the first and last node by $\text{cptw} + 1$, since they either reuse subsets of partitions from the nice cp-tree-decomposition or use a partition that includes the introduced vertex as its own clique. We now can use Lemma 8.1 to bound the weight during the series by $2 \cdot (\text{cptw} + 1) = 2 \cdot \text{cptw} + 2$.

Since we only change the partitions of the already existing nodes and only add partition-change-nodes, which do not add vertices, the width stays the same. ∎

The next step is to show a bound for the number of bags created in our construction.

Lemma 8.3: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of width* tw. *Then, the smooth cp-tree-decomposition constructed above has $\mathcal{O}(\text{tw}^2 \cdot n)$ nodes.*

Proof. For this, we can use, that the nice cp-tree-decomposition we construct, has at most $\mathcal{O}(\text{tw} \cdot n)$ nodes and by replacing each original node with at most tw nodes, we cannot get more than $\mathcal{O}(\text{tw}^2 \cdot n)$ nodes. Since each vertex can change cliques at most once during one series of partition-change-nodes, we do not need to add more than tw nodes to transition from one partition to another. ∎

The final attribute we want to show concerns the running time of our construction.

Lemma 8.4: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw *and at most $\mathcal{O}(\text{tw} \cdot n)$ nodes. Then, the construction above runs in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$.*

Proof. The construction of the nice cp-tree-decomposition can be completed in $\mathcal{O}(\text{tw}^2 \cdot (n+m))$ (Lemma 5.3). Since we do not need to add additional nodes for leaf-nodes, we do not need any additional time for those. For introduce- and forget-nodes the adding of the new introduce- or forget-node can be done in $\mathcal{O}(\text{tw})$, since we only need to copy the existing node and modify one vertex.

To find the series of partition-change-nodes we need to, for each clique of the new partition, compare it to the cliques of the old partition and assign the vertices from their old cliques to the new one. We achieve this by storing for the old partition in an array of size $\text{tw} + 1$ for each vertex to which clique it belongs. Here, we assign each clique and vertex a number and at the index of the vertex store the number of the clique. We can now iterate over all vertices of the clique we want to construct and for each vertex request the clique to which it belongs in the previous partition. By assigning the vertices to the new clique and giving it a new number, we can keep the structure up to date. Thus, the whole process of changing partitions takes time $\mathcal{O}(\text{tw})$ since each of the $\text{tw} + 1$ vertices is moved at most once and request are fulfilled in time $\mathcal{O}(1)$. After each moved vertex we add a new node to the series. For join-nodes we can make similar estimations since the main task is finding the series of partition-change-nodes. Since we have at most $\mathcal{O}(\text{tw} \cdot n)$ nodes in a nice cp-tree-decomposition (Lemma 5.2) we get the total running time. ∎

Theorem 8.5: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, a smooth cp-tree-decomposition of G with weight $2 \cdot \text{cptw} + 2$, width* tw *and $\mathcal{O}(\text{tw}^2 \cdot n)$ nodes can be computed in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$.*

Extended smooth cp-tree-decomposition So far we have used nice cp-tree decompositions as basis for our definition of smooth cp-tree-decompositions. Since some traditional algorithms use extended tree decompositions it is important to also introduce a version for extended cp-tree-decompositions. An *extended smooth cp-tree-decomposition* is a smooth cp-tree-decomposition that additionally allows nodes to be introduce-edge-nodes. Furthermore, we require the partition of an introduce-edge-node and its child to be the same.

By using the construction of Section 5.2, which adds introduce-edge-nodes after each introduce-node, on the smooth cp-tree-decomposition the following result is a direct consequence of the results of Section 5.2 and Section 8.1.

Corollary 8.6: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition of weight* cptw *and width* tw*. Then, an extended smooth cp-tree-decomposition of G with weight $2 \cdot \text{cptw} + 3$, width* tw *and $\mathcal{O}\left(\text{tw}^2 \cdot n \cdot \log(n)\right)$ nodes can be computed in time $\mathcal{O}\left(\text{tw}^2 \cdot (n + m)\right)$.*

Proof. Note that the partition stays the same due to the fact that we added the introduced vertex to its own clique in the introduce-node and the construction keeps this the same[1]. Next, the treewidth is increased by one due to Lemma 5.5 and at most $\log(n)$ introduce-edge-nodes are added since we have the same number of leaf-root-paths as before smoothing the cp-tree-decomposition and thus Lemma 5.6 can be used. Finally, the running time is dominated by the calculation of the smooth cp-tree-decomposition. For this, consider Lemma 5.7 with $\mathcal{O}(\text{tw}^2 \cdot n)$ nodes in the beginning. ∎

Finally, we can analyse how the clique-degree behaves for this new normalization and obtain the following statement as corollary of Lemma 7.7.

Corollary 8.7: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of clique-degree $\Delta(G, T, P)$. Then, the smooth/extended smooth cp-tree-decomposition (T', B', P') of Theorem 8.5/ Corollary 8.6 has clique-degree $\Delta(G, T', P') \leq \Delta(G, T, P)$.*

Similarly, we can extract this corollary from Lemma 7.8.

Lemma 8.8: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of neighbourhood index $\delta(G, T, B, P)$. Then, the smooth/extended smooth cp-tree-decomposition (T', B', P') of Theorem 8.5/ Corollary 8.6 has neighbourhood index $\delta(G, T', B', P') \leq \delta(G, T, B, P)$.*

8.2. Hamilton Cycle and Hamilton Path

For solving Hamilton Cycle the approach we used so far does not work. We cannot simply use the traditional algorithm and only pick a limited number of vertices from each clique. For this problem, by definition, we need all vertices of each clique. Due to this, we use a workaround to represent each clique by fewer vertices and then complete the cycle by adding the left out vertices retrospectively. For this to work, we need to ensure that the clique-partition from one node to the next changes only marginally, which we guarantee by using a smooth cp-tree-decomposition.

[1]The option of moving the introduced vertex to an other clique after some edges were added cannot be used. This is no problem since this effects none of the bounds we did prove.

For the algorithm solving Hamilton Cycle we use a two step approach. We first identify possible representatives for each clique and then for each choice of representatives we use a variant of a traditional algorithm to generate new partial solutions for each bag out of the already existing partial solutions. By keeping track of the representatives and the vertices in the original smooth cp-tree-decomposition in parallel we can solve Hamilton Cycle.

For traditional algorithms we base our algorithm on the comparison of Ziobro et al. [ZP19] from which we use the variant they dubbed the naive approach. We also use the lecture slides of Bläsius [Blä21a] on which we base the description of our dynamic program.

Our description of the algorithm is outlined the following way: First we describe how we choose our partial solutions and representatives, then we explain how the transition to the representatives is achieved. The next step is explaining the dynamic program by giving the instructions on how to create new partial solutions. Finally, we can bound the number of partial solutions and thus prove the running time and obtain our final theorem. We begin by explaining the concept of representatives.

Partial solutions. For a node t a partial solution consists of a subset of the vertices $X \subseteq B(t)$ called representatives and for each representative the information how many edges have been chosen so far for this representative, as well as the information how these representatives are connected. The second part of such a partial solution is exactly the information stored for each partial solution in the traditional algorithm (see for example Bläsius [Blä21a]) and corresponds to a partial solution on the graph using only the representatives.

Representatives. We start with one basic observation: A hamilton cycle using at most two edges between each pair of cliques exists if and only if a hamilton cycle using arbitrarily many edges between each pair of cliques exists. This observation was proven by Ito et al. [IK10].

We can then use this observation to reduce the number of vertices in each clique and thus, we can transform each bag so it uses less vertices. For each clique we thus need to pick at most two vertices for each neighbouring clique that serve as entrance into the clique from its neighbour. All partial solutions that pick more vertices are discarded instantly. Here, we note for each picked vertex in which clique it is. By considering all possible picks we guarantee that our algorithm finds any possible hamilton cycles.

When running the second phase, the algorithm needs to ensure that we only make two connections between each two cliques and the edges inside the clique can be correctly assembled. This means guaranteeing all non-representatives can be added to the partial solution we did calculate for the representatives. For this, at least one edge between two vertices of the same clique is needed, which then can be transformed into a path containing all of the non-representatives (see Lemma 8.9 for the formal proof). We generate our partial solutions without considering this, but when we found a valid solution we check if it fulfils this property. Thus, we can, for each vertex that is not yet added to the hamilton cycle, check to which clique it belongs and whether there is an edge in the cycle between two vertices in the clique. If this is true we can add the vertex, if it is false for every bag we can discard the solution as a false positive. This allows us to extend each partial solution consisting of representatives to a partial solution of all vertices.

Using the representatives in our algorithm. The main idea of the algorithm is now to run a dynamic program on the smooth cp-tree-decomposition and for each bag find all possible sets of representatives. The dynamic program then uses the partial solutions of the last bag to

create new partial solutions similarly to the traditional idea of dynamic programming on tree decompositions. Here, only the representatives are used in these partial solutions since we can simply replace the edge between two vertices of the same clique with a path containing all non-representatives. We thus need to guarantee that the change of representatives corresponds to the change in vertices in the original decomposition. Due to the smooth cp-tree-decomposition this is only important in partition-change-nodes and thus those are covered in our dynamic program. When we reach a partial solution that is a valid final solution, the algorithm has found an hamilton cycle and if not there is no hamilton cycle in the original graph.

In the following lemma we prove the correctness of this approach.

Lemma 8.9: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*. G has a hamilton cycle if and only if we can choose at most $2 \cdot \Delta(G, T, P)$ vertices from each clique as representatives and give a hamilton cycle in G, where all non-representatives are lined up on one path that contains only non-representatives between two representative of the same clique.*

Proof. We prove the non-trivial implication. Given a hamilton cycle of G, we can use the technique of Ito et al. [IK10] to find a hamilton cycle that uses at most two edges between each pair of cliques. We then choose the vertices adjacent to those edges as representatives. These are at most $2 \cdot \Delta(G, T, P)$ per clique since each clique has at most $\Delta(G, T, P)$ neighbouring cliques. We then reorder the non-representatives of each clique C to be on a single path. If the clique contains only representatives, nothing needs to be done. In the other case, at least one path in the hamilton cycle connects two representatives of C and uses only non-representatives of C, due to the fact that each non-representative is only connected to vertices of the same clique by the hamilton cycle. We can then replace this path with a path that begins and ends at the same representatives and that lines up all non-representatives of C. Since we consider a clique, we can always find the needed edges for this. If we do this for each clique, we get the required cycle. ∎

This lemma proves our process of reassembling correct. Since the path beginning and ending at a representative and only containing non-representative is an edge in the partial solution on the representatives, our criterion is equivalent to the existence of any hamilton cycle. The algorithm considers all choices of representatives and thus finds the correct one.

Dynamic program. We now describe how our dynamic program creates new partial solutions based on the representatives for a bag of the smooth cp-tree-decomposition.

This description is similar to the traditional case but contains also partition-change-nodes and make some changes to deal with representatives. We use the same partial solutions as in the traditional case. This means, we remember for each vertex whether it has degree zero, one or two and also remember which vertices are connected. For node t, we use X_t^0 for the nodes for whom we have not picked any edges yet, X_t^1 for those with one edge and X_t^2 for those with both edges chosen already. Remember from above that we also note for each vertex in which clique it is.

We now look at each node type of the smooth cp-tree-decomposition and explain which new partial solutions we can generate from each old partial solution. If and only if we can generate a partial solution for the root, that represents a hamilton cycle, our original graph also has a hamilton cycle.

Leaf-node. For a leaf-node t we can use the empty solution, which is our base case.

Forget-node. For a forget-node t with child t' and forgotten vertex v, where v is a representative, we can remove v from each partial solution of t' and thus gain a partial solution for t, if $v \in X_{t'}^2$. This is, since it already has both edges and thus is no longer needed. If $v \notin X_{t'}^2$ the partial solution cannot lead to a valid hamilton cycle since v can not be in the cycle, but is already forgotten. This means that we can safely discard this partial solution. Note that the clique-partition of both bags are the same except for v and thus the representatives only change insofar that v is removed. If v is a non-representative we can reuse the partial solutions of t' since there was no change in the representatives.

Introduce-node. For an introduce-node t with child t' and introduced vertex v we can generate new partial solutions by considering possible connections to already existing vertices. Here, v is in its own clique and thus chosen as representative. We do not need to consider any edges from v to vertices in X_t^2, since they already have two edges. Thus, we can pick subsets of the remaining edges that have size of at most two. We then adapt our partial solution accordingly: If we added no edges we insert v into X_t^0, if we added one edge v is put into X_t^1 and two edges add v into X_t^2. We similarly need to move the nodes on the other end of the edges up one degree and adapt the connections we remember. Again, the clique-partitions, except for v, are the same and thus, apart from adding v, we need to make no changes.

Join-node. For a join-node t with children t_1 and t_2 we first note that we can chose partial solutions where the representatives are exactly the same for all three nodes since they share a common clique-partition. We thus can combine the partial solutions of the children to gain a partial solution of t. For this, we can add for each representative its degree and thus group it into a new category. We also need to calculate which vertices from X_t^1 belong to the same path. In two cases the newly gained solution is invalid and can be ignored: First if the sum of the degrees is greater than two and second if we close a cycle, since this would create multiple cycles instead of one big cycle. For the second case one important exception exists: If we close a cycle in the root r, such that all vertices are on this cycle (meaning $X_r^0 = X_r^1 = \emptyset$), then we found a valid solution for Hamilton Cycle and are finished.

Partition-change-node. For partition-change-node t with child t' our underlying clique-partition can change and we thus need to adapt our partial solutions of t' to the new representatives. For this, we call the changed vertex v and call the clique v is assigned to in t C_2 and call the clique in t in which v was before C_1. In t' we have C_1' as the clique C_1 corresponds to and C_2' as the clique C_2 corresponds to (apart from the moving of v). In other words: The vertex v moves from C_1' in t' to C_2 in t, with C_1 and C_2' being the corresponding cliques in the other node.

Let us first consider the case where v is a non-representative node. Then, we can remove v from C_1' to get C_1 and can use partial solutions with the same representatives for C_1' and C_1, since v is not one of them. By adding v to C_2' to get C_2 we now get a new partial solution where v also is a non-representative vertex. The case where v is a representative of C_2 does not need to be considered, since it can also be reached by considering the case of v being a representative of C_1'. The second case which we consider is v being a representative in C_1'. Here, we consider all possible arrangements the edges of v can have in t' as subcases: These

are: Having zero edges, having one edge, which can either lead to a vertex of C_2' or to a vertex of a different clique, and having two edges where some of those edges can lead to a vertex of C_2'.

We start with v having zero edges. We then need to chose new representatives for C_1 and add v either as representative or non-representative for C_2. For choosing representatives of C_1, we again choose up to the maximum limit of representatives and discard any duplicate partial solutions. This gives us many new partial solutions.

The next case is v having one edge uv assigned to it. If $u \in C_2'$, then we add v as a representative for C_2, remove u as representative for C_2 and chose up to one new representative for C_1. Thus, the edge is also removed. Else we do the same steps, but do not remove u as representative and also keep the edge.

The final case is v having been assigned two edges. Here, we do not need to cover for any uncertainty of how the edges of v will be chosen, since v has already been assigned all its edges. We can use v as representative for C_2, remove the other endpoint(s) as representative(s) of C_2, if any of the endpoints belongs to C_2'. Again, we choose new representatives up to the maximal level. We also keep the edges of all non-removed vertices. Again, we discard any partial solutions that use more than two edges in between two cliques, since they cannot lead to valid solutions and the simplified variants, where we reduce the number of edges, can be gained through other partial solutions. These cases are exhaustive and we thus defined what to do for a partition-change-edge.

This also ends the description of our DP.

The description of representatives together with the explanation how we use them in our algorithm and the given DP complete the algorithm. We now need to show that our algorithm is in FPT for cp-treewidth. For this, we first need to bound the number of partial solutions we have for each bag and then can assemble the total running time. This is done in the following lemma and theorem.

Lemma 8.10: *Let $G = (V, E)$ be a graph with a given smooth cp-tree-decomposition (T, B, P) of weight* cptw *and clique-degree $\Delta(G, T, P)$. Let t be a node in T. Then, we need to consider at most $2^{\mathcal{O}(\Delta(G,T,P)\cdot \mathrm{cptw}\,\cdot\, \log(\Delta(G,T,P)\cdot \mathrm{cptw}))}$ partial solutions per bag.*

Proof. We bound the number of partial solutions by first bounding the number of representatives chosen and then bound the number of possible choices of representatives.

In each clique we choose at most two vertices per clique that neighbours it in the original graph. Since the number of those is bounded by $\Delta(G, T, P)$, we choose at most $R := 2\cdot\Delta(G, T, P)$ representatives from each clique. Thus, we can get the following upper bound for the number of sets of possible representatives we can choose in node t:

$$\begin{aligned}
\prod_{C\in\mathcal{P}_t} (|C|+1)^R &= \prod_{C\in\mathcal{P}_t} 2^{\log(|C|+1)^R} \\
&= 2^{\sum_{C\in\mathcal{P}_t} \log(|C|+1)^R} \\
&= 2^{R\cdot \sum_{C\in\mathcal{P}_t} \log(|C|+1)} \\
&\leq 2^{R\cdot \mathrm{cptw}}.
\end{aligned}$$

For the dynamic programming, which creates a number of partial solutions for each set of representatives, we can use the traditional estimations of $2^{\mathcal{O}(R\cdot p(B,P)\cdot\log(R\cdot p(B,P)))}$ where we can replace the treewidth with $R\cdot p(B,P)$, since we know that the bag has at most $R\cdot p(B,P)$ vertices. This traditional algorithm can be for example found in the paper of Ziobro et al. [ZP19] where it is called the naive approach.

The total number of partial solutions is thus given by the number of partial solutions (where each vertex is picked and how the vertices are connected) of the DP per set of representatives multiplied by the number of those sets and can be at most:

$$2^{\mathcal{O}(R\cdot p(B,P)\cdot\log(R\cdot p(B,P)))}\cdot 2^{R\cdot\mathrm{cptw}} = 2^{\mathcal{O}(R\cdot p(B,P)\cdot\log(R\cdot p(B,P)))+R\cdot\mathrm{cptw}}$$

We substitue R by its definition and get:

$$= 2^{\mathcal{O}(2\cdot\Delta(G,T,P)\cdot p(B,P)\cdot\log(2\cdot\Delta(G,T,P)\cdot p(B,P)))+2\cdot\Delta(G,T,P)\cdot\mathrm{cptw}}$$

Since we use $\mathcal{O}$-Notation, we can neglect the constant factors and simplify to:

$$= 2^{\mathcal{O}(\Delta(G,T,P)\cdot p(B,P)\cdot\log(\Delta(G,T,P)\cdot p(B,P)))+2\cdot\Delta(G,T,P)\cdot\mathrm{cptw}}$$

We now pull the second additive into the $\mathcal{O}$ to get:

$$= 2^{\mathcal{O}(\Delta(G,T,P)\cdot p(B,P)\cdot\log(\Delta(G,T,P)\cdot p(B,P))+\Delta(G,T,P)\cdot\mathrm{cptw})}$$

Using Lemma 3.6 we get:

$$\begin{aligned}
&\leq 2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw}\cdot\log(\Delta(G,T,P)\cdot\mathrm{cptw})+\Delta(G,T,P)\cdot\mathrm{cptw})}\\
&= 2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw}\cdot(1+\log(\Delta(G,T,P)\cdot\mathrm{cptw})))}\\
&= 2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw}\cdot\log(\Delta(G,T,P)\cdot\mathrm{cptw}))}.
\end{aligned}$$

This proves the lemma. ∎

We can use this lemma in our algorithm.

Theorem 8.11: *Let $G=(V,E)$ be a graph with a given cp-tree-decomposition (T,B,P) of weight* cptw*, width* tw *and clique-degree $\Delta(G,T,P)$. Then,* Hamilton Cycle *can be solved in* $2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw}\cdot\log(\Delta(G,T,P)\cdot\mathrm{cptw}))}\cdot\mathrm{tw}^2\cdot n^2$.

Proof. We begin by constructing a smooth cp-tree-decomposition and thus increase the weight to $2\cdot\mathrm{cptw}+2$. Then, we can use the algorithm given by considering the representatives and the dynamic programming defined above.

Using Lemma 8.10 we need to consider at most $2^{\mathcal{O}(\Delta(G,TP)\cdot\mathrm{cptw}\cdot\log(\Delta(G,T,P)\cdot\mathrm{cptw}))}$ partial solutions, since the constant factor can be neglected when dealing with $\mathcal{O}$-calculus. We, thus, gain with $\mathcal{O}(\mathrm{tw}^2\cdot n)$ bags and $\mathcal{O}(n)$ additional time the resulting total time. ∎

So far we have used the naive approach in our algorithm in order to make it easier to understand how a solution can be computed. We could instead use the rank-based-approach and thus get rid of the logarithmic factor. For this, we can replace the calculations done by the traditional algorithm in our DP with the rank-based-approach. Only for partition change nodes we need to make some modifications, since the other nodes make no changes to representatives that exceed adding or removing a single vertex. For those we can make the above modifications to representatives and then use the reduce algorithm to reduce the size of the modified partial solution. See Ziobro et al. [ZP19] for the running time.

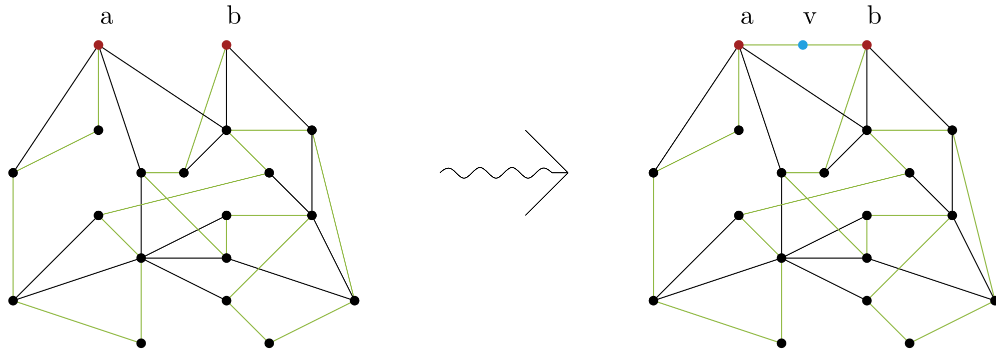


Figure 8.1.: The reduction applied to an example graph and a possible hamilton path of the graph in green. The endnodes are coloured red and the added vertex blue.

Corollary 8.12: *Let* $G = (V, E)$ *be a graph with a given cp-tree-decomposition* (T, B, P) *of weight* cptw*, width* tw *and clique-degree* $\Delta(G, T, P)$*. Then,* Hamilton Cycle *can be solved in* $2^{\mathcal{O}(\Delta(G,T,P)\cdot \mathrm{cptw})} \cdot \mathrm{tw}^2 \cdot n^2$.

We then can use the techniques of the polynomial reductions from Hamilton Path to Hamilton Cycle to gain an algorithm for the Hamilton Path problem[2].

Corollary 8.13: *Let* $G = (V, E)$ *be a graph with a given cp-tree-decomposition* (T, B, P) *of weight* cptw*, width* tw *and clique-degree* $\Delta(G, T, P)$*. Then,* Hamilton Path *can be solved in* $2^{\mathcal{O}(\Delta(G,T,P)\cdot \mathrm{cptw})} \cdot \mathrm{tw}^2 \cdot n^2$.

Proof. For this proof we show the reduction from Hamilton Path to Hamilton Cycle by first constructing a new instance in polynomial time and then showing that the reduction keeps the solution valid.

Let (G, cptw) be a instance of Hamilton Path with cp-treewidth cptw and given end-vertices $a, b \in V(G)$. We add a vertex $v \notin V(G)$ that is connected to a and b resulting in graph $G' = (V', E')$ with $V' = V(G) + v$ and $E' = E + va + vb$[3]. This results in the new instance $(G', \mathrm{cptw} + 1)$ since we can add v to each bag as a new clique, which gives us a cp-tree-decomposition of width at most $\mathrm{cptw} + \log(1 + 1) = \mathrm{cptw} + 1$. The additional constant gets lost in $\mathcal{O}$-Notation.

If we are given a yes-instance, then we know a Hamilton path from a to b exists and adding v completes it to a cycle.

If G' has a Hamilton cycle, a path from a to b is implied since the cycle needs to encompass v which is only connected to a and b and thus forces a Hamilton path from a to b.

Thus, an algorithm is given by performing the reduction and the solving Hamilton Cycle with Corollary 8.12. ∎

An example for this reduction can be seen in Figure 8.1.

[2]Note that our definition of Hamilton Path uses given endpoints of the path, while other definitions of the same problem require only a path between any vertices. By adding two vertices with edges to all of the old vertices and using them as endpoints a reduction between the two problems can be constructed. For the reduction in the other direction add degree-one-vertices to the given endpoints and thus force them to be chosen as endpoints.

[3]Kapamadzin [Kap15] uses a similar construction but add only a edge and not a whole vertex. The adding of a vertex eases the argument for the cp-tree-decomposition

8.3. A framework using clique-partitioned treewidth

Since many of our algorithms use similar concepts, we want to describe a framework that generalizes how we can find those algorithms. For this, note that the algorithms given before are faster, especially when they function without the use of smooth cp-tree-decompositions, but this framework can be used to describe algorithms for those problems and can be used on many other problems. This framework works similar to our algorithm for Hamilton Cycle and also uses the concepts of representatives. We thus generally describe how we can give an algorithm based on cp-treewidth for a graph problem. We begin by giving a framework for treewidth-based DPs and then proceed with our framework.

A framework for dynamic programs of tree decompositions. We define a framework for dynamic programs on traditional tree decompositions (T, B) of width tw with running time $2^{\mathcal{O}(f(\text{tw}))} \cdot n^{\mathcal{O}(1)}$ for computable function $f : \mathbb{R} \rightarrow \mathbb{R}$. A user of this framework needs to describe algorithms `leaf`, `introduceVertex`, `introduceEdge`, `forget`, join and functions `description`, `partialSolution`, `correct` for his problem and proof running time as well as correctness related statements. We explain the components that need to be defined for such an algorithm one by one. First the function defining all partial solutions `partialSolution` has to be defined. Here, `partialSolution` takes the graph induced by a bag $G[B(t)]$ and returns the set of all partial solutions of the node t, called M_t. Note, that p, a partial solution of node t, may only contain information available from $G[B(t)]$. Finally, `correct` takes a partial solution and returns a boolean that indicates whether this partial solutions is a correct partial solution on the graph $G[V_t]$, which is the subgraph of G induced by the nodes of T below t..

The dynamic program then consist of algorithms for each node type. Here, `leaf` is applied to a leaf-node t and returns a set of partial solution $X \subseteq M_t$ for input $G[B(t)]$. The algorithms `introduceVertex`, `introduceEdge` and `forget` are applied to introduce-, introduce-edge[4] and forget-nodes t with child t' respectively and return a set of partial solutions $X \subseteq M_t$ for inputs $p \in M_{t'}$ and $G[B(t)]$. Finally, join is applied to a join-node t with children t_1 and t_2 and returns a set of partial solutions $X \subseteq M_t$ for inputs $p_1 \in M_{t_1}$, $p_2 \in M_{t_2}$ and $G[B(t)]$. We define the set of valid partial solutions of node t, called F_t as the partial solutions of M_t that can be created by traversing the tree decomposition from any leaf to t and for every node applying the correct of the above functions to one of the partial solution generated in the step before. Thus, to be more exact, we only apply the above functions to partial solutions of F_t instead of M_t.

Finally, a function `description` that assigns each partial solution a bitstring is needed. This bitstring is called the description of the partials solution and may have size polynomial in the instance size and can contain any additional information that needs to be transferred with the partial solution. This function may be used in `leaf`, `introduceVertex`, `introduceEdge`, `forget` and join. The description of a partial solution may also be recursively calculated in the process of building it.

To ensure a correct algorithm we require an instance of the problem to be a yes-instance if and only if there is a $p \in F_r$ with `correct`(p) being true for the root r. Due to this the framework is correct if it outputs yes if the above condition is fulfilled. To achieve the running time, we require $|M_t| \in 2^{\mathcal{O}(f(\text{tw}))}$ for each node t and `leaf`, `introduceVertex`, `introduceEdge`, `forget`, join $\in n^{\mathcal{O}(1)}$.

[4]In the case of nice cp-tree-decompositions `introduceEdge` is omitted. In the remainder of this subsection it will not be mentioned but still holds.

Examples. We give two examples of how these frameworks can be used. For Independent Set we can use as partial solutions the intersection of the bag and a subset of V. The description of a partial solution is the number of vertices picked so far and is calculated in the DP. We then define the correct partial solutions as those, whose description number is greater than k using the integer k that gives the minimal size of independent sets allowed in this instance. For the other algorithms we can use the commonly known descriptions of the DP, see for example Cygan et al. [Cyg+15][5].

For Hamilton Cycle we can define a partial solution as a structure that defines for each vertex of $B(t)$ to which vertices it is connected by a path and which degree it has. Additionally, using the description we store if the partial solution is still valid. A partial solution becomes invalid, if any vertices are forgotten that are assigned less or more than two edges. Again, the remaining algorithms can be found in literature.

We can use this to give our framework. We first give an informal explanation and then proceed with a formal description, the explanation of partition-change-nodes and an example.

Informal Explanation. This framework is a dynamic program on a smooth or extended smooth cp-tree-decomposition. Here, the type of cp-tree-decomposition depends on the type of tree decomposition used in the traditional algorithm that forms the basis of the DP we construct. The main idea is to choose up to R so called representatives from each clique and run the traditional DP on all possible choices. Here, it is important that we can reassemble a partial solution for the whole bag from a partial solution on the graph containing only the representatives in polynomial time. Our partial solutions will be a set of up to R representatives and the partial solution of the traditional DP applied to those representatives. The algorithm will, for each bag, first create all possible sets of representatives and then run the traditional algorithm on each possible choice of representatives. We can then proceed in the necessary fashion with the partial solutions we have created in this way.

Formal description. We define the framework for a dynamic program on the cp-tree-decomposition (T, B, P). The framework uses a smooth cp-tree-composition if the underlying traditional algorithm uses a nice tree decomposition and a smooth extended cp-tree-decomposition for an extended cp-tree-decomposition. We build a new DP on the cp-tree decomposition, which we annotate with $'$ and for this use some of the already known definitions on the representatives and on the original graph. First, to use this framework, an algorithm fulfilling the above framework for treewidth needs to be given. Furthermore, algorithms `represent` and `reassemble` as well as correctness and running time related proofs need to be given. We begin by giving the requirements for the first of those two algorithms. We define `represent` as a function that takes $G[B(t)]$ and returns a set, called S_t, containing subgraphs of $G[B(t)]$ with at most R vertices of each clique each. We can define the output of `partialSolution'` as the set of all (s, p), where $s \in S_t$ and $p \in$ `partialSolution`(s). Due to our naming conventions, this is called M'_t. We can use this to define how `reassemble` has to look. We require `reassemble` to take a partial solution on the representatives $(s, p) \in M'_t$ and return a partial solution on the whole bag $q \in M_t$. Here, `reassemble` also takes the description of (s, p), which is given by the DP or by applying `description` to the partial solution on the representatives, and returns the description of q. We then define `correct'` as the function that takes a partial solution $(s, p) \in M'_t$ of a node t and returns the boolean

[5] This description is for the weighted case. Set the weight of each vertex to one to get the unweighted.

value of correct(reassemble(s, p)). Now, we can give the remaining algorithms. When we apply leaf′ to a leaf-node t, it runs represent on $G[B(t)]$, generating S_t, and then applies leaf on each element of S_t to receive, combined with the representatives, the elements of the output set. When we apply introduceVertex′ to an introduce-node t, it runs represent on $G[B(t)]$, generating S_t, and then applies introduceVertex on each element of S_t as well as the inputed partial solution of the child to receive, combined with the representatives, the elements of the output set. If the representatives do not change, nothing needs to be done and the partial solution of the child can be reused. Note, that due to the use of smooth cp-tree-decompositions only one new representative or no change of representatives is possible. The functions introduceEdge′, forget′ and join′ are constructed in analogue fashion.

Here, for correctness, we require, as before, an instance of the problem to be a yes-instance if and only if there is a $p \in F'_r$ with correct′(p) being true for the root r[6]. To achieve the running time we require $|s| \leq R, s \in S_t$, for each node t and represent, reassemble $\in n^{\mathcal{O}(1)}$. Thus, a user of the framework has to give polynomial algorithms represent and reassemble and prove that represent only outputs small enough sets and reassemble guarantees that correctness of the traditional DP is equivalent to correctness of the cp-treewidth based one. Informally speaking, reassemble turns a valid partial solution into a valid one and correct partial solution into a correct one. Formally, two aspects need to be considered. First, if and only if reassemble is given a valid partial solution, meaning one that is in F'_t for current node t, it also returns a valid one, meaning one from F_t. Second, given a partial solution p, with correct(p) being true, that was constructed by the traditional algorithm applied to the graph using the operations $o_1, \dots, o_l$ ($o_i \in$ {leaf, introduceVertex, introduceEdge, forget, join, } for $i \in [l]$), then there exists a choice of representatives of each bag, with s containing the final choice of representatives, such that the partial solution (s, p') also has correct′(s, p') true. Here, p' is constructed using $o'_1, \dots, o'_l$ and any choice of the representatives given by represent for each bag. The other direction needs to also hold[7].

We call a graph problem (R, f)-*representable*, if this can be proven. Note, that R is the upper bound for the number of representatives per clique and f is the function influencing the running time of the traditional DP.

Partition-change-nodes. To ensure transitioning between the representatives of two adjacent nodes is possible we need partition-change-nodes. We describe their usage. For partition-change-nodes we need to adapt the partial solutions from one bag to another. Here, we consider two cases. First, if the vertex is a representative before and after the change of clique, there is no change in representatives and thus the traditional algorithm notices no differences and can work with the old partial solutions on the representatives. This is since the traditional algorithm has no knowledge to which clique each representative belongs and thus changing cliques cannot be noticed by it. Thus, the change of clique is absorbed by the usual assembling of a partial solution for the whole bag from the partial solution of the representatives. In the other case the change is absorbed by our choice of representatives. To the traditional algorithm this looks like a vertex was either added or removed or no change was made. If no change was made, the algorithm can just use the old partial solutions and do no calculations. Note that an alternative, but equivalent point of view, is that when focusing in on

[6] F'_t is defined just as F_t but uses the ′ variants.

[7] Observe, that the following statement, which might be a first idea, is not enough for correctness, since it is based on circular reasoning. This is since we defined correct′ by running correct on the reassembled partial solution: Formally, this means that given a partial solution $(s, p') \in M'_t$ with correct′(s, p') being true reassemble returns a partial solution p with correct(p) being true. And the analogue for a non-correct partial solution.

the representatives this process can be seen as removing the changed vertex as representative of the old clique and adding it as a representative to the new. Also new representatives might be chosen and all partial solutions with to many representatives are discarded.

Example. Using Hamilton Cycle as an example we choose $2 \cdot \Delta(G, T, P)$ vertices from each clique. Thus, `represent` returns all subsets of the bag that contain less than or equal to $2 \cdot \Delta(G, T, P)$ vertices from each clique. We then use those vertices as entries into the cliques and can, in `reassemble`, add all non-representatives by replacing an edge between two representatives of a clique with a path of non-representatives. Here, Lemma 8.9 shows that we can always find such an edges if an hamilton cycle exists. Thus, a (partial) hamilton cycle on the representatives can be turned into a (partial) hamilton cycle on all vertices. By using the lemma for each step it can be shown that the correctness criterion for `reassemble` is fulfilled, since a general hamilton cycle, as found by the traditional algorithm, exists if and only if a special hamilton cycle, as found by the algorithm using representatives, exists. Both, `represent` and `reassemble`, are obviously polynomial. Since a $2^{\mathcal{O}(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm exists (see for example [ZP19]), Hamilton Cycle is $(2 \cdot \Delta(G, T, P), x \longrightarrow x)$-*representable*[8].

Finally, we can analyse the framework. We begin with the running time of this framework.

Theorem 8.14: *Let* $G = (V, E)$ *be a graph with a given (extended) smooth cp-tree-decomposition* (T, B, P) *of weight* cptw *and width* tw*. Then, a* (R, f)*-representable graph problem* Π_{cptw} *can be solved in* $2^{\mathcal{O}(f(R \cdot \text{cptw}) + R \cdot \text{cptw})} \cdot n^{\mathcal{O}(1)}$.

Proof. Since we can handle each partial solution in polynomial time, we need to analyse the number of partial solutions in each bag and the time spend by the traditional algorithm to prove this theorem. This number consists of the number of representatives and the number of partial solutions of the traditional algorithm. The latter is implicitly handled by the traditional algorithm and thus we do not need to consider more than the running time of that algorithm. We can bound the number of representatives in node t by using the fact that we chose at most R representatives from each clique:

$$\begin{aligned}\prod_{C \in \mathcal{P}_t} (|C| + 1)^R &= \prod_{C \in \mathcal{P}_t} 2^{\log(|C|+1)^R} \\ &= 2^{\sum_{C \in \mathcal{P}_t} \log(|C|+1)^R} \\ &= 2^{R \cdot \sum_{C \in \mathcal{P}_t} \log(|C|+1)} \\ &\leq 2^{R \cdot \text{cptw}}.\end{aligned}$$

Together with the running time of $2^{\mathcal{O}(f(R \cdot \text{cptw}))} \cdot n^{\mathcal{O}(1)}$, which is the time the traditional algorithms spends on the representatives, we get the above bound. For this, note that in each bag we have at most $R \cdot p(B, P) \leq R \cdot \text{cptw}$ (Lemma 3.6) representatives. Also note, that we have only polynomially many bags. ∎

[8] Read this as *delta-linear-representable*. Other examples of proposed reading conventions are *1-log-representable* for $(1, x \longrightarrow \log(x))$-representable and *p-quadratic-representable* for $(p(B, P), x \longrightarrow x^2)$-representable.

We can apply this framework to all problems for which we gave algorithms. For example consider Independent Set. Here, we can pick at most one vertex from each clique as representative, leave all non-representatives out of the partial solution, reuse the weight of the traditional partial solution as its description and use the traditional $2^{\mathcal{O}(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm [Cyg+15] to get that Independent Set is $(1, x \to x)$-*representable.*

Finally, observe that each (R, f)-representable problem can be solved in FPT-time when parametrizing by cp-treewidth. We can also set $R = \text{tw}$ and gain a slow algorithm similar to Corollary 4.5 . Generally it will be useful to find small R and f.

8.4. Applying the framework

We apply this framework to some problems to show how it can be used and that rather complex processes of reassembling are possible. We will consider three example problems not considered so far and explain how we can fit them into the framework.

8.4.1. Vertex Adjacent Feedback Edge Set

In this subsection we consider the Vertex Adjacent Feedback Edge Set problem. This is a variant in which we want to find feedback edge sets with a minimum number of incident vertices. For this problem we use the framework we did introduce.

Theorem 8.15: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw. *Assume G has neighbourhood index $\delta(G, T, B, P)$. Then, Vertex Adjacent Feedback Edge Set can be solved in $2^{\mathcal{O}(\delta(G,T,B,P)\cdot \text{cptw})} \cdot n^{\mathcal{O}(1)}$.*

Proof. We thus show that Vertex Adjacent Feedback Edge Set is $(\delta(G, T, B, P), x \to x)$-representable and apply Theorem 8.14. We can pick all vertices that have neighbours outside of the clique as representatives. Those are at most $\delta(G, T, B, P)$ and we thus have given `represent` and shown that the subgraphs generated by `represent` are small enough. Thus, we only need to consider the cycles inside cliques. Here, we can pick at most two edges from each clique since otherwise, we get a cycle. Thus, we can reassemble the partial solutions by removing as few edges between non-representatives, that are inside a clique, as possible, without having more than two edges remaining. This gives `reassemble`. This function fulfils our correctness criteria, since the solutions created by `reassemble` are valid if and only if the input is valid. Here, valid means that they do not contain cycles and this is achieved by ensuring that the non-representatives create no cycles. The other criterion is also fulfilled since we can always choose as representative those nodes chosen by the traditional algorithm that also have neighbours outside of the clique.

For the traditional algorithm we use the $2^{\mathcal{O}(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm mentioned in the work of Zhang et al. [ZLS04]. ∎

8.4.2. Colour

Next, we consider the Colour problem. While every graph with a given tree decomposition of width tw can be coloured with tw +1 colours [Mar20a], the number of colours needed to colour a graph can be exponential in its cp-treewidth. To prove this, consider cliques. Thus, designing a cp-treewidth-based algorithm is more difficult. We use our framework to find an algorithm.

Theorem 8.16: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and with neighbourhood index $\delta(G, T, B, P)$. Then,* Colour *can be solved in* $2^{\mathcal{O}(\delta(G,T,B,P)\cdot\text{cptw}\cdot\log(\delta(G,T,B,P)\cdot\text{cptw}))}\cdot n^{\mathcal{O}(1)}$.

Proof. We can use the neighbourhood index to bound the number of representatives since this value bounds the number of vertices that are connected to other cliques. Furthermore, we can use the traditional $\text{tw}^{\text{tw}}\cdot n^{\mathcal{O}(1)}$ algorithm [Mar20a]. By colouring non-representatives with colours not yet used by the representatives, we can reassemble a partial solution for the whole bag. Here, colour each non-representative with a different colour and begin by using already existing, but in this clique not used colours, and then use new colours and adjust the weight. This can obviously be done in a BFS search. This gives `represent` and `reassemble`. Since valid solutions of the traditional DP are all assignments of colours to vertices, the criterion for creating valid solutions is fulfilled. For creating correct solutions, we can use as representatives the vertices with neighbours outside of the clique and get a correct solution. Thus, Colour is $(\delta(G, T, B, P), x \to x \cdot \log(x))$-representable and we can apply Theorem 8.14 to get[9] the theorem. ∎

By using the $k^{\text{tw}}\cdot n^{\mathcal{O}(1)}$ algorithm [Mar20a] for the traditional problem in the above proof, we can reduce the runtime bound for k-Colour.

Corollary 8.17: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw*. Then, k-*Colour *can be solved in* $2^{\mathcal{O}(\delta(G,T,B,P)\cdot\text{cptw}\cdot\log(k))}\cdot n^{\mathcal{O}(1)}$.

8.4.3. Max Cut

As the final of the three problems we want to consider MaxCut which is one of Karp's 21 problems [Kar10] and is of interest since the process of reassembling is not straight forward. Here, the task is to find a vertex set that maximises the number of edges from inside to outside.

Theorem 8.18: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*. Assume G has neighbourhood index of $\delta(G, T, B, P)$. Then,* Max Cut *can be solved in* $2^{\mathcal{O}(\delta(G,T,B,P)\cdot\text{cptw})}\cdot\text{tw}^2\cdot n^2$.

Proof. For this, we show that MaxCut is $(\delta(G, T, B, P), x \to x)$-representable and then apply Theorem 8.14. We choose all vertices in the clique that neighbour a vertex outside the clique as representative and thus have at most $\delta(G, T, B, P)$ representatives. Since a partial solution of the bag consists of the vertices picked into the set X, which induces the cut, we can calculate the value of this partial solution the following way. For node t, we first split the edges that lead from X to $B(t) - X$ into two categories. First those for whom both endpoints are in the same clique and those for whom they are in different cliques. The value of the first type can simply be calculated by $|X| \cdot |B(t) - X|$ since we connect every vertex of X to every vertex of $B(t) - X$. The second one is considered by the algorithm of the framework. Informally speaking, those nodes that are not chosen as representatives have no edges leading outside of the clique and thus we only need to optimize the number of such edges. We obtain the weight by adding the highest total value of edges of the first type to the weight calculated on the representatives and gain a partial solution by adding the non-representatives that induce this highest value. This means for each partial solution on the representatives we find the set of non-representatives such that the gained value is maximal. For this, note that the value is

[9] The increase of weight due to the usage of smooth cp-tree-decompositions is absorbed by the $\mathcal{O}$-Calculus.

maximal when the number of picked vertices is close to half. We thus can efficiently calculate the value by searching through the number of non-representatives. Again, this describes the needed algorithms and explains correctness. Finally, we use the traditional algorithm by Bodlaender and Jansen [BJ94].

∎

It is rather obvious that this can be extended to the weighted case by using a traditional algorithm that handles weights[10] and using weights in the calculation.

Corollary 8.19: *Let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight* cptw *and width* tw*. Assume G has neighbourhood index of $\delta(G, T, B, P)$. Then,* Weighted Max Cut *can be solved in $2^{\mathcal{O}(\delta(G,T,B,P)\cdot \text{cptw})} \cdot \text{tw}^2 \cdot n^2$.*

8.5. How clique-partitioned treewidth and BBKMZ-treewidth algorithms relate to each other

In this section we want to compare the BBKMZ-treewidth and cp-treewidth and find out if there are problems for which the dependence on cp-treewidth is higher than on BBKMZ-width. Note that the currently known BBKMZ-treewidth-based algorithms for the problems of Chapter 6 and Section 7.5 also use those additional factors. We begin by proving that as long as we can find a BBKMZ-algorithm that fulfils certain criteria, we can also find a cp-treewidth algorithm of the same running time.

Theorem 8.20: *Let $G = (V, E)$ be a graph with a given BBKMZ-tree-decomposition (T, B, P) of BBKMZ-weight* bbkmz *and width* tw*. Let Π_{bbkmz} be a graph problem parametrized by BBKMZ-treewidth. Let $f : \mathbb{R} \to \mathbb{R}$ be a function such that Π_{bbkmz} has an $2^{f(R\cdot \text{bbkmz})} \cdot n^{\mathcal{O}(1)}$ algorithm, which can be described as a dynamic program on the tree decomposition where only partial solutions are considered that pick at most R representatives from each clique. Let this algorithm use a traditional $2^{f(\text{tw})} \cdot n^{\mathcal{O}(1)}$ algorithm, fulfilling our framework, when parametrizing by treewidth. Then, an algorithm solving Π_{cptw} parametrized by cp-treewidth* cptw *in time $2^{f(R\cdot \text{cptw})} \cdot n^{\mathcal{O}(1)}$ exists.*

Proof. We show that the problem is (R, f)-representable and then use Theorem 8.14. We can use the number of representatives for R and use the traditional algorithm the bbkmz-algorithm is based on. We give `represent` as the function that returns all subsets with less than R vertices and give `reassemble` by performing the same operations as the BBKMZ-algorithm. ∎

Note that all currently known algorithms using the BBKMZ-treewidth are of this type. If we try to generalize this to all algorithms, we run into some problems. First, it is difficult to extract an upper bound from the generalized running time, since we only know a bound on the number of partial solutions. And second, it is not obvious that we can find a related algorithm on traditional treewidth that has the needed running time.

We nonetheless state this conjecture and afterwards explain some reasons why we believe this to be true.

[10] Lokshtanov et al. [LMS18] prove that for the weighed case the current algorithm is optimal under SETH and we use this bound.

Conjecture 8.21: *Let $G = (V, E)$ be a graph with a given BBKMZ-tree-decomposition (T, B, P) of BBKMZ-weight* bbkmz *and width* tw. *Let* Π_{bbkmz} *be a graph problem parametrized by BBKMZ-treewidth. Let $f : \mathbb{R} \to \mathbb{R}$ be a function such that* Π_{bbkmz} *has an* $2^{f(\mathrm{bbkmz})} \cdot n^{\mathcal{O}(1)}$ *algorithm. Then, an algorithm solving* Π_{cptw} *parametrized by cp-treewidth* cptw *in time* $2^{f(\mathrm{cptw})} \cdot n^{\mathcal{O}(1)}$ *exists.*

Here, we think that the most likely way to prove this is to show that the problem is (R, f)-representable for the given R and f. This conjecture is supported by four ideas. First, we know that all algorithms of currently known types or concepts fulfil Theorem 8.20 and thus support the conjecture. Second, the only difference between BBKMZ-treewidth and cp-treewidth is the fact that the partition of the bags in the BBKMZ-tree-decomposition can be reused for all bags since it is global and for cp-treewith this is not directly possible. This difference is bridged by the smooth cp-tree-decomposition and thus a problem falsifying the conjecture needs to rely on the partition being the same and does not allow for any of the adaptations of smooth cp-tree-decompositions. Third, to build a BBKMZ-treewidth-based algorithm that is faster than the treewidth-based one, it is necessary to use the additional structure given by the clique-partition. This usage could then be replicated by the cp-treewidth. And finally, we know that a problem is in FPT when parametrizing by cp-treewidth if and only if it is in FPT when parametrizing by treewidth (Corollary 4.4). This result can be extended to BBKMZ-treewidth by using the comparisons of Chapter 3 and thus a treewidth-based algorithm exists, if a BBKMZ-treewidth-based one exists. This lets us hope for a related traditional algorithm of the same running time.

These arguments can hopefully be used to in the future prove this conjecture or prove some preliminary results that allow for longer running times in the created algorithm or add more requirements on the BBKMZ-algorithm.

9. Conclusion

In this thesis we have discussed the algorithmic potential and limits of cp-treewidth. Here, we have considered three main objectives. First to compare our parameter to other similar parameters. Second, to find algorithms using the parameter. And third, to establish lower bounds and thus show how far we can get. An overview of the running times of our algorithms as well as our lower bounds can be found in Table 9.1.

For the first task, we have found comparisons to a large range of parameters and have shown that we can sort those parameters into two groups. Those that depend on the size of the cliques in their definitions and those that do not. We found that cp-treewidth is generally the lowest of the first group and that finding a general bound of cp-treewidth by a member of the second one is impossible.

In terms of solving problems, we gave parametrized algorithms for many problems and showed that finding parametrized algorithms for cp-treewidth is possible if and only if this is possible for treewidth. For this, we first laid some ground work and adapted nice and extended tree decompositions to cp-treewidth. Furthermore, we introduced a new normalization step called smooth cp-tree-decomposition which allows us to link clique-partitions of different bags. Those normalizations allowed us to describe our first slate of algorithms, but to proceed for more problems we needed (as we showed later on) additional parameters. We introduced the clique-degree and the neighbourhood index as such parameters and showed that computing the former is $\mathcal{NP}$-complete and that, in a certain way, it can be considered to be lower than the corresponding parameter used by de Berg et al. [Ber+18] in their definition of BBKMZ-treewidth. These parameters then allowed us to find algorithms for more problems. We then summarized our results by giving a general framework and thus generalizing our approach.

The final topic of our work were lower bounds. Here, we proved that we can adapt the lower bounds of treewidth to cp-treewidth and used this to prove many of our algorithms optimal under ETH. Additionally, we showed that for many problems no $2^{\left(\mathrm{cptw}^{\mathcal{O}(1)}\right)} \cdot n^{\mathcal{O}(1)}$ algorithm can be found without the use of a second parameter, again assuming ETH. In this process we analysed different subclasses of graphs and proved similar results for those.

9.1. Future Work

While this work has explored cp-treewidth and furthered the understanding of this parameter, many questions are still open for research. A first open question, that can be asked for almost every similar work, is whether fast algorithms for more problems can be found. So far we did discover that algorithms parametrized by cp-treewidth could be found when algorithms for treewidth can be found. Since those algorithms are as fast as the traditional algorithm and thus of no practical use, it remains to be seen for which problems we can find faster algorithms. With our framework we established a general approach for finding such algorithms. Thus, for this topic two tasks remain. First, to find more problems to which the framework can be

applied by showing (R, f)-representability and second, to close the gap between lower bounds and the framework by either showing that all problems that are not (R, f) representable have no fast algorithm or extending the framework to cover the remaining problems.

Another open question concerns the parameter $\Delta(G, T, P)$, since we so far have only given proofs for the necessity of any additional parameter for finding fast algorithms and shown that with the current approach this parameter is necessary. Thus, it might be of interest to search for a reduction or other proof that shows that no algorithm with a substantially smaller parameter can be found or to search for such an algorithm.

Similarly, showing more bounds is still possible. On the one hand, one can improve the already existing bounds and thus close gaps between the best known algorithms with and without additional parameters. For example it might be possible to find a linear reduction for Dominating Set or Steiner Tree and thus close the currently existing gap. On the other hand, one can search for new kinds of bounds like bounds on the base of the exponential terms. Such bounds have been shown for many problems parametrized by treewidth with the use of SETH (see [LMS18] for examples) and thus it might be of interest to search for such bounds for this parameter.

Finally, we did show that we can find a corresponding cp-treewidth algorithm for each BBKMZ-treewidth algorithm of the current architecture and have stated as conjecture that this is also possible for all other algorithms. Proving this conjecture would be a major step in understanding the relationship between the two parameters.

As always, new questions might arise during the process of answering the above ones.

Table 9.1.: This table provides a summary of the running times of our algorithms for each problem and the lower bound, for which algorithms can exists (under ETH). For this, let $G = (V, E)$ be a graph with a given cp-tree-decomposition (T, B, P) of weight cptw, clique-degree $\Delta(G, T, P)$ and neighbourhood index $\delta(G, T, B, P)$. For simplicity sake we omit the polynomial in n term and use a bound that is expressible in a simpler manner for 3-Clique Cover (see Section 4.3 for a tighter bound). For some problems we used stronger lower bounds that are only valid when we parametrize by cp-treewidth alone and allow for no additional factor (see Chapter 6 and Section 7.5). Due to this the given algorithms, that use additional parameters do not break those bounds. Also note that for those problems were we did not give a lower bound, we could not find a lower bound for the treewidth-based problem.

Problem	Algorithm	Bound
Independent Set	2^{cptw}	$2^{\Omega(\mathrm{cptw})}$
Weighted Independent Set	2^{cptw}	$2^{\Omega(\mathrm{cptw})}$
Vertex Cover	2^{cptw}	$2^{\Omega(\mathrm{cptw})}$
Connected Vertex Cover	$2^{\mathcal{O}(\mathrm{cptw})}$	$2^{\Omega(\mathrm{cptw})}$
Maximum Induced Forest	$2^{\mathcal{O}(\mathrm{cptw})}$	$2^{\Omega(\mathrm{cptw})}$
Feedback Vertex Set	$2^{\mathcal{O}(\mathrm{cptw})}$	$2^{\Omega(\mathrm{cptw})}$
Connected Feedback Vertex Set	$2^{\mathcal{O}(\mathrm{cptw})}$	$2^{\Omega(\mathrm{cptw})}$
Dominating Set	$2^{3\cdot\Delta(G,T,P)\cdot\mathrm{cptw}}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
r-Dominating Set	$2^{3\cdot\Delta(G,T,P)\cdot\mathrm{cptw}}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Connected Dominating Set	$2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Steiner Tree	$2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Weighted Steiner Tree	$2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Clique	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Hamilton Cycle	$2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\mathrm{cptw}}\right)}$
Hamilton Path	$2^{\mathcal{O}(\Delta(G,T,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\mathrm{cptw}}\right)}$
Vertex Adjacent Feedback Edge Set	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw})}$	-
Colour	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw}\cdot\log(\delta(G,T,B,P)\cdot\mathrm{cptw}))}$	$2^{\Omega\left(2^{\mathrm{cptw}}\right)}$
k-Colour	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw}\cdot\log(k))}$	$2^{\Omega\left(2^{\mathrm{cptw}}\right)}$
Max Cut	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
Weighted Max Cut	$2^{\mathcal{O}(\delta(G,T,B,P)\cdot\mathrm{cptw})}$	$2^{\Omega\left(2^{\sqrt{\mathrm{cptw}}}\right)}$
3-Clique Cover	$2^{\mathcal{O}(2^{\mathrm{cptw}})}$	$2^{\Omega\left(2^{\left(\mathrm{cptw}^{1/(1+\varepsilon)}\right)}\right)}$

Bibliography

[AH76] K. Appel and W. Haken. “Every planar map is four colorable”. In: *Bulletin of the American Mathematical Society* Volume 82 (1976), pp. 711–712. DOI: *10.1090/s0002-9904-1976-14122-5*.

[Aig95] Martin Aigner. “Turán’s Graph Theorem”. In: *The American Mathematical Monthly* Volume 102 (1995), pp. 808–816. ISSN: 00029890, 19300972.

[AJKL22] Jungho Ahn, Lars Jaffke, O-joung Kwon, and Paloma T. Lima. “Well-partitioned chordal graphs”. In: *Discrete Mathematics* Volume 345 (2022), p. 112985. ISSN: 0012-365X. DOI: *https://doi.org/10.1016/j.disc.2022.112985*.

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

[Aya] Hanan Ayad. “Independent Set and Vertex Cover”. In: ().

[AZ19] Ernst Althaus and Sarah Ziegler. “Optimal Tree Decompositions Revisited: A Simpler Linear-Time FPT Algorithm”. In: *CoRR* Volume abs/1912.09144 (2019). arXiv: *1912.09144*.

[Bar05] David Mix Barrington. “CMPSCI 611: Graduate Theory of Algorithms: Solutions to Practice Final Exam”. 2005. URL: *https://people.cs.umass.edu/~barring/cs611/exams/finpracsol.html*.

[BCKN15] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. “Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth”. In: *Information and Computation* Volume 243 (2015), pp. 86–111. ISSN: 0890-5401. DOI: *https://doi.org/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* Volume abs/2302.08870 (2018), pp. 574–586. DOI: *10.1145/3188745.3188854*.

[BJ00] Hans L. Bodlaender and Klaus Jansen. “On the Complexity of the Maximum Cut Problem”. In: *Nordic J. of Computing* Volume 7 (Mar. 2000), pp. 14–31. ISSN: 1236-6064.

[BJ94] Hans L. Bodlaender and Klaus Jansen. “On the complexity of the maximum cut problem”. In: *STACS 94*. Edited by Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 769–780. ISBN: 978-3-540-48332-8.

[BKW23] Thomas Bläsius, Maximilian Katzmann, and Marcus Wilhelm. “Partitioning the Bags of a Tree Decomposition Into Cliques”. In: *Computing Research Repository (CoRR)* Volume abs/2302.08870 (2023). DOI: *10.48550/arXiv.2302.08870*.

[Blä21a] Thomas Bläsius. *Parametrisierte Algorithmen: Baumweite und dynamische Programme auf Baumzerlegungen*. 2021.

[Blä21b] Thomas Bläsius. *Parametrisierte Algorithmen: Untere Schranken: ETH und SETH.* 2021.

[BM19] Joshua Brakensiek and Weiyun Ma. "Lecture 17: The Strong Exponential Time Hypothesis". 2019. URL: *http://web.stanford.edu/class/cs354/scribe/lecture17.pdf*.

[BM90] Hans Bodlaender and Rolf Möhring. "The Pathwidth and Treewidth of Cographs". In: vol. 6. Jan. 1990, pp. 301–309. DOI: *10.1137/0406014*.

[BP83] R. Balakrishnan and P. Paulraja. "Powers of chordal graphs". In: *Journal of the Australian Mathematical Society* Volume 35 (1983), pp. 211–217. DOI: *10.1017/S1446788700025696*.

[BST20] Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. "A complexity dichotomy for hitting connected minors on bounded treewidth graphs: the chair and the banner draw the boundary". In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2020, pp. 951–970. eprint: *https://epubs.siam.org/doi/pdf/10.1137/1.9781611975994.57*.

[CEF12] Yijia Chen, Kord Eickmeyer, and Jörg Flum. "The Exponential Time Hypothesis and the Parameterized Clique Problem". In: *Parameterized and Exact Computation.* Edited by Dimitrios M. Thilikos and Gerhard J. Woeginger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 13–24. ISBN: 978-3-642-33293-7.

[Cer+08] M.R. Cerioli, L. Faria, T.O. Ferreira, C.A.J. Martinhon, F. Protti, and B. Reed. "Partition into cliques for cubic graphs: Planar case, complexity and approximation". In: *Discrete Applied Mathematics* Volume 156 (2008), pp. 2270–2278. ISSN: 0166-218X. DOI: *https://doi.org/10.1016/j.dam.2007.10.015*.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. "The Complexity of Satisfiability of Small Depth Circuits". In: *Parameterized and Exact Computation.* Edited by Jianer Chen and Fedor V. Fomin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 75–85. ISBN: 978-3-642-11269-0.

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: *10.1145/800157.805047*.

[Cou90] Bruno Courcelle. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs". In: *Information and Computation* Volume 85 (1990), pp. 12–75. ISSN: 0890-5401. DOI: *https://doi.org/10.1016/0890-5401(90)90043-H*.

[CP84] D.G. Corneil and Y. Perl. "Clustering and domination in perfect graphs". In: *Discrete Applied Mathematics* Volume 9 (1984), pp. 27–39. ISSN: 0166-218X. DOI: *https://doi.org/10.1016/0166-218X(84)90088-X*.

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

[DF13] Rodney G. Downey and Michael R. Fellows. "Courcelle's Theorem". In: *Fundamentals of Parameterized Complexity.* London: Springer London, 2013, pp. 265–278. ISBN: 978-1-4471-5559-1. DOI: *10.1007/978-1-4471-5559-1_13*.

[Dir61] G. A. Dirac. "On rigid circuit graphs". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* Volume 25 (1961), pp. 71–76.

[DMR05] Irit Dinur, Elchanan Mossel, and Oded Regev. "Conditional Hardness for Approximate Coloring". In: *SIAM Journal on Computing* Volume 39 (May 2005). DOI: *10.1145/1132516.1132567*.

[DMŠ22] Clément Dallard, Martin Milanič, and Kenny Štorgel. *Treewidth versus clique number. II. Tree-independence number*. 2022. arXiv: *2111.04543*.

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

[Fal+21] Joshua Fallon, Kirsten Hogenson, Lauren Keough, Mario Lomelí, Marcus Schaefer, and Pablo Soberón. *A Note on the Maximum Rectilinear Crossing Number of Spiders*. 2021. arXiv: *1808.00385*.

[Fei98] Uriel Feige. "A Threshold of Ln n for Approximating Set Cover". In: *J. ACM* Volume 45 (July 1998), pp. 634–652. ISSN: 0004-5411. DOI: *10.1145/285055.285059*.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. "Improved Approximation Algorithms for Minimum-Weight Vertex Separators". In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 563–572. ISBN: 1581139608. DOI: *10.1145/1060590.1060674*.

[GHY93] Olivier Goldschmidt, Dorit S. Hochbaum, and Gang Yu. "A modified greedy heuristic for the Set Covering problem with improved worst case bound". In: *Information Processing Letters* Volume 48 (1993), pp. 305–310. ISSN: 0020-0190. DOI: *https://doi.org/10.1016/0020-0190(93)90173-7*.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[GJS76] M.R. Garey, D.S. Johnson, and L. Stockmeyer. "Some simplified NP-complete graph problems". In: *Theoretical Computer Science* Volume 1 (1976), pp. 237–267. ISSN: 0304-3975. DOI: *https://doi.org/10.1016/0304-3975(76)90059-1*.

[HH00] Andrew Hodges and Douglas Hofstadter. *Alan Turing: The Enigma*. Walker and Company, 2000. ISBN: 0802775802.

[htt] Rotenberg (https://math.stackexchange.com/users/242055/rotenberg). *Reduction from Hamiltonian cycle to Hamiltonian path*. eprint: *https://math.stackexchange.com/q/1290804*.

[HW12] Lane A. Hemaspaandra and Ryan Williams. "SIGACT News Complexity Theory Column 76: An Atypical Survey of Typical-Case Heuristic Algorithms". In: *SIGACT News* Volume 43 (Dec. 2012), pp. 70–89. ISSN: 0163-5700. DOI: *10.1145/2421119.2421135*.

[IK10] Hiro Ito and Masakazu Kadoshita. "Tractability and intractability of problems on unit disk graphs parameterized by domain area". In: *Proceedings of the 9th International Symposium on Operations Research and Its Applications* (2010), pp. 120–127.

[IP01] Russell Impagliazzo and Ramamohan Paturi. "On the Complexity of k-SAT". In: *Journal of Computer and System Sciences* Volume 62 (2001), pp. 367–375. ISSN: 0022-0000. DOI: *https://doi.org/10.1006/jcss.2000.1727*.

[IP99] R. Impagliazzo and R. Paturi. “Complexity of k-SAT”. In: *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*. 1999, pp. 237–240. DOI: *10.1109/CCC.1999.766282*.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. “Which Problems Have Strongly Exponential Complexity?” In: *Journal of Computer and System Sciences* Volume 63 (2001), pp. 512–530. ISSN: 0022-0000. DOI: *https://doi.org/10.1006/jcss.2001.1774*.

[IRS16] Madhu Illuri, P. Renjith, and N. Sadagopan. “Complexity of Steiner Tree in Split Graphs - Dichotomy Results”. In: *Algorithms and Discrete Applied Mathematics*. Edited by Sathish Govindarajan and Anil Maheshwari. Cham: Springer International Publishing, 2016, pp. 308–325. ISBN: 978-3-319-29221-2.

[Jai20] Shweta Jain. *Counting Cliques in Real-World Graphs*. UC Santa Cruz, 2020.

[Kap15] Nikola Kapamadzin. “NP Completeness of Hamiltonian Circuits and Paths”. 2015. URL: *http://web.math.ucsb.edu/~padraic/ucsb_2014_15/ccs_problem_solving_w2015/Hamiltonian%20Circuits.pdf*.

[Kar10] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Edited by Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 219–241. ISBN: 978-3-540-68279-0. DOI: *10.1007/978-3-540-68279-0_8*.

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*. Vol. 842. Springer, 1994. ISBN: 3-540-58356-4.

[KLS00] Sanjeev Khanna, Nathan Linial, and Shmuel Safra. “On the Hardness of Approximating the Chromatic Number”. In: *Combinatorica* Volume 20 (Mar. 2000), pp. 393–415. DOI: *10.1007/s004930070013*.

[KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[KT17] Ken-Ichi Kawarabayashi and Mikkel Thorup. “Coloring 3-Colorable Graphs with Less than N1/5 Colors”. In: *J. ACM* Volume 64 (Mar. 2017). ISSN: 0004-5411. DOI: *10.1145/3001582*.

[Kur30] Casimir Kuratowski. “Sur le problème des courbes gauches en Topologie”. fre. In: *Fundamenta Mathematicae* Volume 15 (1930), pp. 271–283.

[Lam21] Michael Lampis. “Exact Algorithms: Lecture 7: The (S)ETH and Implications”. 2021. URL: *https://www.lamsade.dauphine.fr/~kim/lecture/algorithms2021/lecturenotes/[lecture06]ETH.pdf*.

[Lee14] James R. Lee. “CSE 431 Theory of Computation: Lecture 15”. 2014. URL: *https://courses.cs.washington.edu/courses/cse431/14sp/scribes/lec15.pdf*.

[Lic82] David Lichtenstein. “Planar Formulae and Their Uses”. In: *SIAM Journal on Computing* Volume 11 (1982), pp. 329–343. eprint: *https://doi.org/10.1137/0211025*.

[LMS18] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. “Known Algorithms on Graphs of Bounded Treewidth Are Probably Optimal”. In: *ACM Trans. Algorithms* Volume 14 (Apr. 2018). ISSN: 1549-6325. DOI: *10.1145/3170442*.

[Lyu08] Yuh-Dauh Lyuu. “Another variant of 3-Sat”. 2008. URL: *https://www.csie.ntu.edu.tw/~lyuu/complexity/2008a/20080403.pdf*.

[Mar15] Dániel Marx. “Lower Bounds Based on ETH”. 2015. URL: *https://simons.berkeley.edu/sites/default/files/docs/3678/marx-simons3.pdf*.

[Mar20a] Dániel Marx. “Treewidth: Vol. 1”. 2020. URL: *https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/summer20/paraalg/Lectures/lecture_7.pdf*.

[Mar20b] Dániel Marx. “Treewidth: Vol. 2”. 2020. URL: *https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/summer20/paraalg/Lectures/lecture_8.pdf*.

[McC14] Sean McCulloch. “Discussions of NP-Complete Problems: Exact Cover by 3-Sets”. 2014. URL: *https://npcomplete.owu.edu/2014/06/10/exact-cover-by-3-sets/*.

[MM65] J. Moon and L. Moser. “On cliques in graphs”. In: *Israel Journal of Mathematics* Volume 3 (1965), pp. 23–28. ISSN: 0021-2172.

[Mou05] Dave Mount. “CMSC 451: Lecture 21: NP-Completeness: Clique, Vertex Cover, and Dominating Set”. 2005. URL: *https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect21-np-clique-vc-ds.pdf*.

[MSJ19] Silviu Maniu, Pierre Senellart, and Suraj Jog. “An Experimental Study of the Treewidth of Real-World Graph Data”. In: *22nd International Conference on Database Theory (ICDT 2019)*. Edited by Pablo Barcelo and Marco Calautti. Vol. 127. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 12:1–12:18. ISBN: 978-3-95977-101-6. DOI: *10.4230/LIPIcs.ICDT.2019.12*.

[Mül96] Haiko Müller. “Hamiltonian circuits in chordal bipartite graphs”. In: *Discrete Mathematics* Volume 156 (1996), pp. 291–298. ISSN: 0012-365X. DOI: *https://doi.org/10.1016/0012-365X(95)00057-4*.

[Pil17] Michał Pilipczuk. “Parameterized Complexity Summer School: ETH and SETH lower bounds”. 2017. URL: *https://algo2017.ac.tuwien.ac.at/wp-content/uploads/eth.pdf*.

[Pis95] David Pisinger. “A minimal algorithm for the multiple-choice knapsack problem”. In: *European Journal of Operational Research* Volume 83 (1995), pp. 394–410. ISSN: 0377-2217. DOI: *https://doi.org/10.1016/0377-2217(95)00015-I*.

[RS86] Neil Robertson and P.D Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *Journal of Algorithms* Volume 7 (1986), pp. 309–322. ISSN: 0196-6774. DOI: *https://doi.org/10.1016/0196-6774(86)90023-4*.

[Tur37] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* Volume s2-42 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. eprint: *https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf*.

[Wag37] K. Wagner. “Über eine Eigenschaft der ebenen Komplexe”. In: *Mathematische Annalen* Volume 114 (Dec. 1937), pp. 570–590. ISSN: 1432-1807. DOI: *10.1007/BF01594196*.

[You08] Neal E. Young. “Greedy Set-Cover Algorithms”. In: *Encyclopedia of Algorithms*. Edited by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 379–381. ISBN: 978-0-387-30162-4. DOI: *10.1007/978-0-387-30162-4_175*.

[ZLS04] Shaoqiang Zhang, Guojun Li, and Moo-Young Sohn. "Two feedback problems for graphs with bounded tree-width". In: *Applied Mathematics* Volume 19 (June 2004), pp. 149–154. DOI: *10.1007/s11766-004-0048-3*.

[ZP19] Michał Ziobro and Marcin Pilipczuk. "Finding Hamiltonian Cycle in Graphs of Bounded Treewidth: Experimental Evaluation". In: *ACM J. Exp. Algorithmics* Volume 24 (Dec. 2019). ISSN: 1084-6654. DOI: *10.1145/3368631*.

A. Problem definitions

Here, we give the formal definitions of the used problems. We adapt our definitions from [Ber+18], [Cyg+15], [Kar10], [GJ90], [Pis95], [Cer+08] and [ZLS04].

Clique
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $C \subseteq V$ of size k such that each pair of vertices are adjacent.

Clique Cover
Input: Graph $G = (V, E)$ and integer k
Question: Decide if V is the union of k or fewer cliques.

Clique Partition
Input: Graph $G = (V, E)$ and integer k
Question: Decide if V is the disjoint union of k or fewer cliques.

3-Clique Cover
Input: Graph $G = (V, E)$
Question: Decide if V is the union of three or fewer cliques.

k-Clique Cover
Input: Graph $G = (V, E)$
Question: Decide if V is the union of k or fewer cliques.

Colour
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a colouring $c : V \to [k]$ of the vertices of G with k colours such that the endpoints of each edge have different colours.

k-Colour
Input: Graph $G = (V, E)$
Question: Decide if there is a colouring $c : V \to [k]$ of the vertices of G with k colours such that the endpoints of each edge have different colours.

Dominating Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertexset $D \subseteq V$ of size k such that all vertices in $V - D$ are adjacent to at least one vertex in D.

r-Dominating Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertexset $D \subseteq V$ of size k such that all vertices in $V - D$ have at least one vertex of D within distance r.

Connected Dominating Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertexset $D \subseteq V$ of size k such that D induces a connected subgraph and all vertices in $V - D$ are adjacent to at least one vertex in D.

Exact Cover
Input: Universe U and family $\mathcal{A}$ of sets over U
Question: Decide if there is a family $\mathcal{A}' \subseteq \mathcal{A}$ of pairwise disjoint sets such that $\uplus_{F \in \mathcal{F}'} F = U$.

3-Exact Cover
Input: Universe U of size divisible by 3 and family $\mathcal{A}$ of sets of size 3 over U
Question: Decide if there is a family $\mathcal{A}' \subseteq \mathcal{A}$ of pairwise disjoint sets such that $\uplus_{F \in \mathcal{F}'} F = U$.

Feedback Vertex Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $F \subseteq V$ of size k such that $V - F$ induces a forest.

Connected Feedback Vertex Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $F \subseteq V$ of size k such that D induces a connected subgraph and $V - F$ induces a forest.

Hamilton Cycle
Input: Graph $G = (V, E)$
Question: Decide if there is a cycle $C \subseteq E$ that visits all vertices of G.

Hamilton Path
Input: Graph $G = (V, E)$ and two vertices $a, b \in V$
Question: Decide if there is a path $P \subseteq E$ from a to b in G that visits all vertices of G.

Independent Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $I \subseteq V$ of size k that induces no edges.

Weighted Independent Set
Input: Graph $G = (V, E)$ with a weight function $w : V \to \mathbb{R}$ and integer k
Question: Decide if there is a vertex set $I \subseteq V$ of weight $w(I) = k$ that induces no edges.

3D-Matching
Input: Universe U, a set $A \subseteq U \times U \times U$
Question: Decide if there is a set $X \subseteq A$ such that $|X| = |U|$ and no two elements of X agree in any coordinate.

Max Cut
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $X \subseteq V$ such that at least k edges have one endpoint in X and the other in $V - X$.

Weighted Max Cut
Input: Graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}_0$ and integer k
Question: Decide if there is a vertex set $X \subseteq V$ such that the edges that have one endpoint in X and the other in $V - X$ have cumulative weight of k.

Maximum Induced Forest
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $F \subseteq V$ of size k such that F induces a forest.

Sat
Input: CNF formula φ
Question: Decide if there is a satisfying assignment for φ.

q-Sat
Input: Conjunctive normal form (CNF) formula φ, where each clause consist of at most q literals
Question: Decide if there is a satisfying assignment for φ.

3-Sat
Input: Conjunctive normal form (CNF) formula φ, where each clause consist of at most 3 literals
Question: Decide if there is a satisfying assignment for φ.

3,3-Sat
Input: Conjunctive normal form (CNF) formula φ, where each clause consist of at most 3 literals and each variable has exactly three occurrences being exactly once negative.
Question: Decide if there is a satisfying assignment for φ.

Set Cover
Input: Universe U, family $\mathcal{A}$ of sets over U and integer k
Question: Decide if there is a family $\mathcal{A}' \subseteq \mathcal{A}$ of size at most k such that $\bigcup_{A \in \mathcal{A}'} A = U$.

Steiner Tree
Input: Graph $G = (V, E)$ and integer k, as well as a set of terminal vertices $K \subseteq V$
Question: Decide if there is a vertex set $X \subseteq V$ of size at most k such that $K \subseteq X$ and X induces a connected subgraph of G.

Weighted Steiner Tree
Input: Graph $G = (V, E)$ with a weight function $w : V \rightarrow \mathbb{R}_0$ and integer k, as well as a set of terminal vertices $K \subseteq V$
Question: Decide if there is a vertex set $X \subseteq V$ of weight $w(X)$ at most k such that $K \subseteq X$ and X induces a connected subgraph of G.

Vertex Adjacent Feedback Edge Set
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is an edge set $F \subseteq E$ such that $E - F$ induces a forest and F is incident to k vertices.

Vertex Cover
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $S \subseteq V$ of size k such that all edges are incident to at least one vertex from S.

Connected Vertex Cover
Input: Graph $G = (V, E)$ and integer k
Question: Decide if there is a vertex set $S \subseteq V$ of size k such that D induces a connected subgraph and all edges are incident to at least one vertex from S.

Weighted Circuit Satisfiability (WCS)
Input: directed acyclic Graph $G = (V, E)$ with inputs-vertices of indegree 0, negation-vertices of indegree 1, and- as well as or-vertices of indegree greater than 1 and one vertex with outdegree 0 labelled as output-vertex. In addition an integer k (parameter)
Question: Decide if there is a an assignment of input-vertices to 0 or 1 that generates a 1 at the output-vertex an has exactly k input-vertices being assigned a 1.

WCS[t]
Input: directed acyclic Graph $G = (V, E)$ with inputs-vertices of indegree 0, negation-vertices of indegree 1, and- as well as or-vertices of indegree greater than 1 and one vertex with outdegree 0 labelled as output-vertex. G is of constant depth and uses at most t vertices with indegree greater than 2 on each directed path. In addition an integer k (parameter)
Question: Decide if there is a an assignment of input-vertices to 0 or 1 that generates a 1 at the output-vertex an has exactly k input-vertices being assigned a 1