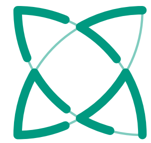
KIT
Karlsruhe Institute of Technology



Engineering Algorithms for CP-Treewidth

Bacheolor's Thesis of

Tobias Gröger

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

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

28.11.2023 – 28.03.2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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

Karlsruhe, 28.03.2024

....

(Tobias Gröger)

Abstract

Dynamic programming is a powerful and much used technique in the design of fixed-parameter tractable (FPT) algorithms for $\mathcal{NP}$-hard problems. However, there is a distinct lack of high performance implementations of these algorithms. Thus, we develop such an implementation on a clique-partitioned tree decompositions to solve the Maximum Independent Set problem with a focus on its optimization and evaluation. First, we improve a well known version of this dynamic program and consider relevant aspects for a practical implementation. We show that, while the cp-treewidth is a better parameter to estimate the runtime of this dynamic program than the treewidth, the clique-partitioning structure is not required, thus our algorithm operates on a regular tree decomposition. In addition, we consider the application of reduction rules and lower and upper bound pruning. Even though pruning seems to be effective in theory, it leads to a higher variation in runtime depending on the choice of root and we were unable to achieve any benefits in practice. However, the reductions rules are highly effective for most instances. We find that only with reduction rules our algorithm outperforms other solvers on instances with a low cp-treewidth.

Zusammenfassung

Die dynamische Programmierung ist eine mächtige und viel verwendete Technik im Entwurf von parametrisierbaren (FPT) Algorithmen für $\mathcal{NP}$-schwere Probleme. Es gibt jedoch einen deutlichen Mangel an hoch performanten Implementierungen dieser Algorithmen. Daher entwickeln wir eine solche Implementierung auf Basis einer Cliquen-partitionierten Baumzerlegungen, um das Problem der Maximalen Unabhängigen Menge, mit Schwerpunkt auf dessen Optimierung und Evaluation, zu lösen. Zunächst verbessern wir eine bekannte Version dieses dynamischen Programms und berücksichtigen relevante Aspekte für eine praktische Implementierung. Wir zeigen, dass die cp-Baumweite ein besseres Maß ist, um die Laufzeit dieses dynamischen Programms abzuschätzen, als die Baumweite. Jedoch ist die Cliquen-partitionierungs Struktur nicht erforderlich, weshalb unser Algorithmus auf einer regulären Baumzerlegung arbeitet. Zusätzlich betrachten wir die Anwendung von Reduktionsregeln, sowie untere und obere Schranken. Obwohl die Beschränkung mit Hilfe oberer und unterer Schranken in der Theorie effektiv zu sein scheint, führt sie zu einer höheren Variabilität in der Laufzeit für unterschiedliche Wahlen der Wurzel und wir konnten keine Vorteile in der Praxis erzielen. Die Reduktionsregeln sind hingegen für die meisten Instanzen äußerst effektiv. Wir stellen fest, dass unser Algorithmus nur mit Hilfe der Reduktionsregeln schneller als andere Löser auf Instanzen mit niedriger Baumweite ist.

Contents

1 Introduction **1**
1.1 Related Work . . . 2

2 Preliminaries **5**
2.1 Notation . . . 5
2.1.1 Graph Notation . . . 5
2.2 Tree Decomposition . . . 5
2.3 Maximum Independent Set . . . 6
2.4 Clique Partitioned Tree Decomposition . . . 7
2.5 Experiment Setup . . . 8

3 Algorithm Engineering **9**
3.1 Generative Algorithm . . . 9
3.2 Positive Instance Driven . . . 9
3.3 Data structures . . . 11
3.4 Comparison with State of the Art . . . 14

4 Algorithmic Optimizations **17**
4.1 Reduction Rules . . . 17
4.1.1 Evaluation . . . 19
4.2 Upper/Lower Bound Pruning . . . 20
4.2.1 Pruning . . . 21
4.2.2 Specific Bounds . . . 23
4.2.3 Bounds Evaluation . . . 26
4.3 Choice of Root . . . 28

5 Evaluation **37**
5.1 Runtime Evaluation . . . 37
5.2 Graph Parameter Evaluation . . . 37

6 Conclusion **41**
6.1 Future Work . . . 41

Bibliography **43**

1 Introduction

In the 1950s Bellman [Bel54] presented a novel strategy for developing algorithms in a broad range of computational problems, that he called dynamic programming. In loose terms, it exploits the structure of a problem by recursively breaking it down into smaller and simpler sub-problems. The solutions to these sub-problems can then be used to reconstruct a solution for the whole problem. A problem which exhibits this property, that optimal solutions can be constructed from optimal solutions of their sub-problems, is said to have an *optimal substructure* [CLRS09].

In the realm of graph problems, graph decomposition structures, like the tree decomposition, can be utilized to obtain this property for many different problems. This can be exploited to develop fixed-parameter tractable (FPT) algorithms for $\mathcal{NP}$-complete problems. That is, algorithms which have a polynomial complexity in the input size, but an exponential one in some parameter k. As a result, for an input with bounded parameter, we can solve $\mathcal{NP}$-hard problems in polynomial time.

One decomposition structure often used for these problems is the tree decomposition with its respective parameter, the treewidth. It was first proposed by Robertson and Seymour [RS86] in 1986. This parameter is a measure of how tree-like a given graph is. A tree decomposition structures the graph into bags, which are subsets of vertices, who themselves form a tree. Then, the width of this tree decomposition is the size of the largest bag with the treewidth being the smallest width of any tree decomposition. Each bag acts as a separator in the graph, thus allowing a DP to exploit the tree structure to efficiently solve problems. As it turns out, this parameter has indeed been used to find many innovative algorithms [AN02 | Cyg+15 | BCKN15]. Among other problems, the Maximum Independent Set problem can also be solved using this framework [BBL13]. However, the majority of these algorithms have been studied primarily in their theoretical aspects. As a result, to our knowledge, there are no existing high performance solvers for this problem using dynamic programming on tree decompositions. We aim to rectify this in this thesis, by implementing such a dynamic program and evaluating whether this approach is competitive with other state of the art solvers.

One of the major motivations in the development of these algorithms is that the chosen parameter is low for the anticipated input instances. However, a notable cause for a high treewidth is the presence of large cliques in the graph, as the size of a clique is a lower bound for the treewidth. Unfortunately, large cliques are quite common in many real-world networks which results in a rather high treewidth for many of these graphs [MSJ19]. As a result, since its discovery in 1986, many alternative and improved parameters, which attempt to address this problem, have been proposed [BKW23 | Aro21 | dBer+18]. This has lead to the proposal of the clique-partitioned treewidth by Bläsius, Katzmann, and Wilhelm [BKW23] in 2023. This parameter further utilizes the structure of each bag, partitioning it into cliques. Then, the size of each clique only has a logarithmic impact towards the global width. They have shown empirically that this does reduce the cp-treewidth in comparison to the regular treewidth on many real-world networks. Additionally, they have already shown in their paper how this

parameter can be used to solve the Maximum Independent Set problem. However, their work mainly focuses on the computation of this parameter and the theoretical aspects of its usage. Again, not much about its practical applications are known.

Therefore, the aim of this thesis is to use this parameter to develop a fast and efficient dynamic program solving the Maximum Independent Set problem and comparing it to a state of the art branch and bound solver. First, we present the general concepts and the used notation in Chapter 2. Then, we discuss the details related to a high performance implementation in Chapter 3 and present a much improved version of the original dynamic program. We find that the regular treewidth suffices for this improved version, and it doesn't require the clique partitioning of the cp-treewidth. Then, we compare this algorithm with the state of the art solver, discovering there is still much need for improvement to become competitive. Thus, we further optimize our algorithm by incorporate strategies similar to the ones used in the branch and bound solver into our own algorithm in Chapter 4. These are reduction rules in Section 4.1 and the pruning of some solutions based on upper and lower bounds in Section 4.2. In addition, we evaluate whether the choice of root of the tree decomposition has any effect on the runtime of our algorithm in Section 4.3. Based on these techniques we find that our algorithm is competitive on graphs with a low cp-treewidth. Finally, in Chapter 5, we do a last evaluation of our algorithm and analyse the runtime of our algorithm in regards to several graph parameters. We find, that the cp-treewidth predicts the running time of the algorithm better than the regular treewidth. Lastly, we end this thesis with some final conclusions and some ideas for further research in this area in Chapter 6.

1.1 Related Work

The treewidth of graphs and its application to parameterized algorithms has been extensively studied. Different variations of this parameter have been proposed with the intention of proving a subexponential complexity for various graph problems on specific graph families. One of these was proposed by de Berg et al. [dBer+18] which partitions the graph into cliques and contracts every clique into a single weighted vertex. For a clique of size s, the vertex gets assigned a weight of $\log(s+1)$. A weighted tree decomposition, where the weight of each bag is the sum of the weights of its vertices, is then calculated on the contracted graph. This parameter yields a subexponential complexity for a multitude of problems on geometric intersection graphs, one of which is the independent set problem.

A different, but similar parameter, the *tree-clique width*, was introduced by Aronis [Aro21]. They use a different technique which augments a tree decomposition with an edge clique cover for every bag of the decomposition. The width-weight of a bag is the size of the edge clique cover and the width-weight of the augmented tree decomposition is the maximum width-weight of any bag. The tree-clique width is then defined as the minimum possible width-weight of any augmented tree decomposition. The paper shows several hardness results for this parameter.

Another related parameter was proposed by Dallard, Milanič, and Štorgel [DMŠ23]. They introduce the *independence number* of a tree decomposition as the maximum independence number of any of its bags. The *tree-independence number* of a graph is then defined as the minimum possible independence number of any of its tree decompositions. This paper uses a more theoretical approach to prove a polynomial complexity for the Max Weight Independent Packing problem on graphs given that they have a bounded tree-independence number.

A recent variation, also based on the partitioning of bags into cliques, is the *clique-partitioned treewidth* presented by Bläsius, Katzmann, and Wilhelm [BKW23]. It is calculated by first producing a tree decomposition and then partitioning every bag into a clique cover, in contrast to the parameter of de Berg et al., which calculates a clique partition of the graph, before calculating the tree decomposition. They present different heuristic algorithms to compute an upper bound on this parameter. Additionally, they show empirically that this parameter is similar to the regular treewidth on most considered graphs, but there are some instances, especially graphs with a high clustering coefficient, where the cp-treewidth is significantly lower (sometimes by a factor of over 10).

This parameter will be used to implement a variation of the well known dynamic program for solving the Maximum Independent Set problem, like presented in *Parameterized Algorithms* [Cyg+15]. The worst-case complexity of this dynamic program has been shown to be linear in the graph size and exponential in the treewidth [Bod88 | AP84]. Bodlaender, Bonsma, and Lokshtanov [BBL13] presented an algorithm to solve this problem with a runtime of $\mathcal{O}(2^k n)$ using such a dynamic program and discuss relevant aspects for a practical implementation. Tree decompositions can be efficiently computed by the htd framework, which was implemented by Abseher, Musliu, and Woltran [AMW17]. Additionally, this framework has been used by Fichte, Hecher, Morak, and Woltran [FHMW21] to implement an efficient solver for answer set programming (ASP), using dynamic programming on a tree decomposition. They empirically show that their algorithm is competitive with other state of the art solvers for instances of low treewidth. However, to our knowledge no actual high performance implementation of a dynamic program to solve the Maximum Independent Set problem has been attempted. Current state of the art solvers for this problem usually implement a branch and bound strategy supported by various data reduction rules. These reduction rules have been studied by Akiba and Iwata [AI16]. Both the first [HLSS19] and second [Tri19] place of the PACE 2019 vertex cover challenge [DFH19] use these reduction rules. The winning algorithm *We Got You Covered* was implemented by Hespe, Lamm, Schulz, and Strash [HLSS19] and consists of a hybrid architecture using these reductions in its first phase.

2 Preliminaries

In this chapter, we introduce the basic concepts required to understand this thesis. We present the mathematical notation used and introduce the tree decomposition of a graph, a fundamental concept utilized in this thesis. Then, we show some important properties of tree decompositions and present the main focus of this thesis, the Maximum Independent Set problem and a well-known dynamic program for solving it. In the end, we describe the clique-partitioned tree decomposition, a variation of the regular tree decomposition.

2.1 Notation

In this section, we present the general mathematical notation and abbreviations used in this thesis. We also introduce the notation used for graphs.

For two disjoint sets A and B we write the disjoint union as $A \dot{\cup} B$.

2.1.1 Graph Notation

Throughout this thesis, we assume a graph $G = (V, E)$ is simple and undirected unless stated otherwise. For the set of vertices of a graph G we write $V(G)$ and for the set of edges we write $E(G)$. For a subset $S \subseteq V$ we write $G[S]$ as the subgraph of G induced by S. For the subgraph $G[V(G) - S]$ we write $G - S$. For the neighbourhood of a vertex v we write $N(v) := \{u | \{u, v\} \in E(G)\}$, and for the degree we write $deg(v) := |N(v)|$. The closed neighbourhood we define as $N[v] := N(v) \cup \{v\}$. For a vertex subset $S \subseteq V$ we write $N(S) := \bigcup_{v \in S} N(v) \setminus S$ and $N[S] := N(S) \cup S$. The k-neighbourhood of a vertex v, that is the set of vertices at distance k from v, is denoted by $N^d(v)$.

2.2 Tree Decomposition

Given a graph G, a *tree decomposition* is a pair (T, B), where T is a tree and B a function $B : V(T) \to 2^{V(G)}$. B maps tree nodes to subsets of graph vertices, such that T and B satisfy the following three properties: (1) every vertex of the graph is part of at least one tree node: $\bigcap_{X \in V(T)} B(X) = V(G)$, (2) for every edge of the graph, there exists at least one tree node which contains both adjacent nodes: $\forall \{u, v\} \in E(G) : \exists X \in V(T) : \{u, v\} \subseteq B(X)$, (3) for every graph vertex v, the set of tree nodes $X \in V(T)$, such that $B(X)$ contains v, that is $\{X \in V(T) : v \in B(X)\}$, form a connected subtree of T.

To distinguish between graph vertices and tree vertices we utilize the naming scheme from Cygan et al. [Cyg+15] and refer to tree vertices as nodes. For a tree node $X \in V(T)$ we call the set of associated graph vertices $B(X)$ a *bag*. The width of a tree decomposition is the size of the largest bag minus 1. The *treewidth*, $\mathrm{tw}(G)$, of a graph G is the smallest width of any valid tree decomposition. A rooted tree decomposition is a tuple (T, B, r), where (T, B) is a regular tree decomposition and $r \in V(T)$ is a valid root of T. For a rooted tree decomposition

(T, B, r) and a tree node $X \in V(T)$ we refer to all nodes in the subtree of T rooted at X and including X as T_X. We also expand the definition of B to subsets of tree nodes. That is, we define $B(S)$ where $S \subseteq V(T)$ as $B(S) := \bigcup_{Y \in S} B(Y)$.

We now consider an important property of tree decompositions that is essential for understanding the dynamic program described later on in this chapter.

Theorem 2.1: *Given a graph G and a rooted tree decomposition of that graph (T, B, r). The bag $B(X)$ of every tree node $X \in V(T)$ separates the graph G into $G[B(T_X) \setminus B(X)]$ and $G - B(T_X)$.*

For a tree decomposition (T, B) we can introduce the concept of a *nice* tree decomposition. A tree decomposition is nice, if it is rooted and satisfies the following conditions:

1. $B(r) = \emptyset$ and for every leaf l, $B(l) = \emptyset$. That is, the bag of the root and every leaf is empty.
2. All other nodes of T are of the following three types
   - **Introduce node**: a node X with a single child Y such that $B(X) = B(Y) \dot\cup \{u\}$. In other words, to the bag of Y a single vertex u is introduced to get to the bag of X.
   - **Forget node**: a node X with a single child Y such that $B(Y) = B(X) \dot\cup \{v\}$. In other words, from the bag of Y a single vertex v is forgotten to get to the bag of X.
   - **Join node**: a node X with two children Y_1 and Y_2 such that $B(X) = B(Y_1) = B(Y_2)$. In other words, X joins two children with the same bag.

The existence and size of nice tree decomposition has been proven by Cygan et al. [Cyg+15] according to the following theorem.

Theorem 2.2 (Lemma 7.4 [Cyg+15]): *For a graph G with a tree decomposition (T, B) of width at most k, there exists a nice tree decomposition of width at most k with at most $\mathcal{O}(k|V(G)|)$ vertices which can be constructed in time $\mathcal{O}(k^2 \max\{V(G), V(T)\})$.*

2.3 Maximum Independent Set

Given a graph G, a set of vertices $S \subseteq V(G)$ is called *independent*, if no two vertices of S are adjacent. A maximum independent set for a graph G is an independent set of largest possible size for that graph. The size of that set is called the *independence number* of a graph.

The Maximum Independent Set problem for a graph G can be solved as a dynamic program on the the tree decomposition of the graph. The complete explanation and proof can be found in *Parameterized Algorithms* [Cyg+15], but is described here briefly for completeness.

Given a tree decomposition (T, B) for a graph G, we formulate a subproblem for every node $X \in V(T)$ and every possible subset $S \in 2^{B(X)}$. Let $\hat{S}$ be a maximum independent set of $G[B(T_X)]$ such that $\hat{S} \cap B(X) = S$. We then define the solution of the subproblem as

$$c[X, S] = |\hat{S}| \,. \tag{2.1}$$

If such a set does not exists, e.g., S itself is not independent, we set the value to $-\infty$ and call the solution invalid, otherwise we call a solution valid. By applying Theorem 2.2 we can assume we have a nice tree decomposition (T, B, r) rooted at some node r. As such, the bag of the root vertex is empty and, assuming all subproblems have been solved, the size of the maximum independent set for the whole graph is $c[r, \emptyset]$ as $G[B(T_r)] = G$. To calculate the

solutions of the subproblems we perform bottom-up traversal of the tree decomposition. As the tree decomposition is nice, there are only four cases for every node X. Let S be any subset of $B(X)$. If S is not independent, we can set $c[X, S] = -\infty$, thus in the following we assume S to be independent.

1 **Leaf node**: If X is a leaf node, $B(X) = \emptyset$ and $G[B(T_X)]$ is the empty graph. $c[X, \emptyset] = 0$.

2 **Introduce node**: If X is an introduce node with child Y such that $B(X) = B(Y) \dot{\cup} \{u\}$:

$$c[X, S] = \begin{cases} c[Y, S] & \text{if } u \notin S \\ c[Y, S \setminus \{u\}] + 1 & \text{if } u \in S \end{cases}$$

3 **Forget node**: If X is a forget node with child Y such that $B(X) = B(Y) \setminus \{v\}$:

$$c[X, S] = \max\{c[Y, S], c[Y, S \cup \{v\}]\}$$

4 **Join node**: If X is a join node with children Y_1 and Y_2 such that $B(X) = B(Y_1) = B(Y_2)$:

$$c[X, S] = c[Y_1, S] + c[Y_2, S] - |S|$$

For the above calculations, we assume the rule $-\infty + \lambda = -\infty$ for $\lambda \in [-\infty, \infty)$. Bottom up traversal of the tree decomposition ensures correct order of calculation for the subproblems, as we only depend on the solutions of the child nodes. Thus, the size of the maximum independent set can be found in $c[r, \emptyset]$.

There are multiple methods of reconstructing the actual maximum independent set from the dynamic program. One possible method is traversing the tree in reverse order of calculating the solution and keeping track of the subset S which resulted in the maximum solution. Every time a vertex which was included in this subset is forgotten, it is part of the maximum independent set.

2.4 Clique Partitioned Tree Decomposition

A variation of a tree decomposition introduced by Bläsius, Katzmann, and Wilhelm [BKW23] is called the *clique-partitioned tree decomposition*. In addition to a regular tree decomposition, for every bag, it partitions the subgraph induced by that bag into cliques. That is, for a graph G, a clique-partitioned tree decomposition is a tree decomposition (T, B) where for every tree node $X \in V(T)$ we have a partition $\mathcal{P}_X$ of $G[B(X)]$ into cliques. They defined the weight of a clique C as $\log(|C| + 1)$ and the weight of a bag $B(X)$ as the sum of weights of the cliques in its partition $\mathcal{P}_X$. The weight of a clique-partitioned tree decomposition is the maximum weight of any of its bags. Similarly, the clique-partitioned treewidth (short: cp-treewidth), $\operatorname{cptw}(G)$, of a graph G is the minimum weight of any clique-partitioned tree decomposition.

As shown in their paper, this parameter can be used to bound the number of valid solutions for each tree node of the dynamic program for maximum independent sets presented in Section 2.3 according to the following lemma.

Lemma 2.3 (Lemma 2 [BKW23]): *Given a graph G with a tree decomposition (T, B), the number of valid solutions for each tree node $X \in V(T)$ is smaller than $2^{\operatorname{cptw}(G)}$.*

2.5 Experiment Setup

In this thesis, we will evaluate and analyse different algorithms for the Maximum Independent Set problem. For this purpose, we require multiple data sets with graphs of different structure and characteristics. Bläsius and Fischbeck [BF22] analysed and presented a list of 2751 real-world networks which are well suited for this purpose. This list will be used as our input whenever we use real-world networks (*rwg*). However, we limit the instances to those with a cp-treewidth of less than 25, as otherwise our algorithm would not finish within the time limit.

Additionally, we also use a set of geometric inhomogeneous random graphs (*girg*), a model for random graphs designed to resemble real-world networks. These can be efficiently generated using a library implemented by Bläsius et al. [Blä+19]. There are multiple parameters to affect the structure of the generated graph. The clustering can be influenced by the α parameter, while the underlying geometry is determined by the power-law exponent. We use a different number of vertices and values for the α and power-law exponent parameters depending on the required application. We always choose 10 as the average degree.

Small girgs. Whenever we require small instances for computationally complex experiments we use this data set. It contains graphs generated with 100 and 200 vertices, α parameters of 1.3, 1.6 and ∞. As the power-law exponent we choose 2.1 and 8.

Girgs. For all other experiments we use this data set which contains larger instances. It contains graphs generated with 1000 and 2000 vertices, α parameters of 1.1, 1.3, 1.5, 1.8, 2, 10 and ∞. As the power-law exponent we choose 2.1, 4 and 8. We choose disproportional many instances with a small α parameter, as instances with a larger α value are reduced fully with the reduction rules proposed later on in Section 4.1. To be able to evaluate these reductions properly, we generate more instances in this range.

We executed our experiments on different machines. However, all experiments are performed deterministically, with every component which contains some randomness always executed with the same seed. As such, experiments which do not measure execution time result in the same output independent of the machine they were executed on. Thus, we ran every experiment which does measure execution time on the same machine to guarantee comparability. This machine is a Transtec PHI 4230 Workstation with two Intel(R) Xeon(R) CPU E5-2680 CPUs at 2.70 to 3.50 GHz and 256GB of DDR3 1333 MHz ECC main memory. It is running Ubuntu 22.04.4 LTS. To measure the execution time, we ran the algorithm multiple times, usually two to fives times, depending on the total runtime, on the same input. Then, we take the average as our final measurement. Generally, we only measure the execution time of our algorithm which already expects a nice tree decomposition together with the graph input. As such, we don't take the time it takes to calculate such a tree decomposition into account To limit the completion time of our experiments, we use a time limit for running each algorithm. That is, the We Got You Covered algorithm [HLSS19], the calculation of the tree decomposition and our dynamic program, each get a time limit of 20 minutes.

All our algorithms are implemented in C++ and compiled with gcc 12.3.0.

3 Algorithm Engineering

In this chapter, we present different algorithms which realise the dynamic program presented in Section 2.3. First, we discuss the general algorithmic idea behind the algorithm and how it can be improved. Additionally, we highlight the most relevant data structures from a performance perspective and show the possible implementation options. Furthermore, we evaluate these approaches from a runtime and memory consumption perspective. Lastly, we compare this improved algorithm to a current state of the art solver for Maximum Independent Set.

3.1 Generative Algorithm

In Section 2.3 we present a dynamic program to calculate the maximum independent set of a graph G. The DP table stores the subproblem solutions described in Equation (2.1). The table is filled by bottom up traversal of the tree decomposition. For every tree node X, every possible subset $S \subseteq B(X)$ is generated. Then, S is checked for its independence. If S is independent, the solution $c[X, S]$ is calculated according to the formulas described in Section 2.3 and stored in the DP table. Otherwise, $-\infty$ is stored as the result.

We call this algorithm the *generative algorithm*.

One immediate observation from this algorithm is that invalid solutions never lead to another valid solution. As such, invalid solutions do not need to be stored or calculated. As a result, instead of generating all possible subsets, it is sufficient to generate only independent subsets, as subsets which are not independent result in an invalid solution anyway. This way, a more efficient method of generating subsets can be implemented. When a clique-partitioned tree decomposition is known, the partition $\mathcal{P}_X$ of X into cliques can be utilized to even further optimize the generation of independent subsets. An independent set can only intersect any clique in one vertex. That means, only one vertex from any clique in $\mathcal{P}_X$ has to be chosen for any independent subset.

3.2 Positive Instance Driven

The algorithm described in the previous section generates all possible subsets (or all independent subsets) for every bag anew. Instead of iterating over all these possible subsets, we can take advantage of the work we have already done in the children and construct all subsets which lead to a valid solution from the existing solutions of the children. We will call this algorithmic approach the *positive instance driven algorithm*, as we only keep valid solutions and use them to generate further valid solutions. This observation has been formalised in the following algorithm: Start with the generative algorithm and follow the same order of traversal. When encountering a leaf l continue with the same algorithm, but when encountering any other node type do the following instead:

Introduce node. Let X be an introduce node with child Y such that $B(X) = B(Y) \dot{\cup} \{u\}$. For every set S such that $c[Y, S]$ is a valid solution, set $c[X, S] = c[Y, S]$ and if $S \dot{\cup} \{u\}$ is independent, set $c[X, S \cup \{u\}] = c[Y, S] + 1$.

In other words, iterate over all valid solutions of the child and copy the corresponding solution to the current node. Additionally, add another solution where the subset includes the introduced node if that subset stays independent.

Forget node. Let X be a forget node with child Y such that $B(X) = B(Y) \setminus \{v\}$. For every set S such that $c[Y, S]$ is a valid solution, let $\tilde{S} := S$ if $v \notin S$ otherwise let $\tilde{S} := S \setminus \{v\}$, then set $c[X, \tilde{S}] = \max\{c[X, \tilde{S}], c[Y, S]\}$. Where $c[X, \tilde{S}] = -\infty$ if such a value is not present in c.

In other words, iterate over all valid solutions of the child. Then, copy that solution to the current node if it does not already contain that solution or the current solution is smaller.

Join node. Let X be a join node with children Y_1 and Y_2 such that such that $B(X) = B(Y_1) = B(Y_2)$. For every set S such that $c[Y_1, S]$ and $c[Y_2, S]$ are valid solutions, set $c[X, S] = c[Y_1, S] + c[Y_2, S] - |S|$.

In other words, copy all valid solutions which are present in both children. This can be achieved by parallel iteration of all solutions of both children in sorted order similar to merge sort.

Theorem 3.1: *The above algorithm is correct. That is, it will yield the correct maximum independent set size and allow for the same reconstruction of the actual independent set described in Section 2.3.*

Proof. To prove this theorem, we compare the above algorithm to the dynamic program described in Section 2.3, which we call the reference. Let a be the DP table containing the subproblem solutions according to Equation (2.1) of the reference and c be the DP table calculated by the above algorithm.

We propose that for every node $X \in V(T)$ and set S such that $a[X, S] > -\infty$, that is $a[X, S]$ is a valid solution, $c[X, S] = a[X, S]$. We give a proof by induction on the traversal of the tree nodes. The algorithm uses the same procedure for leaf nodes, so the proposition holds for every leaf. Assume the proposition holds for every tree node already encountered. We will now show that the proposition also holds for the next node X encountered during the traversal: (1) X is an introduce node with the child Y where the vertex u is introduced. Let S be an arbitrary subset of $B(X)$ such that $a[X, S] > -\infty$. It follows that S is independent. If $u \notin S$ then $a[Y, S] > -\infty$ and as such $c[X, S] = c[Y, S]$ is properly calculated. If $u \in S$ then $a[Y, S \setminus \{u\}] > -\infty$ and as such $c[X, S] = c[Y, S \setminus \{u\}] + 1$ is properly calculated. (2) X is a forget node with the child Y where the vertex v is forgotten. Let S be an arbitrary subset of $B(X)$ such that $a[X, S] > -\infty$. If $a[Y, S] > a[Y, S \dot{\cup} \{v\}]$, then $c[X, S] = c[Y, S]$ is calculated properly when S is encountered as a valid solution of Y. Otherwise $c[X, S] = c[Y, S \dot{\cup} \{v\}]$ is properly calculated when $S \dot{\cup} \{v\}$ is encountered as a valid solution of Y. The correct solution is not overwritten in case the larger solution is encountered later on in the algorithm, as the algorithm checks for an existing larger value before writing $c[X, S]$. (3) X is a join node with children Y_1 and Y_2. Let S be an arbitrary subset of $B(X)$ such that $a[X, S] > -\infty$. It follows that $a[Y_1, S] > -\infty$ and $a[Y_2, S] > -\infty$ and as such $c[X, S] = c[Y_1, S] + c[Y_2, S] - |S|$ is calculated properly. This concludes the proof that the proposition holds for X and consequently holds for all tree nodes.

As a result the value in $c[r, \emptyset] = a[r, \emptyset]$, so the maximum independent set size is correct. Similarly, the value of all valid solutions of subproblems also equals that of the reference, hence the independent set reconstruction works analogous. ∎

Using this algorithm, we do not have to check every possible subset for each bag, but can rather build upon the solutions already calculated. This should save us many lookup operations in the processing of each node. Thus, we expect this algorithm to significantly outperform the generative algorithm. In addition, we note that this algorithm does not take advantage of the clique partitioning of the clique-partitioned treewidth in any form. As such, our algorithm operates on a regular tree decomposition.

3.3 Data structures

One of the most important considerations for a practical implementation of the described algorithms is the data structure used for storing the DP table, that is, the values from Equation (2.1). The basic part of such a data structure is the mapping from tree node and subset of graph vertices to solution values.

As already stated in Section 3.1 we do not need to store invalid solutions. As such, a trivial space optimization is to store invalid solutions in the mapping, i.e. when a certain subset does not have a stored mapping, the solution is assumed to be invalid. As a result the solution can be stored as a simple number.

The mapping from tree nodes can be easily fulfilled by a contiguous array. The only prerequisite is a mapping of tree nodes to a continuous range which is always met in our case. As there are always values for every tree node this approach does not waste any space and achieves a high performance.

The data structure for the subsets needs to support easy comparison so it can be used as a key in a mapping. Additionally, it should be space efficient. A basic approach would be to store the included vertices in a sorted array which would allow for easy comparison but requires space for a full number for each member member of the subset. However, we can observe that all relevant subsets are only subsets of a single bag and not the whole graph. As such, we can store a subset of a sorted bag $B(X)$ as a bit vector of size $|B(X)|$ where each entry in the bit vector indicates whether the vertex at that index is part of the subset. This results in a much tighter packing than the other approach. However, there is a drawback when comparing subsets of different bags, as they cannot be compared directly. This means a mapping between subset and bag is required before comparison is possible.

For the mapping from subsets to solution values there are different considerations. The generative algorithm (Section 3.1) and positive instance driven algorithm (Section 3.2) require a different number of operations for each node type, as shown in Table 3.1. In short, the generative algorithm requires numerous lookups, but does not require any form of iteration support. The positive instance driven algorithm requires far less lookups but does require sorted iteration support. This leads to a few choices for a specific data structure. An obvious choice is either an ordered or an unordered map where the unordered map usually has an advantage in insertion and lookup times but does not support ordered iteration. This means the unordered map is not suitable for the positive instance driven algorithm. An alternative is a simple array which can be either sorted to support quick lookup times or be left unsorted. If left unsorted, it has to be sorted first, when sorted iteration is required (join node) and when lookup is required some form of helper data structure is required. That is, a tree-like lookup data structure has to be computed first.

Table 3.1: Number of operations needed on the mapping saving the solutions for a tree node X depending on its type and the algorithm. The value k represents the number of independent subsets of the bag $B(X)$, $k \in \mathcal{O}(2^{|B(X)|})$. The value m represents the maximum number of solutions in any of the children. The iteration row indicates whether no iteration (-), iteration in arbitrary order (y) or ordered iteration (o) is required.

Operation	Generative (Section 3.1)				Positive Instance Driven (Section 3.2)			
	Leaf	Introduce	Forget	Join	Leaf	Introduce	Forget	Join
Insert	1	$< k$	$< k$	$< k$	1	$< 2m$	$< m$	$< m$
Lookup	0	k	$2k$	$< 2k$	0	0	m	0
Iteration	-	-	-	-	-	y	y	o

For a further analysis both the generative as well as the positive instance driven algorithm have been implemented with different data structures to analyse the runtime requirements in different scenarios on different input graphs. For this purpose, we examine the runtime for a node X in respect to bag size for each different data structure and node type. Let k be the number of independent subsets of $B(X)$ and let m be the maximum number of solutions in any of the children of X.

Generative algorithm. The generative algorithm has been implemented with both an ordered map (`std::map`, *genSMap*) as well as an unordered map (`std::unordered_map`, *genSHashMap*). The ordered map supports a lookup and insertion time of $\mathcal{O}(\log(\text{size}))$, while the unordered map supports an average lookup and insertion time of $\mathcal{O}(1)$. Each node type requires at most k insertions. An introduce node requires k lookups, a forget node requires $2k$ lookups and a join node requires at most $2k$ lookups. Each lookup is performed in the solutions of a child node.

As the unordered map has both faster lookup and faster insertion time on average, the unordered map should theoretically outperform the ordered map. This is also reflected in the experimental results, as shown in Figure 3.1, where the unordered map outperforms the ordered one consistently. The worse worst-case runtime of the unordered map does not hurt the performance in our case and thus the unordered map is the better choice for our application.

Positive Instance Driven. The positive instance driven algorithm has been implemented with an ordered map (`std::map`, *itCMap*), an unsorted array (`std::vector`, *itCVecMap*, *itCVec HashMap*) and a sorted array (`std::vector`, *itCVecSorted*). Each node requires iteration over the child solutions while the join node additionally requires ordered iteration.

As stated above, the ordered map has logarithmic lookup and insertion time. When using the ordered map, an introduce node requires at least m and at most $2m$ insertions. A forget node requires m lookups and at most m insertions. A join node requires at most m insertions. Additionally, each iteration takes place over the underlying tree data structure of the map and is therefore not very efficient.

Taking a look at the unsorted array, the insertions are constant time. The introduce node requires the same operations as the ordered map and thus, the runtime for an introduce node is faster than the map. On the other hand, when a forget node is encountered a helper map is needed to facilitate lookups, otherwise they would result in linear runtime. As lookups

Table 3.2: Runtime for each node type depending on which data structure is used for a tree node X. The value k represents the number of independent subsets of the bag $B(X)$, $k \in \mathcal{O}(2^{|B(X)|})$. The value m represents the maximum number of solutions in any of the children of X. The runtime is shown as the worst-case number of operations required times the runtime of this operation in big-$\mathcal{O}$ notation. When a node requires different types of operations, they are all listed separated by a comma.

Data structure	Introduce	Forget	Join
genSMap	$2k \cdot \mathcal{O}(\log(m))$	$3k \cdot \mathcal{O}(\log(m))$	$3k \cdot \mathcal{O}(\log(m))$
genSHashMap	$2k \cdot \mathcal{O}(1)$	$3k \cdot \mathcal{O}(1)$	$3k \cdot \mathcal{O}(1)$
itCMap	$2m \cdot \mathcal{O}(\log(m))$	$2m \cdot \mathcal{O}(\log(m))$	$m \cdot \mathcal{O}(\log(m))$
itCVecMap	$2m \cdot \mathcal{O}(1)$	$2m \cdot \mathcal{O}(\log(m)), \mathcal{O}(m)$	$2 \cdot \mathcal{O}(m\log(m)), m \cdot \mathcal{O}(1)$
itCVecHashMap	$2m \cdot \mathcal{O}(1)$	$2m \cdot \mathcal{O}(1), \mathcal{O}(m)$	$2 \cdot \mathcal{O}(m\log(m)), m \cdot \mathcal{O}(1)$
itCVecSorted	$2m \cdot \mathcal{O}(1), \mathcal{O}(m)$	$m \cdot \mathcal{O}(1), m \cdot \mathcal{O}(\log(m))$	$m \cdot \mathcal{O}(1)$

are only required in the newly inserted elements, they are instead first inserted into either an ordered map or an unordered map and only when the node has been completed, they are moved to the actual array. As a result, a forget node requires m lookups and at most m insertions in the helper map. In addition, the solutions have to be copied from the helper map to the solution array. As stated above, the inferior worst-case runtime of the unordered map is irrelevant for us and thus, the unordered map outperforms the ordered one. A join node requires ordered iteration and thus the array has to be sorted first. Then, at most m insertions are required. In contrast to the map, the iteration takes place over the contiguous array which results in much higher performance.

When we always keep the array sorted, we do not need to explicitly sort it for a join node. On the other hand, lookups and arbitrary insertions require logarithmic time. In an introduce node, we iterate over the sorted solutions of the child and insert this solution and an additional one, if the introduced node can be included while staying independent. This results in two sorted ranges, as the new solutions themselves are also sorted, which can then be merged in linear time. This means, an introduce node requires merging of two sorted arrays with length at most m. For a forget node, the insertions of new solutions are always in sorted order and can thus be achieved in linear time. Additionally, the required lookups run in logarithmic time. Thus, a forget node requires at most m constant time insertions and at most m lookups which run in $\mathcal{O}(\log(m))$. A join node only requires a single ordered iteration over the solutions of both children. This means a join node requires at most m constant time insertions. This method also profits from very fast iteration, just the same as the unordered array.

As a summary, the number of operations and their runtime are shown in Table 3.2. We find, for the positive instance driven algorithm, the sorted array has the fastest theoretical runtime for the join node and a very similar runtime to the next fastest for the forget node. On the other hand, the unsorted array has a faster runtime for the introduce node. Additionally, theoretical runtime comparisons between the generative algorithm and the positive instance driven algorithm are quite hard, as they use a fundamentally different approach. Thus, for further analysis and comparison we use experimental results.

For the experimental results, we run the algorithm for each proposed data structure. Then we measure the average time spent processing each node type and the total runtime. For the input we use the *girgs* dataset described in Section 2.5 with the difference that the α parameter was set to 2, 10 and ∞, as we don't use reductions in this experiment and the original values were chosen with them in mind.

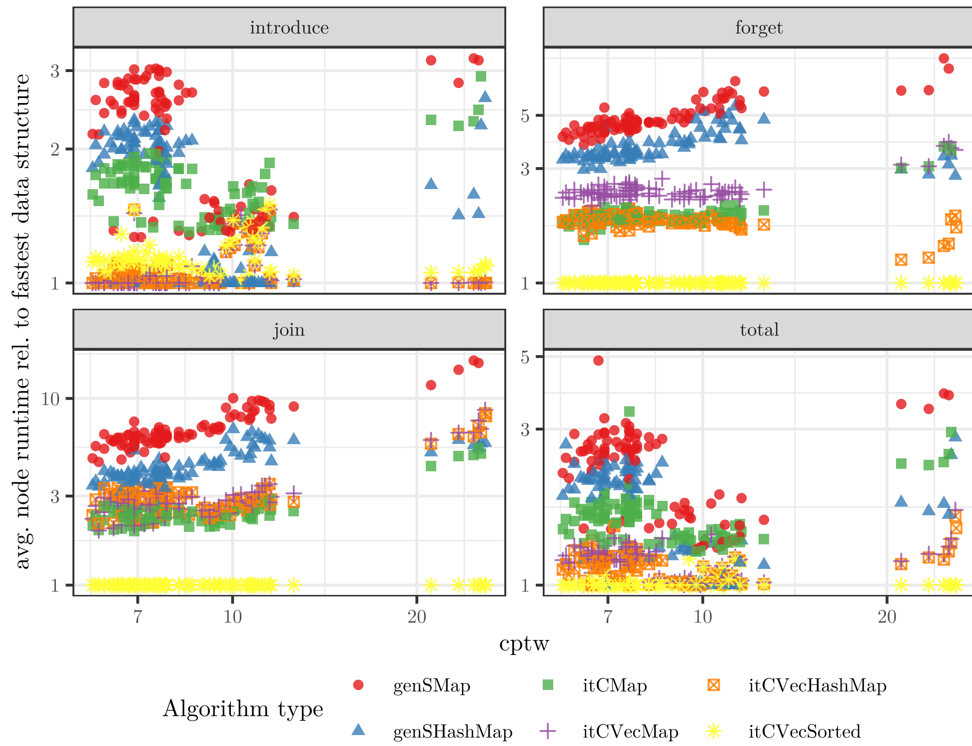
Figure 3.1 shows these results. The runtime for each node type is shown relative to the minimum runtime of any examined data structure. That means a value of 1 shows that the given data structure was the fastest for the given instance and node type. First, we find that the positive instance driven algorithm seems to be faster than the generative algorithm. Next, we want to look at the individual node types in more detail. Comparing the data structures on the node types, we can see that the simpler data structures using an array seem to show a significant advantage. In contrast, even though the asymptotic runtimes are similar, the additional overhead of the map data structures leads to a significant disadvantage for this node type. When looking at the forget node, we can see that the sorted array performs better than the unsorted array, even though the unsorted array has the faster asymptotic runtime. This highlights the importance of this experimental evaluation, as the results can be quite different than the theoretical analysis. The join node demonstrates another advantage of the array data structures. The contiguous memory composition allows for much faster iteration, in comparison to the more complex iteration of a map. In particular, the sorted array outperforms all other data structures. This advantage pervades throughout the whole algorithm and yields an advantage to the sorted array in the overall runtime. Consequently, the implementation using the sorted array will be used for all further experiments.

3.4 Comparison with State of the Art

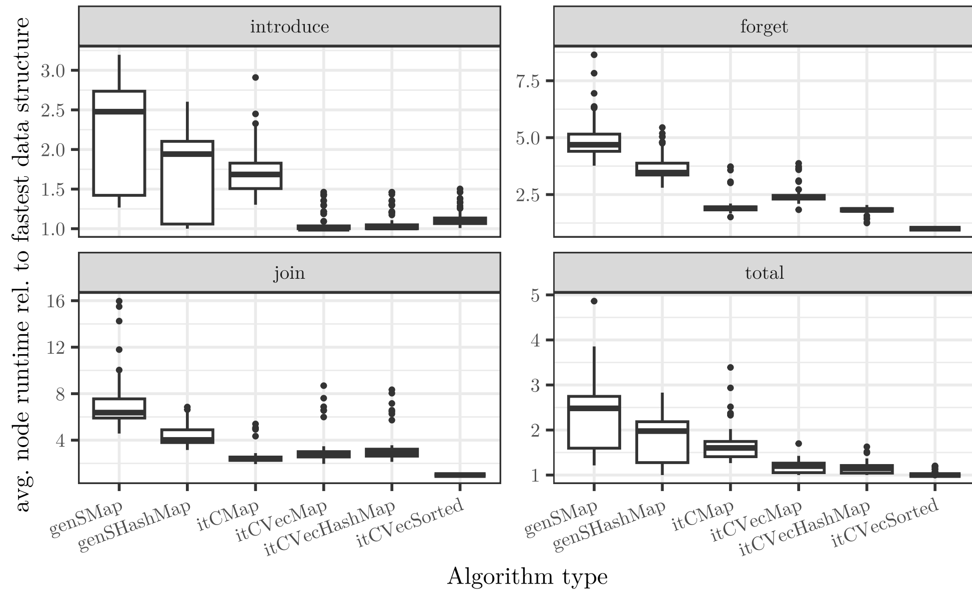
In the previous section, we have analysed our algorithm and its utilized data structures with the goal to improve its performance. Overall, the final version outperforms the initial algorithm by a large factor. Thus, our efforts were quite successful. With this results, we want to evaluate our algorithm in regards to other state of the art solver for Maximum Independent Set. As a comparison, we chose the winner of the PACE 2019 vertex cover challenge [DFH19] *We Got You Covered* by Hespe, Lamm, Schulz, and Strash [HLSS19].

For this, we evaluated the algorithms on the *girgs* data set described in Section 2.5 and the real-world networks. Then, for each instance, we measure the execution time for both our algorithm (MIS) and the We Got You Covered algorithm (WGYC). The results are shown in Figure 3.2.

We find that our algorithm outperforms the comparison on some instances. However, the majority of instances are significantly slower on our algorithm, sometimes even as much as 1000 times slower. A considerable difference like this is usually due to a major algorithmic advantage. After further investigation, we find that a big part of this, is likely the utilization of reduction rules. The We Got You Covered algorithm applies a series of reduction rules to calculate a kernel of the input graph. These reduction rules can be calculated very efficiently, while the size of the kernel can be much smaller than the original input, explaining this huge difference in runtime. Thus, we want to examine these and other potential algorithmic optimizations for our algorithm in the next chapter.



(a) The relative runtime in regards to the cp-treewidth. The data structure used is shown by the shape and colour and the x-axis shows the cp-treewidth. Note the logarithmic scale on the y-axis.



(b) The relative runtime in a boxplot. The x-axis shows the different data structures used.

Figure 3.1: The average runtime for each node type relative to the minimum runtime of all data structures on this node type on girgs. The relative runtime is shown on the y-axis. The algorithm types starting with `genS` use the generative algorithm while the algorithms starting with `itC` use the positive instance driven algorithm. The bottom right shows the total runtime of the algorithm.

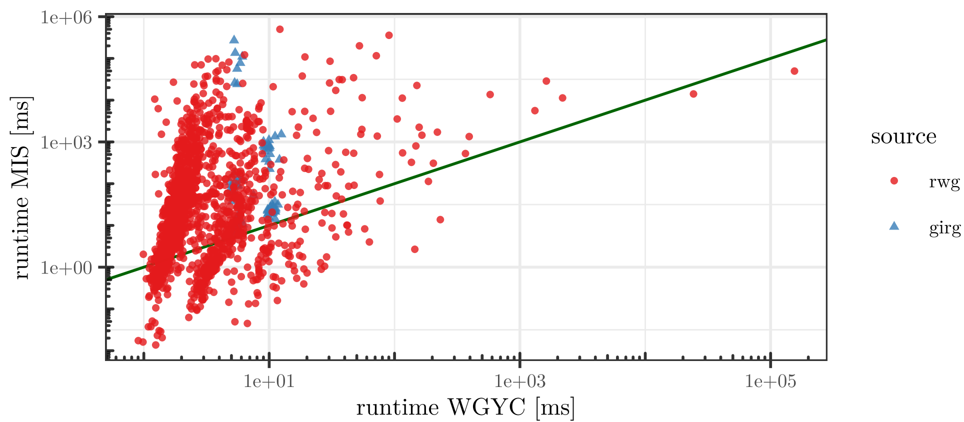


Figure 3.2: Runtime comparison between our algorithm and the We Got You Covered algorithm on both the *girgs* data set and real-world networks. The x-axis shows the runtime for the WGYC algorithm, while the y-axis shows the runtime of our algorithm. Note the logarithmic scale on both the x- and y-axis. The green line indicates where the runtime of both algorithms match.

4 Algorithmic Optimizations

As we discovered in the previous chapter, our algorithm is significantly outperformed by another state of the art Maximum Independent Set solver. Our hypothesis is that this is due to the reduction rules and other possible optimizations used in this solver. Thus, in this chapter we want to evaluate these reduction rules and other possible algorithmic optimizations. First, we present and describe the reduction rules and their application. Then, we evaluate their effect on the cp-treewidth and the performance of our algorithm. Additionally, we propose a strategy to use upper and lower bounds to prune some solutions from our search space. We evaluate multiple different upper and lower bound implementations and their impact on the runtime of our algorithm. Lastly, we evaluate whether the choice of root in the nice tree decomposition has an impact on the performance of our algorithm.

4.1 Reduction Rules

As stated above, a common practice for Maximum Independent Set solvers is the computation of a kernel to obtain a smaller input instance before applying the maximum independent set algorithm. This kernel is computed by repeatedly applying a set of reduction rules $r_1, \dots r_k$.

A collection of reduction rules and the method of applying them has been studied by Akiba and Iwata [AI16]. These rules have then be used by Hespe, Lamm, Schulz, and Strash [HLSS19] in a high performance solver for vertex cover which won the PACE 2019 Track 1a Vertex Cover/Exact challenge [DFH19]. Our implementation uses an adaptation of their [HLSS19] code. We now discuss the specific reduction rules and the process of applying them. For more details and proofs we refer to Akiba and Iwata [AI16].

We apply the reduction rules to the input graph before computing a tree decomposition to yield a smaller kernel. We then compute a tree decomposition of the kernel and apply our dynamic program. To compute the kernel, the reduction rules are repeatedly applied in order from r_1 to r_k for all vertices. If a reduction rule r_i reduces at least one vertex we restart at rule r_1. If all reduction rules $r_1, \dots r_k$ have been applied and the last reduction rule r_k did not reduce any vertices, all rules have been applied exhaustively. The resulting graph is the kernel. There are two types of reduction rules. The first type allows us to remove vertices from the graph, while adding some of them to our independent set. The second type removes some vertices from the graph and replaces them with new ones. After an independent set is calculated on this graph, we can choose which original vertices are part of the final independent set. If a maximum independent set of the kernel is found, every reduction can be undone in reverse order to yield a maximum independent set of the original graph. As a maximum independent set cannot contain two adjacent vertices, every time a reduction rule states a vertex can be included into the maximum independent set, all its neighbours can be automatically excluded from the independent set and removed from the graph as well. This can be interpreted as its own simple reduction and for brevity we assume this is used in all following reduction rules. We now describe each utilised reduction rule in more detail.

The following reductions have been proposed by Fomin, Grandoni, and Kratsch [FGK09].

Degree-1 reduction. If a graph contains a vertex v of degree at most one, there always exists a maximum independent set which contains v. So v can be removed from the graph and included into the maximum independent set.

Dominance rule. If the closed neighbourhood of a vertex v includes the closed neighbourhood of another vertex u, that is, $N[u] \subseteq N[v]$, we say v dominates u. If a vertex v dominates another vertex u, there always exists a maximum independent set that contains u. Therefore, u can be included into the maximum independent set and removed from the graph.

Degree-2 folding. If v is a vertex of degree two with neighbours u and w where u and w are not adjacent, either v or u and w are in some maximum independent set. Thus, $N[v]$ can be contracted to a single new vertex x. Depending on whether x ends up in the computed maximum independent set either v or $\{u, w\}$ can be included into the maximum independent set.

Linear programming relaxation. A LP-based reduction rule for vertex cover was first developed by Nemhauser and Trotter [NT75]. They formulate a LP relaxation of vertex cover with the variables $x_v \in \mathbb{R}$ for $v \in V(G)$ as follows:

$$\begin{aligned} \text{minimize} \sum_{v \in V(G)} x_v \\ \text{such that: } x_u + x_v \geq 1 \text{ for } \{u, v\} \in E(G) \\ x_v \geq 0 \text{ for } v \in V \end{aligned}$$

They showed that an optimal half-integral solution (that is, every variable is either 0, 1 or $\frac{1}{2}$) exists and additionally, a maximum independent set including all vertices with the value of 1 in the solution exists as well. Consequently, every vertex with a value of 1 can be removed from the graph and included in the maximum independent set. Nemhauser and Trotter also presented an algorithm which calculates the optimal half-integral solution using a bipartite matching. We further use an optimization by Iwata, Oka, and Yoshida [IOY13] which calculates a solution with a minimal half-integral part.

Unconfined vertices. Xiao and Nagamochi [XN13] define a vertex v to be unconfined, if it satisfies the following algorithm. Start with $S = \{v\}$. Then find $u \in N(S)$ such that $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ is minimized. If no such vertex exists, v is confined. If $N(u) \setminus N[S] = \emptyset$, then v is unconfined. If $N(u) \setminus N[S] = \{w\}$, then add w to S and repeat the algorithm. Otherwise, $|N(u) \setminus N[S]| > 1$ and v is confined. For every unconfined vertex v there always exists a maximum independent set that does not contain v, therefore, v can be removed from the graph.

Xiao and Nagamochi [XN13] propose the following two additional reductions which, together with the other rules, can eliminate nearly all vertices with degree three or less from the graph.

Twin vertices. If u and v are vertices of degree three, such that $N(u) = N(v)$, they are called twins. If there exists a twin u, v, we remove u, v and $N(u)$ from the graph. If there exists an edge among $G[N(u)]$, we add $\{u, v\}$ to the maximum independent set. Otherwise, we add a new vertex w which is connected to $N^2(u) \setminus \{v\}$. If w is in the computed maximum independent set, $N(u)$ is added to the maximum independent set, otherwise $\{u, v\}$ are added to the maximum independent set.

Alternative vertices. Two subsets of vertices A and B are called alternatives if $|A| = |B| \geq 1$ and there exists a maximum independent set S, such that $S \cap (A \cup B)$ is equal to A or B. When there exists an alternative A and B, we can remove $N(A) \cap N(B)$, A and B from the graph and introduce an edge between every $x \in N(A) \setminus N[B]$ and every $y \in N(B) \setminus N[A]$. Finally, A can be added to the maximum independent set, when $N(B) \setminus N[A]$ is part of the computed maximum independent set, otherwise $N(A) \setminus N[B]$ will be part of the computed maximum independent set and B can be added to the maximum independent set. There are two structures which Xiao and Nagamochi [XN13] present as an alternative: (1) Two vertices u and v are called a *funnel* when $N(v) \setminus \{u\}$ induces a complete graph. For a funnel u and v, $\{u\}$ and $\{v\}$ are alternatives. (2) Let $a_1b_1a_2b_2$ be a chordless 4-cycle, $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$. $a_1b_1a_2b_2$ is called a *desk*, if every vertex has degree at least three, $N(A) \cap N(B) = \emptyset$, $|N(A) \setminus B| \leq 2$ and $|N(B) \setminus A| \leq 2$. For a desk $a_1b_1a_2b_2$, A and B are alternatives.

Akiba and Iwata [AI16] also present another form of reductions, so called packing reductions. These are inequalities maintained throughout the branching process and thus cannot simply be applied to our approach.

4.1.1 Evaluation

The reduction rules described above prove to be quite effective for most graphs. Sometimes even reducing the input graph completely, rendering the DP for calculating the maximum independent set unnecessary. Though, in our application, in addition to graph size, the cp-treewidth is a quite important parameter for the runtime of our algorithm. Consequently, we will evaluate the impact these reduction rules will have on the cp-treewidth

The results of this experiment are shown in Figure 4.1. First, we can see that for many instances the reduction rules already completely reduce the input and thus, solve the Maximum Independent Set problem by themselves. Looking at the graphs which are not reduced completely, we can see that the cp-treewidth either stays similar or is reduced by a small factor. In addition, for the instances where the cp-treewidth does not change significantly, the kernel size also does not reduce much from the input. Thus, we can assume there are certain graphs, for which the reduction rules are not very efficient, both in terms of kernel size and cp-treewidth reductions. For instances where the cp-treewidth is reduced, the kernel size is also smaller than the input by usually more than 50%. As a result, we can say that on instances where the reduction rules are able to reduce the kernel size, the cp-treewidth is usually reduced as well. In conclusion, we observe that the reduction rules are very effective at reducing the input size with only a small number of graphs proving to be almost irreducible.

Next, we want to repeat our evaluation from Section 3.4 with the addition of applying reduction rules before executing our algorithm. For this, we use the same experiment setup from before, except, this time we reduce the input graph fully, before running our algorithm. That is, after running all reduction rules exhaustively, we calculate a tree decomposition which we use, together with the kernel, as the input to our algorithm. We measure the runtime

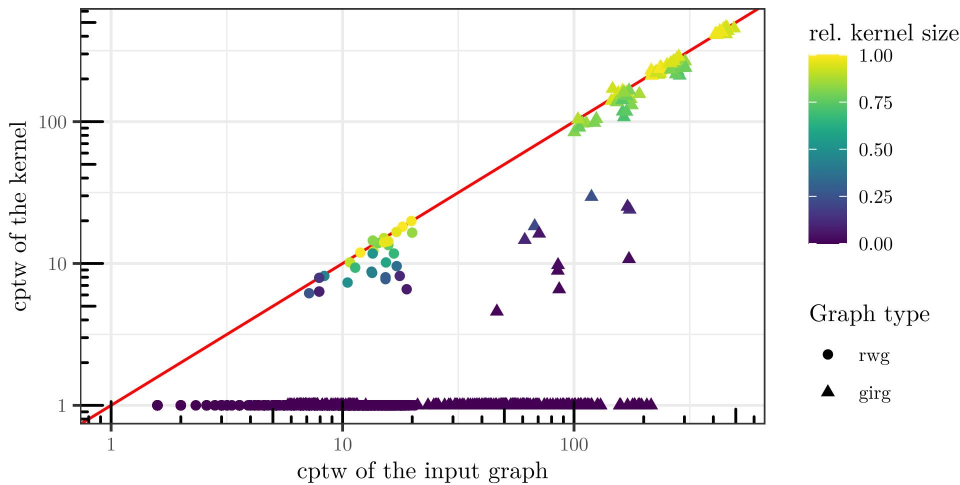


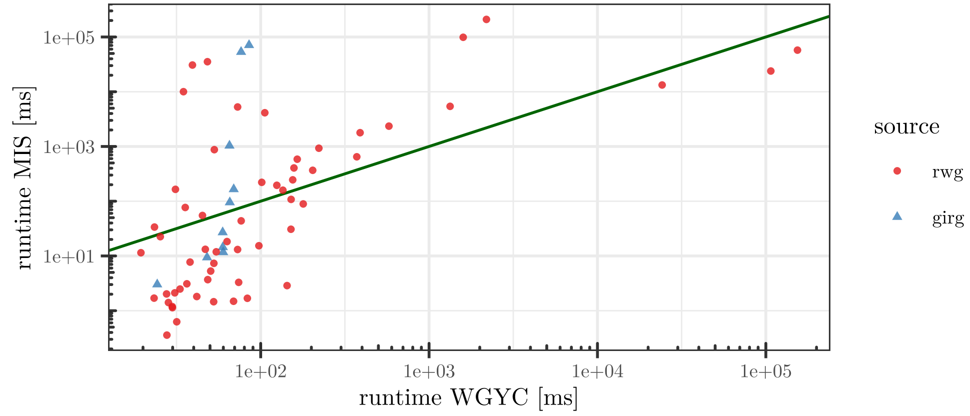
Figure 4.1: Evaluation of the impact the reductions have on the cp-treewidth on both the *girgs* data set and the real-world networks. The graph shows the cp-treewidth of the input graph on the x-axis and the cp-treewidth of the kernel on the y-axis. Note the logarithmic scale on both axes. The colour shows the relative kernel size. Whether the input graph was a real-world network or a girg is shown in the shape of the data points.

as the time taken to apply all reductions rules and the runtime of our algorithm. That is, we don't include the time to calculate the tree decomposition in the runtime. As the We Got You Covered algorithm and our implementation use basically the same reduction rules, we only consider instances which were not fully reduced in our evaluation.

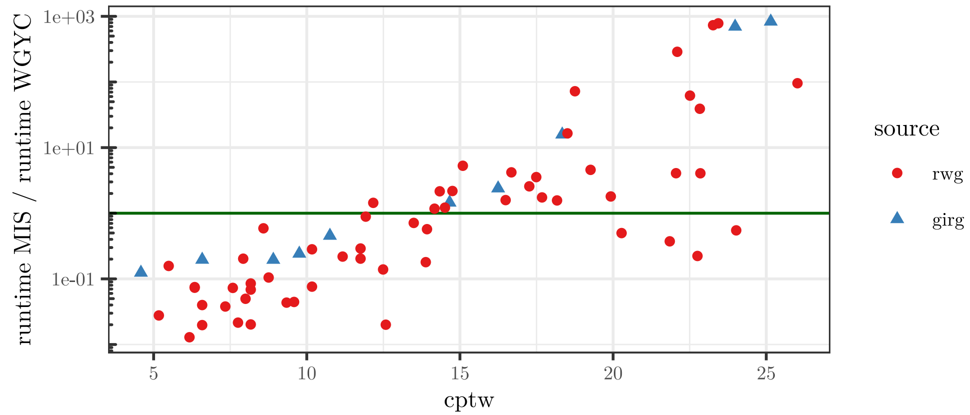
These results are shown in Figure 4.2a. As we can see, now the algorithms are much closer in their runtime. There seem to be a few outlier, where our algorithm is severely outperformed, but other than that, the algorithms are quite close in their runtime, with our algorithm outperforming the comparison on average. To further investigate these outlier, we look at the relative runtime of both algorithms in regards to the cp-treewidth of the kernel in Figure 4.2b. Here, we can see that these outliers have a large cp-treewidth. This is in line with our expectation, that our algorithm is only suitable for inputs with a relatively small cp-treewidth. When looking at inputs that have a small cp-treewidth, we observe that our algorithm consistently outperforms the comparison. This confirms our proposal, that this algorithmic strategy is indeed a viable alternative to other state of the art solvers for Maximum Independent Set on inputs with a small cp-treewidth.

4.2 Upper/Lower Bound Pruning

After showing the effectiveness of reduction rules, we now want to have a closer look at using upper and lower bounds to prune the search space. This is a common strategy for other Maximum Independent Set solvers using a branch and bound strategy [Lam+19 | HLSS19 | AI16]. First, we discuss how pruning is possible inside our DP and how it differs to regular branch and reduce solver. After proving the theoretical potential of this approach, we propose multiple different lower and upper bounds. Then, we evaluate these bounds on how exact and how effective the are. Lastly, we show the effect these bounds have on the runtime of our algorithm.



(a) Comparison over total runtime. The x-axis shows the runtime of the wgyc algorithm and the y-axis shows the runtime of our algorithm. Note the logarithmic scale on both axes.



(b) Comparison over cp-treewidth. The x-axis shows the cptw of the kernel and the y-axis shows the runtime of our algorithm divided by the runtime of the WGYC algorithm. Note the logarithmic scale of the y-axis.

Figure 4.2: Runtime comparison between our algorithm and the We Got You Covered algorithm on the real-world networks and the *girgs* data set. The shape and colour of the data points indicate whether the input graph was a real-world network or a girg. The green line indicates where the runtime of both algorithms match.

4.2.1 Pruning

To prune the search space, after calculating the solution for a given node X and a subset S, we can examine whether they violate some easily verifiable upper or lower bound. If they do, we can discard the result and do not have to consider it in the following calculations. While regular branch and bound solvers for the Maximum Independent Set problem can examine the current search tree for global bounds, we cannot do that in our dynamic program as the solutions of the subproblems are not maximum independent sets of a subgraph but rather the maximum independent set $\hat{S}$ of the subgraph $G[B(T_X)]$ with the additional condition that $\hat{S} \cap X = S$. As such, we always have to consider the bounds of the remaining graph as well.

Let X be a tree node and S be a subset of $B(X)$. We define $G_{\text{rem}} := G - B(T_X)$. Additionally, let ℓ be some lower bound for the maximum independent set size on the whole graph G and u_{rem} some upper bound on G_{rem}.

Theorem 4.1: *If $c[X, S] + u_{rem} < \ell$ then there is no maximum independent set $\hat{S}$ of G that will intersect the bag $B(X)$ such that it is equal to S. That is, $\hat{S} \cap B(X) = S$.*

Proof. Let's assume there is a maximum independent set $\hat{S}$ of G which intersects $B(X)$ such that $\hat{S} \cap B(X) = S$.

We define $\hat{S}_{\text{rem}} := \hat{S} \cap V(G_{\text{rem}})$. This means $\hat{S}_{\text{rem}}$ is an independent set in G_{rem} and as such $|\hat{S}_{\text{rem}}| \leq u_{\text{rem}}$. Additionally, we define $\hat{S}_{sub} := \hat{S} \cap B(T_X)$, which is an independent set in $G[B(T_X)]$, which satisfies $\hat{S}_{sub} \cap B(X) = S$. According to Equation (2.1), this means $|\hat{S}_{sub}| \leq c[X, S]$. We also know that $|\hat{S}| \geq \ell$ and by definition of G_{rem} we know that $\hat{S} = \hat{S}_{sub} \dot{\cup} \hat{S}_{\text{rem}}$. Putting everything together, we get

$$\ell \leq |\hat{S}| = |\hat{S}_{sub}| + |\hat{S}_{\text{rem}}| \leq c[X, S] + u_{\text{rem}} < \ell \ .$$

This is a contradiction to our assumption that such a maximum independent set exists. ∎

As a result of the above theorem, a subset of a tree node not satisfying this bound will never be used to calculate the correct solution. Consequently, its value can be set as invalid and thus reducing the number of calculations required in further nodes.

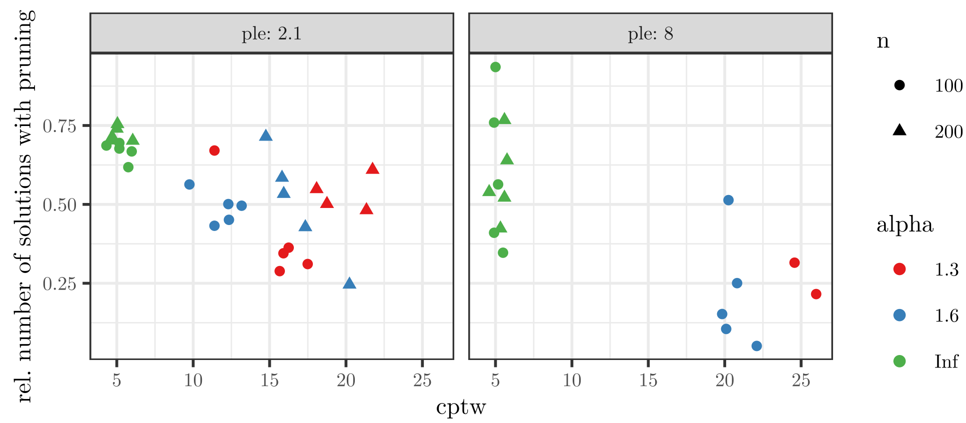
4.2.1.1 Theoretical Potential

To investigate the impact of applying these bounds, the positive instance driven algorithm using the sorted array has been implemented with this pruning strategy. For the evaluation, we keep track of certain metrics while running this algorithm. We both count the number of solutions that were pruned, as well as the number of solutions that were actually calculated. The number of pruned solutions by itself is not a good measure for the effectiveness, as pruning a solution also allows us to ignore any further solutions, which depend on the initial pruned solution. This means pruning a solution early on in the DP removes significantly more solutions from the search space, as a solution which is pruned much later. As a result, we mainly use the total number of calculated solutions for our evaluation, as this is a better reflection of the computational complexity required for the algorithm.

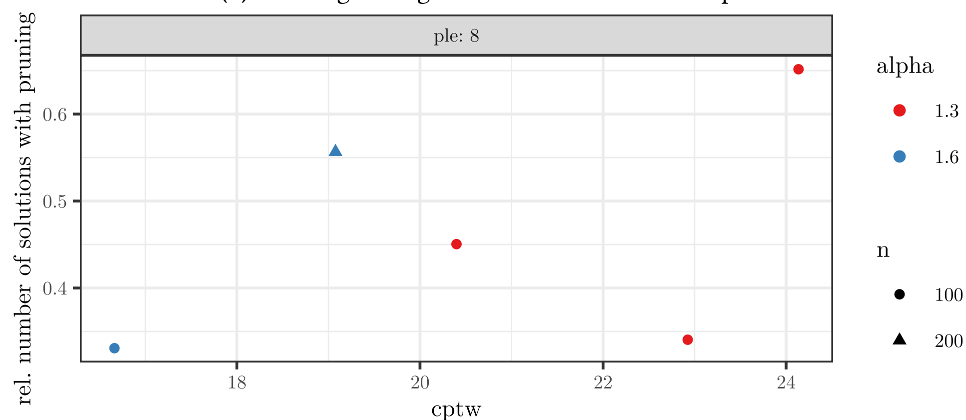
Using these metrics, first, we want to evaluate the maximum theoretical effectiveness of our pruning strategy. That is, assuming we have a perfect bound, how many solutions can be pruned from our search space. To achieve that, we use the We Got You Covered [HLSS19] algorithm to calculate the independence number instead of calculating an upper/lower bound. While this is obviously not very performant, it can be used to evaluate the potential of our pruning strategy.

For the evaluation, we use the *small girgs* dataset described in Section 2.5 as our input, as the calculation of the exact bounds is computationally quite complex. Figure 4.3 shows the number of solutions which had to be calculated when pruning with perfect bounds relative to the number of solutions which had to be calculated with no pruning in respect to the cp-treewidth of the graph. We evaluate the results both on the unreduced input, as well as the kernel input.

We find, that the potential for pruning gets larger when the cp-treewidth increases. Overall, pruning seems to be quite promising, pruning more than 50% of the solutions for most instances with a cp-treewidth of more than 15. When running on the kernel of a graph, we get far fewer results, as many instances are already solved by the reduction rules. This means the results are less representative, as we have far fewer data points. However, our results indicate that pruning is still effective, though less so than on regular graphs.



(a) Running the algorithm on an unreduced input.



(b) Running the algorithm on the kernel of girgs.

Figure 4.3: Relative number of solutions calculated when pruning with perfect bounds. The input are unreduced *small girgs* instances with different values for α (colour) and power-law exponent (left to right) (a) and their respective kernel input (b). The x-axis shows the cp-treewidth and the y-axis shows the number of solutions calculated when pruning with perfect bounds divided by the number of solutions calculated when using no pruning.

4.2.2 Specific Bounds

After showing the potential of pruning, we want to evaluate the pruning strategy on different upper and lower bounds. The upper bound has to be calculated once for every bag of the tree decomposition, while the lower bound only has to be calculated once for the whole graph. As such, the performance of the upper bound is a lot more important.

First, as we have already shown the potential of the reduction rules from Section 4.1, it is evident to take advantage of them before calculating an upper or lower bound. That is, we can use the reduction rules to calculate a kernel of the graph before calculating the bound. Reducing the graph hopefully allows us to reduce the graph size, therefore speeding up the bound calculation. In addition, the smaller graph size will mean the inaccuracy of the bound will hopefully have a smaller impact on the final bound. As a result, we will evaluate all of the following upper or lower bounds with and without this reduction strategy. This means we now have two places in our algorithm where reduction rules may be used. First, they can be used prior to the execution of our whole algorithm, so the input to our algorithm is already a

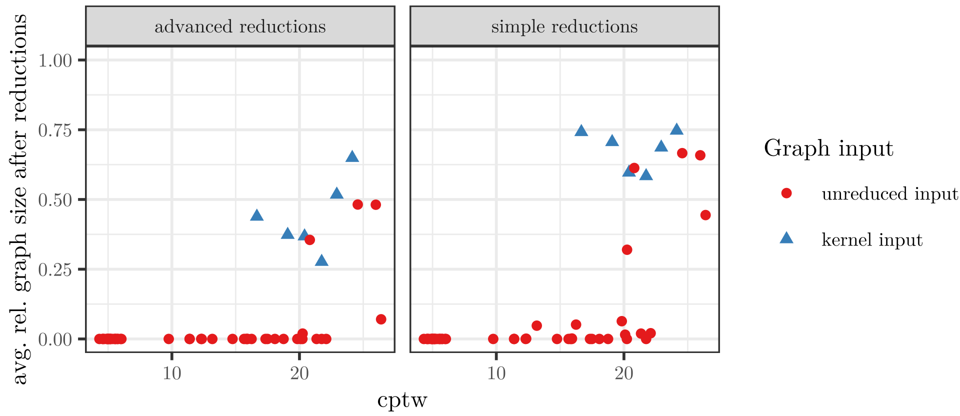


Figure 4.4: Average relative kernel size after applying the reduction rules on the *small girgs* instances. The x-axis shows the cp-treewidth and the y-axis shows the average of the kernel size relative to the input size, every time a bound is calculated. The left shows the kernel calculated with advanced reduction rules and the right side shows the kernel only calculated with simple reductions. The color and shape indicate whether the input was unreduced or a kernel.

kernel (*kernel input*), in comparison to an *unreduced input*. Second, they can be used prior to each execution of a bound, which we will call *bounds with reductions* in comparison to *bounds without reductions*, when they are not used.

However, some of the upper bound algorithms we propose precompute some information about the graph once in the beginning, then reuse that information for every subgraph. As a result, modifying the structure of the graph will break these algorithms. That is, not just removing vertices, but adding new vertices or inserting new edges. This means we can't use some of the reduction rules (in particular twin vertices and alternative vertices), which introduce new vertices. We will call these reduction rules *advanced reductions* while the remaining rules, which can be used with every bound, are called *simple reductions*.

We want to evaluate this reduction strategy. For this, we use the *small girgs* described in Section 2.5. For every graph, we calculate the average of the kernel size after reductions divided by the subgraph size before the reductions, on both the unreduced input and the kernel input. This is shown in Figure 4.4, with advanced reductions on the left and using only simple reductions on the right.

As expected, for the unreduced input graphs, the reductions are quite effective, reducing most instances completely. Those instances, which are not reduced completely are usually reduced to an average of less than 50% of the input size. Additionally, when reducing with only simple reductions, the average size gets slightly larger, though still significantly smaller than without reductions. Next, we find that even when using a kernel input, the reduction rules still prove to be effective. Consequently, a subgraph of a fully reduced graph can usually be reduced even further. As a result, we find that applying these reductions yields a large advantage in terms of the remaining graph size.

4.2.2.1 Lower Bound

According to a comparison of different approximation algorithms for the independence number from Dahshan [Dah14], the greedy-min algorithm performs well for most graphs. It chooses the vertex with the minimal degree in every step and removes all neighbours from the graph. This is repeated until no more vertices are in the graph. The set of chosen vertices is independent and as such it's size a lower bound for the independence number. This algorithm will be referred to as *greedyMin*.

Another approach takes advantage of the reduction rules proposed in Section 4.1. We start by calculating the kernel of the graph. If the kernel is empty, we already have an exact maximum independent set, otherwise we remove the vertex with the largest degree. Then, we calculate the kernel again and repeat until the kernel is empty. This algorithm will be referred to as *greedyMax*.

4.2.2.2 Upper Bound

A simple upper bound for the independence number of a graph G is the size of a clique cover. That is, for a set of disjoint cliques $C_1, \dots C_k$ of G, such that $C_1 \cup \dots \cup C_k = V(G)$, we can choose at most a single vertex from every clique for an independent set. As a result, the independence number of a graph is always smaller or equal to k.

greedyDegCC. A greedy algorithm for finding a clique cover for a graph is presented by Akiba and Iwata [AI16]. This algorithm iterates over each vertex in ascending order of their degree. Then, a vertex v is added to a clique of the clique cover, or if no such clique exists, a new one with v as its single vertex is added. This algorithm will be referred to as *greedyDegCC*. When using bounds with reductions, we can use all reduction rules for this bound.

greedyQCCC. Another greedy algorithm uses the Quick Cliques algorithm [ES11 | ELS13] to first find all maximal cliques in a graph. We will only calculate the maximal cliques once for the whole graph, as it takes quite long to compute and then reuse these maximal cliques restricted to the respective subgraph for every bag. Then, the largest maximal clique is chosen for the cover and all containing vertices are removed from the remaining maximal cliques. This is repeated until all vertices are covered. This algorithm will be referred to as *greedyQCCC*. When using bounds with reductions, we can only use simple reduction rules for this bound.

minimalCC. A modification of this algorithm uses a Hitting Set solver by Bläsius, Friedrich, Stangl, and Weyand [BFSW22] to calculate a minimal clique cover from the list of maximal cliques. That is, a clique cover as described above with the minimal number of cliques. Given a universe $\mathcal{U}$ and a set of subsets $\mathcal{S} \subseteq 2^{\mathcal{U}}$, the Hitting Set solver calculates a set cover $C \subseteq \mathcal{S}$ with minimal cardinality. That is, $\bigcup_{C_i \in C} C_i = \mathcal{U}$ and $|C|$ is minimal. Given a graph G with all maximal cliques in $M = \{M_1, \dots, M_k\}$, we define $\mathcal{U} := V(G)$ and $\mathcal{S} := M$. The resulting set cover C calculated by the Hitting Set solver is a minimal clique cover. We refer to this algorithm as *minimalCC*. Additionally, a minimal clique cover is indeed only an upper bound for the independence number. The graph shown in Figure 4.5 has a minimal clique cover of size 3 and an independence number of 2. When using bounds with reductions, just as for the *greedyQCCC* algorithm, we can only use simple reduction rules for this bound.

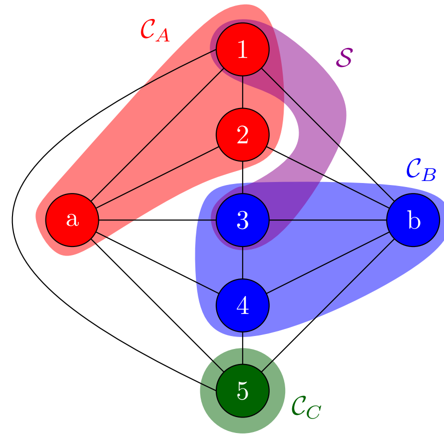


Figure 4.5: A graph whose minimal clique cover ($\mathcal{C} = \{\mathcal{C}_A, \mathcal{C}_B, \mathcal{C}_C\}$, $|\mathcal{C}| = 3$) is larger than its independence number ($\mathcal{S}$, $|\mathcal{S}| = 2$).

As stated above, the upper bound has to be calculated for the remaining graph $G_{\text{rem}} = G - B(T_X)$ for every bag $B(X)$ of the tree decomposition. This can obviously be done by calculating the clique cover with one of the algorithms described above. However, this method requires the calculation of a full clique cover for every bag. In contrast, a single clique cover of the whole graph can be calculated once. Then, for the subgraph G_{rem} of G, every clique of the cover which contains a vertex from G_{rem} is a clique cover of G_{rem}. That is, $\{C_i | i \in \{1, \ldots, k\} \wedge C_i \cap V(G_{\text{rem}}) \neq \emptyset\}$ is a clique cover for G_{rem}. This will result in a worse bound, but requires far less computation. Additionally, when reducing before utilizing this strategy, we can only use simple reduction rules. Thus, we will additionally consider all of the above bounds with this modification. These algorithms will be referred to by the same name with a *GlobalCC* added to the name.

4.2.3 Bounds Evaluation

In the following, we evaluate and compare the different bound algorithms proposed above. For this, we track the same metrics as before, with the addition of the accuracy of the considered bound. That is, the difference between our calculated bound and the exact independence number. For the evaluation, we use the *small girgs* dataset described in Section 2.5, as the calculation of the exact bounds for comparison is computationally quite complex. We execute the algorithm both on each unreduced input, as well as its kernel input.

During the execution, each time a bound is calculated (that is once for every bag for the upper bound and once for the lower bound), we additionally calculate the exact independence number. Then, we calculate the delta between these two numbers. The average of this number is shown in Figure 4.6 in respect to the exact solution. The same data in the form of a box plot is shown in Figure 4.7.

We find that, as expected, using bounds with reductions is very effective with each bound significantly outperforming its counterpart without reductions. One exception, where there is no difference, are the lower bounds with a kernel input. This can be explained by the fact that the lower bounds are calculated on the whole graph. When the input is a kernel this graph is already reduced, thus we cannot reduce any further before calculating the lower bound.

Comparing the lower bounds, we can see that the *greedyMax* outperforms the *greedyMin* algorithm significantly, when not using reductions. As the *greedyMax* algorithm already profits from the reductions internally, this is expected. When using reductions, this gap decreases. However, especially when the input is a kernel, we can still see an advantage of the *greedyMax* algorithm. This can be explained by the repeated application of reductions which allows this bound to profit from them even more, in comparison to the *greedyMin* algorithm.

Comparing the upper bound algorithms is not as clear cut. Overall, we can see that the variants using the single global clique cover perform slightly worse than their counterpart. This is expected and we will evaluate the impact on performance later on. Additionally, we find that the *greedyDeg* algorithm usually outperforms the *greedyQCCC* one, as this bound is also the computationally more efficient one, we expect this advantage to only increase when considering runtime. As expected, the *minimalCC* algorithm calculates the best bound. As all upper bounds are based on a clique cover, the *minimalCC* bound will always perform the best. The only exception occurs when using reductions, where the advanced reductions usable by some bounds might reduce some instances further than the simple reductions allow for the *minimalCC* bound.

Next, we evaluate the bounds on their performance when utilized in the algorithm in Figure 4.8. It shows the number of solutions which had to be calculated with pruning relative to the number of solutions calculated without pruning. In contrast to the previous evaluation, which looked at the bounds independent of the actual algorithm, now, we look at how many solutions can actually be pruned with the given bound.

We can see that the number of solutions corresponds to the quality of the bounds shown before. As expected, a bound which was closer to the exact solution results in more solutions being pruned. However, as the bounds are not exact, the pruning is less efficient than theoretically possible. Additionally, we find that pruning on the kernel of a graph seems to be less efficient than on the regular graph itself. This is in part due to the fact, that those instances which cannot be reduced fully, are generally harder. However, pruning on these instances with an unreduced input is on average still quite effective, with the relative number of solutions that had to be calculated significantly lower, than on the same instances but with a kernel input.

Next, we want to evaluate the pruning in regards to runtime. Up until now we have shown that pruning will reduce the number of solutions we have to calculate. Now, we want to evaluate whether the savings in number of solutions to calculate compensates for the extra runtime required to calculate the bounds. For this, we use the same setup as before and measure the execution time for each bound algorithm.

Figure 4.9 shows the runtime with pruning in regards to the runtime without any pruning. The green line indicates where the runtimes are the same. Data points to the left of the line are faster without pruning, while data points on the right are faster with pruning. To better compare the runtime of different bounds to each other, the runtime relative to the minimum runtime of any bound is shown in Figure 4.10. Note that the *minimalCC* upper bound is removed from this plot, to preserve detail in the faster bounds, as it is significantly slower than any other upper bound. Almost all instances are more than 10 times slower when using the *minimalCC* upper bound in comparison to the fastest bound. The only exception occurs when using bounds with reductions on an unreduced input. Here, we find, that the reduction rules are often sufficient for calculating the bound, resulting in a comparable runtime to the other bounds. However, when these reductions aren't enough, the runtime of the *minimalCC* bound is usually more than 10 times slower than the minimum runtime of any other bound again.

First, we take a look at the results on an unreduced input. We see that, as expected, using bounds with reductions has a large runtime benefit. Without it, every bound performs similarly or worse than no pruning. Looking at the two lower bounds, we can see that the *greedyMax* algorithm performs slightly better than the *greedyMin* algorithm. Therefore, in this case, the better quality bound also has the advantage when considering runtime. Though the difference is quite small with the upper bound having a much larger effect on runtime than the lower bound. Looking at the upper bounds, we can see that the more accurate bounds perform significantly worse than the computationally more efficient ones. This is due to the fact, that the better quality does not compensate for the far higher runtime. As a result, the *minimalCC* algorithm, which performed the best in terms of quality, is consistently the worst regarding runtime. On the other hand, the *minimalGlobalCC* algorithm performs quite well, even outperforming the algorithm with no pruning in many instances. Thus, we can see that the strategy of only calculating a single global clique cover, sacrificing quality for runtime, is beneficial in this case. However, the *greedyDegCC* algorithm with reductions performs the best overall, being among the fastest bounds and having much fewer outlier when looking at its runtime relative to the other bounds. It outperforms the algorithm with no pruning in most, but not all instances. Especially instances with a higher overall runtime seem to benefit the most from pruning.

Looking at the runtime when the input is a kernel, we see a different picture. Overall, the same general conclusions can be drawn, however, no bound could outperform the runtime without pruning. As we noted above, the bounds seem to allow for less pruning on kernels and the additional runtime overhead of using bounds doesn't seem to be worth it for them. However, they still offer a benefit of memory consumption, resulting in a smaller DP table overall.

In conclusion, pruning seems like a promising strategy. Our theoretical experiments show that in principal, a large number of solutions can be pruned. However, for unreduced graphs, it leads to only a small runtime benefits, especially for harder instances. Looking at kernel inputs, the results are even less favourable, with no benefit or pruning in regards to runtime at all. Especially when considering that our algorithm is only competitive when used on a kernel input and then mostly for instances of lower cp-treewidth, we are not able to achieve any major benefits from using pruning.

4.3 Choice of Root

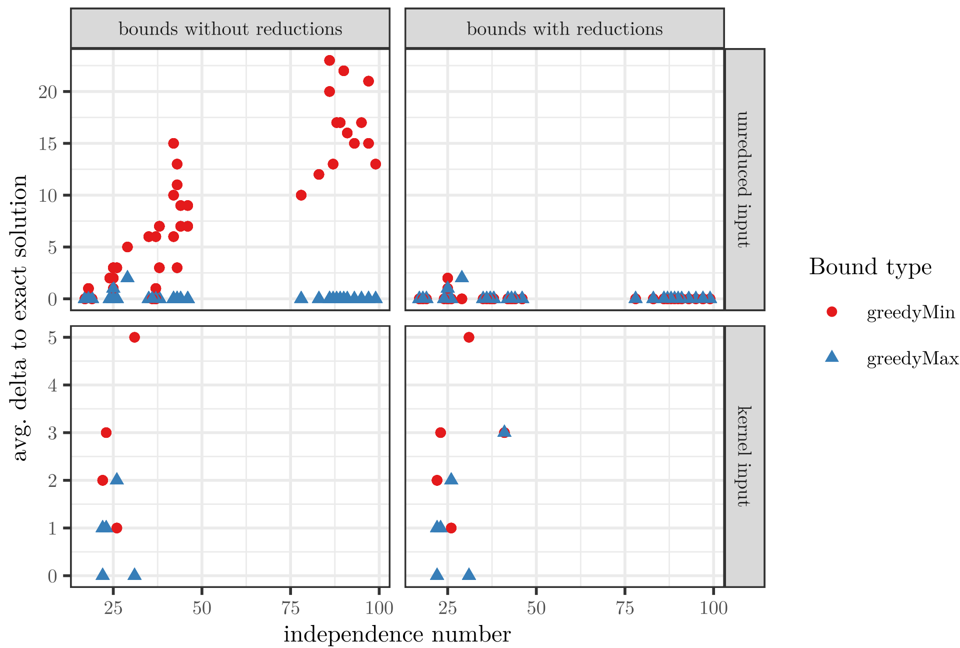
For the algorithms proposed in this work, we always assumed a tree decomposition to be rooted. We call a vertex a *nice* root, when the tree decomposition stays nice when rooted at that vertex. However, given a nice tree decomposition, we can easily choose an arbitrary vertex as the new root. Then, with some minor modifications, we can restore the niceness of the tree decomposition. This poses an interesting question, as to whether the choice of a root can have an impact on the runtime of the algorithm. Especially when using pruning, the choice of root could impact how much or at which point in the algorithm solutions can be pruned. This could potentially result in less solutions being calculated and as a result, a faster runtime.

To examine this question, we want to choose roots which lead to different traversals of the tree decomposition. In particular, we want to change the maximum depth during the traversal of the tree decomposition. Thus, we use the eccentricity of a vertex to choose different roots. Given a graph G, the eccentricity of a vertex v is the maximum distance to any other vertex $u \in V(G)$.

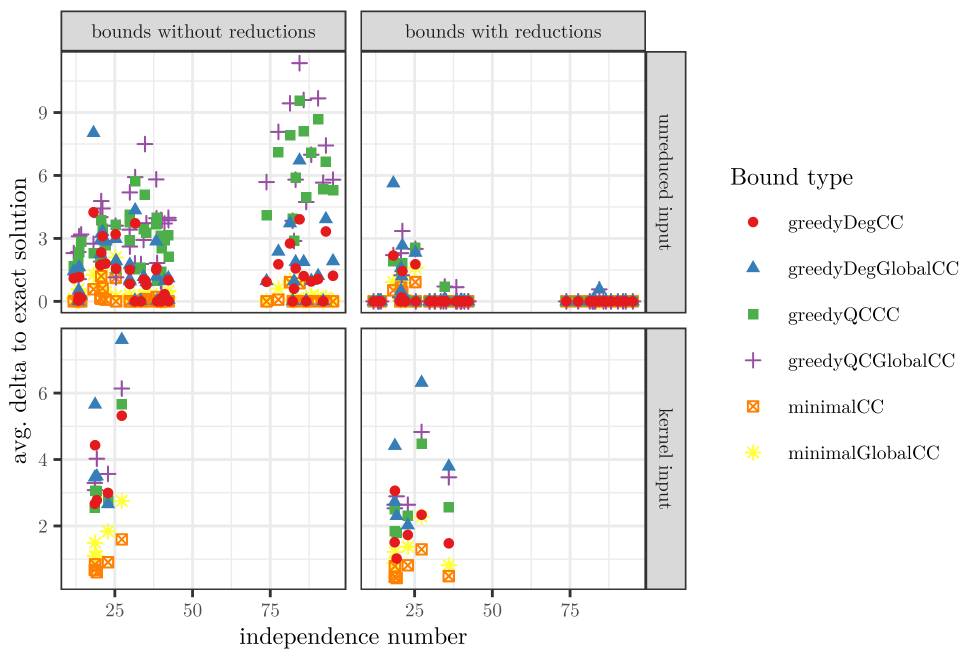
For the experiment we use the *small girgs* dataset described in Section 2.5. Additionally, we use the algorithm without pruning and the algorithm with the *greedyMax* lower bound, *greedyDegCC* upper bound and bounds with reductions. Then, we choose a nice root of the nice tree decomposition, in addition to both, three roots with a high eccentricity and three roots with a low eccentricity. We choose these roots by iterating over all vertices and including it every time it raises or lowers the overall eccentricity. This way we always include the vertex with the highest, as well as the vertex with the lowest overall eccentricity. Additionally, we avoid choosing multiple adjacent vertices on a path and promote the selection of vertices further apart. However, the selection of the vertices relies on the order of traversal and can thus vary based on how the vertices are numbered. During the algorithm, we track both the number of solutions calculated and the runtime required. Then, for each graph, we calculate the mean of both metrics over all considered roots, as well as the radius (minimum eccentricity of all vertices) and diameter (maximum eccentricity of all vertices) of the graph.

These results are shown in Figure 4.11. As expected, the results for the runtime and the number of solutions are quite similar. This means the number of solutions are a good approximation for the runtime in this case, excluding some outliers, where the runtime varies significantly more than the number of solutions.

Next, we can see that the results are similar for both unreduced graph inputs, as well as kernel inputs, with the difference for kernel inputs being generally smaller. We find, that the choice of root does influence both the runtime as well as the number of solutions. As expected, pruning does increase this influence, with an even larger difference between runtimes of different roots. Sometimes, the runtime for a specific root is less than 75% of the mean runtime when using pruning. Additionally, we can see that a faster runtime generally corresponds to a larger eccentricity. Though, not every root with a high eccentricity has a faster runtime, with some roots with a high eccentricity actually resulting in a slower runtime. Consequently, there is some correlation between eccentricity and runtime but nothing we can quantify exactly. In conclusion, choosing a root with a high eccentricity is a good strategy for now. Additionally, the vertex with the highest eccentricity will always be a nice root of the tree decomposition. As such, no modifications are required to restore the niceness of the tree decomposition.

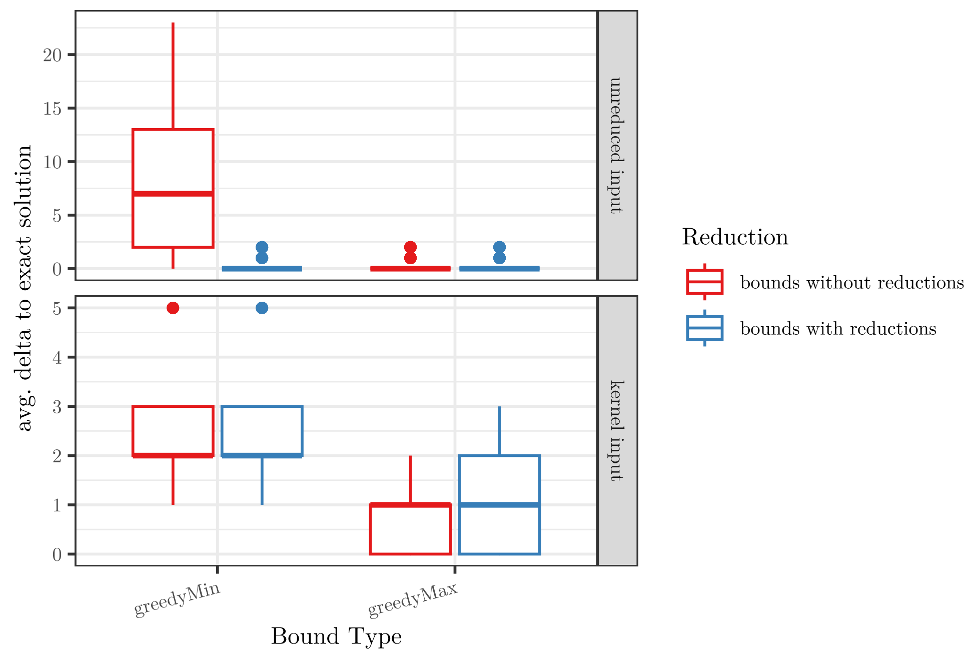


(a) Lower Bound.

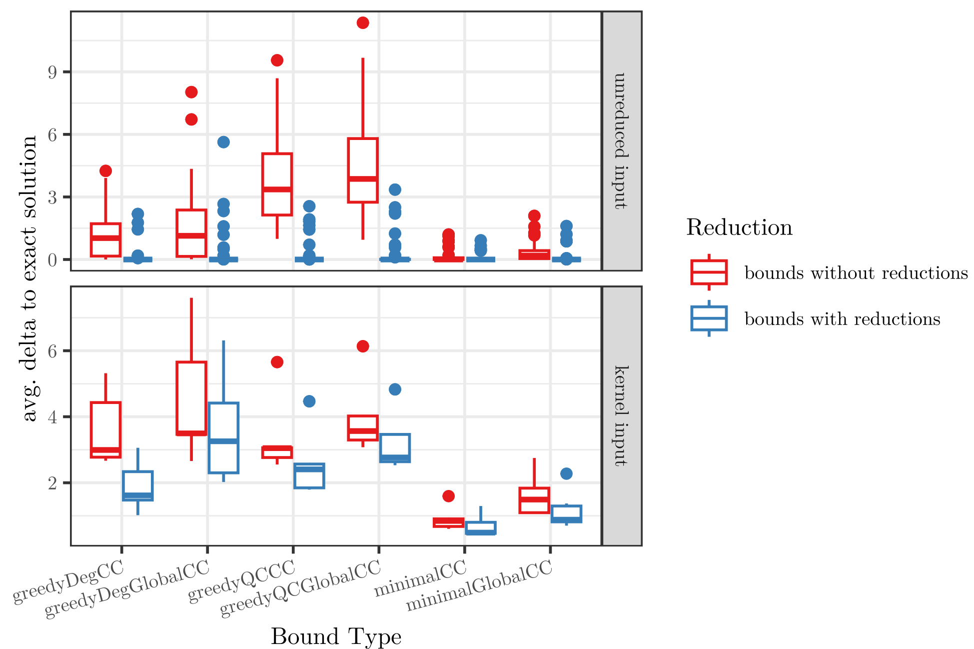


(b) Upper Bound.

Figure 4.6: Average delta between bound and exact solution on *small girgs* instances. Whether bounds with reductions or without reductions are used is shown from left to right and whether an unreduced or kernel input was used is shown from top to bottom. The bound algorithm is shown by the shape and colour. The x-axis shows the average exact solution every time a bound is calculated, while the y-axis shows the average delta between the bound and the exact solution.

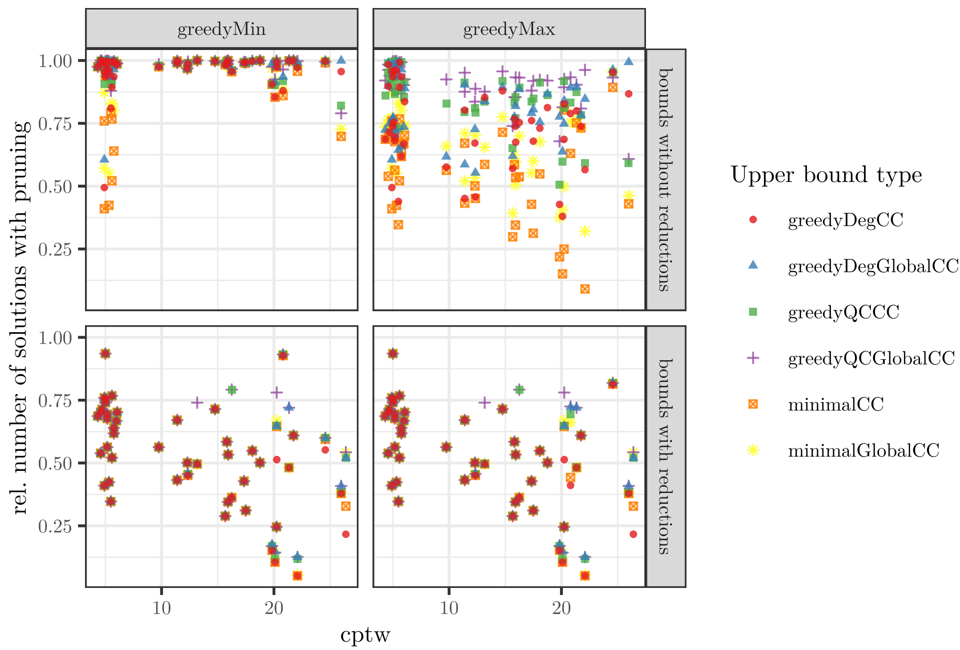


(a) Lower Bound.

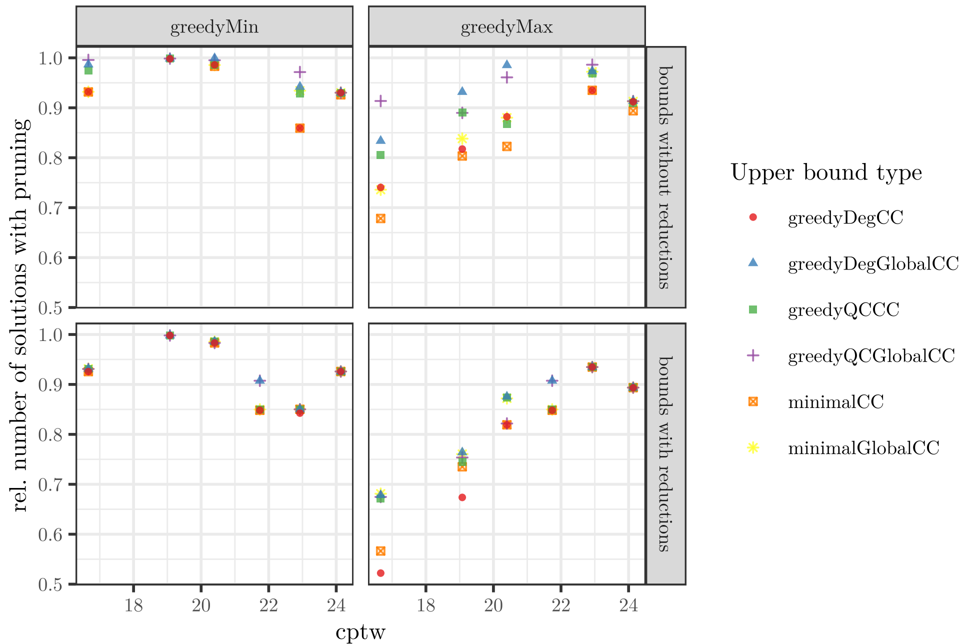


(b) Upper Bound.

Figure 4.7: Average delta between bound and exact solution on *small girgs* instances. Whether bounds with reductions are used is shown by the color and whether the input was unreduced or a kernel is shown from top to bottom. The x-axis shows the bound algorithm, the y-axis shows the average delta between the bound and the exact solution.

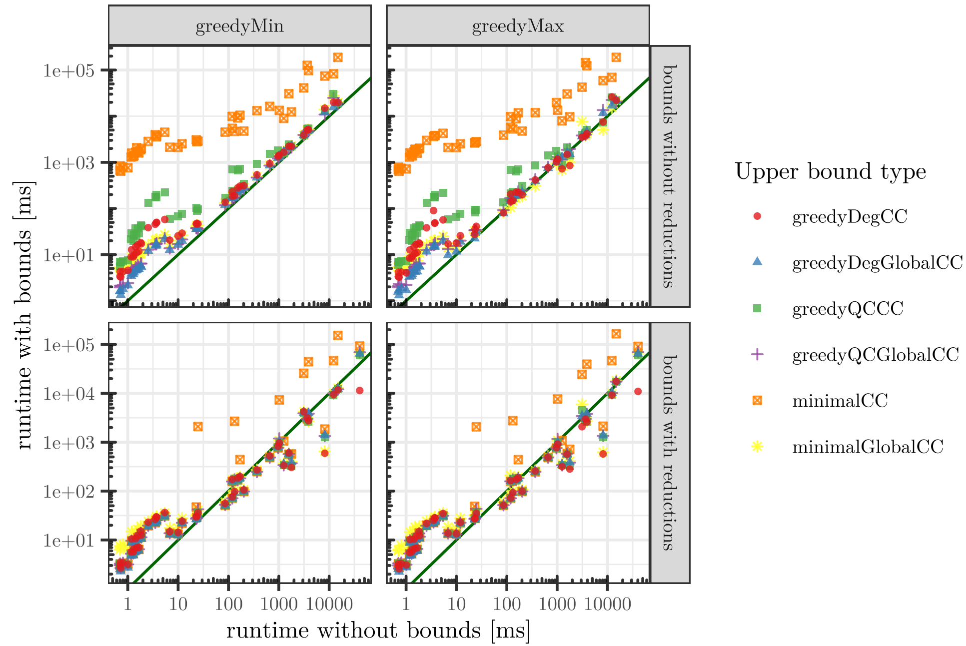


(a) Running the algorithm on unreduced girgs.

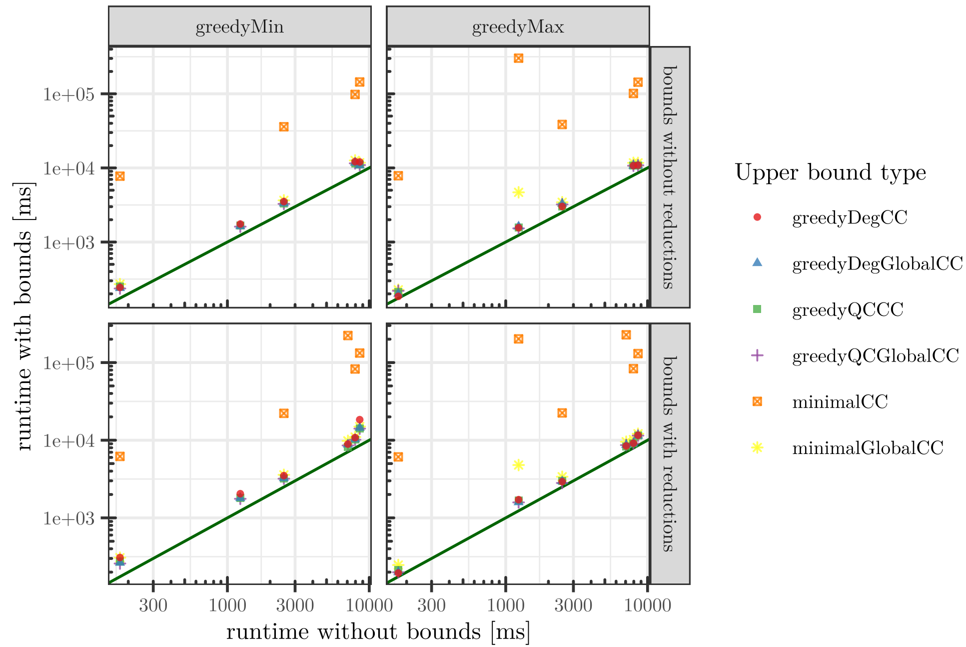


(b) Running the algorithm on the kernel of girgs.

Figure 4.8: Relative number of solutions calculated with pruning on *small girgs* instances. The lower bound is shown from left to right while the upper bound is shown in the shape and colour. Whether bounds with reductions or without reductions are used is shown from top to bottom. The x-axis shows the cp-treewidth of the graph, the y-axis shows the number of solutions with pruning relative to the number of solutions without pruning.

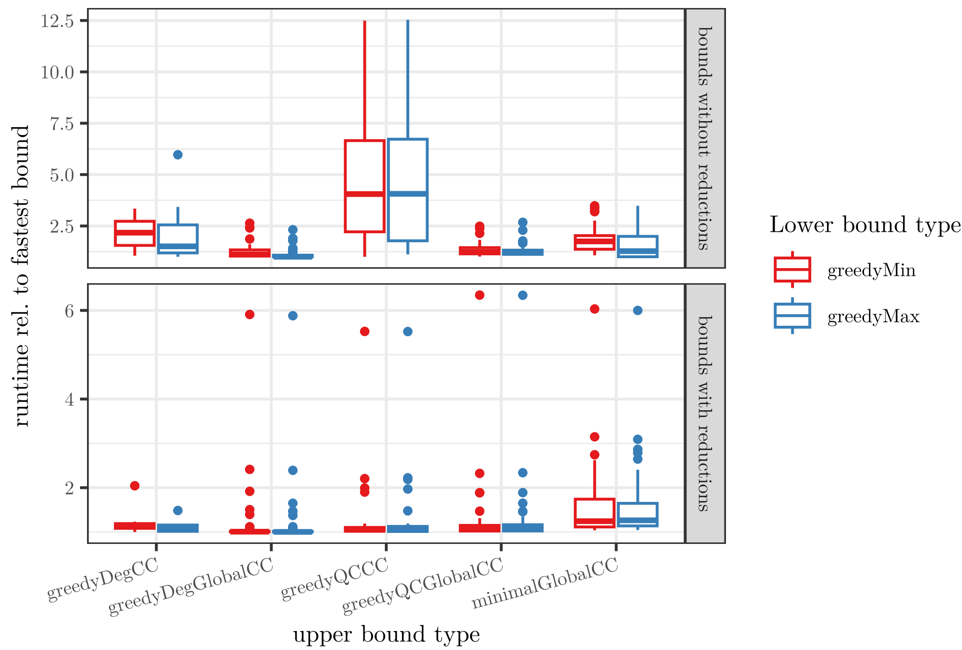


(a) Running the algorithm on unreduced girgs.

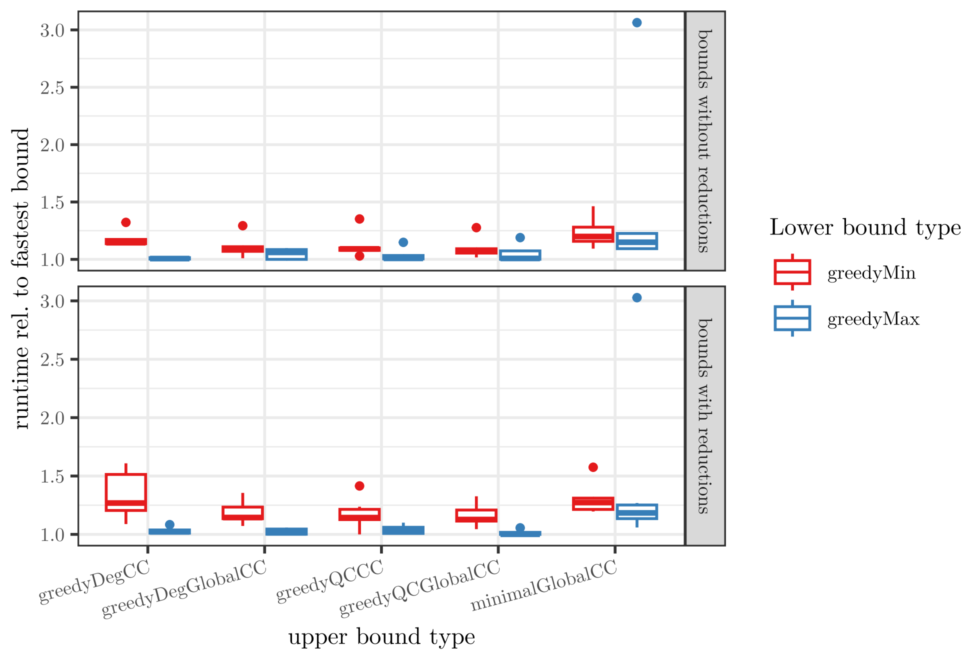


(b) Running the algorithm on the kernel of girgs.

Figure 4.9: Runtime of the algorithm with pruning relative to the runtime without on *small girgs* instances. The lower bound is shown from left to right while the upper bound is shown in the shape and colour. Whether bounds with reductions or without reductions are used is shown from top to bottom. The x-axis shows the runtime with no pruning, the y-axis shows the runtime with pruning. The green line indicates where the runtime of the algorithm with and without pruning match. Note the logarithmic scale on both axes.



(a) Running the algorithm on unreduced girgs.



(b) Running the algorithm on the kernel of girgs.

Figure 4.10: Runtime of the algorithm with the specified upper and lower bound relative to the minimum runtime of all bounds on *small girgs* instances. The upper bound is shown on the x-axis while the lower bound is shown by the color. The y-axis shows the runtime relative to the minimum runtime of all upper or lower bounds in the respective plot. Whether bounds with reductions are used is shown from top to bottom. Note that the *minimalCC* upper bound was removed, to preserve detail in the faster bounds.

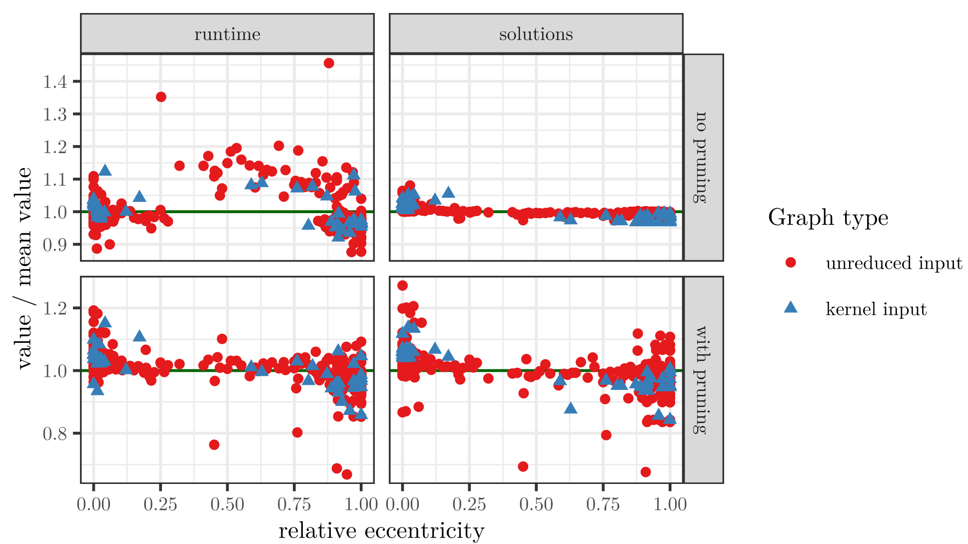


Figure 4.11: Relative runtime and number of solutions for different roots in regards to the relative eccentricity of the root on *small girgs* instances. The x-axis shows the eccentricity relative to the radius and diameter of the root and the y-axis shows the relative runtime on the left and the relative number of solution on the right. The shape indicates whether the input was an unreduced input or a kernel input.

5 Evaluation

In this chapter, we do a final evaluation of the different algorithms proposed throughout this thesis. First, we compare a few configurations of our algorithms with another state of the art solver and evaluate the runtime in regards to the cp-treewidth of the input graph. Then, we analyse the runtime of our algorithm in regards to different graph parameters to determine which can be used to best approximate the runtime.

5.1 Runtime Evaluation

We want to do a final evaluation of our algorithm in comparison to the We Got You Covered [HLSS19] algorithm. For this, we used the final version of our algorithm both with and without pruning. With pruning, we compare the *greedyDegCC* and *minimalGlobalCC* upper bounds, while always using the *greedyMax* lower bound and bounds with reductions. We use the full real-world networks and *girgs* data set as input.

Figure 5.1 shows the relative runtime of these algorithms. We find, as stated previously, that with an unreduced input, our algorithm is outperformed quite significantly by the comparison. In addition, pruning seems to be more effective for inputs of larger cp-treewidth, where our algorithm is not competitive. Looking at the results with a kernel input, we find that pruning is less effective. However, our algorithm without pruning does outperform the comparison for instances with a low cp-treewidth. Thus, we can confirm that we are unable to achieve a speedup by using pruning, while reduction rules yield a significant runtime benefit. Overall, we can say that our algorithm is competitive on instances of low cp-treewidth. Lastly, Table 5.1 shows the number of instances where our algorithm was able to outperform the WGYC solver in regards to which algorithm configuration was used. These include, in particular, four instances which the WGYC algorithm was unable to solve in the time limit of 20 minutes, while our algorithm without pruning was able to solve them in an average of under one minute. These instances have an average size of 1284 vertices and an average cp-treewidth of 19.5. This demonstrates that for some instances our algorithm is significantly superior to a state of the art branch and bound solver. Overall, these results show, that our algorithm is able to outperform the comparison on instances of low cp-treewidth.

5.2 Graph Parameter Evaluation

Now, we want to evaluate whether the cp-treewidth is a good parameter to estimate the computational complexity of our algorithm. For this, we run the algorithm on different real-world networks and girgs while recording multiple parameters. For this evaluation, we compare the treewidth, cp-treewidth and number of vertices with the runtime and attempt to identify any relationships.

The runtime in respect to these parameters is shown in Figure 5.2. We can see that there does not appear to be any meaningful relationship between the graph size and runtime beyond a general trend of longer runtime for larger graphs. The runtime spread for a single graph

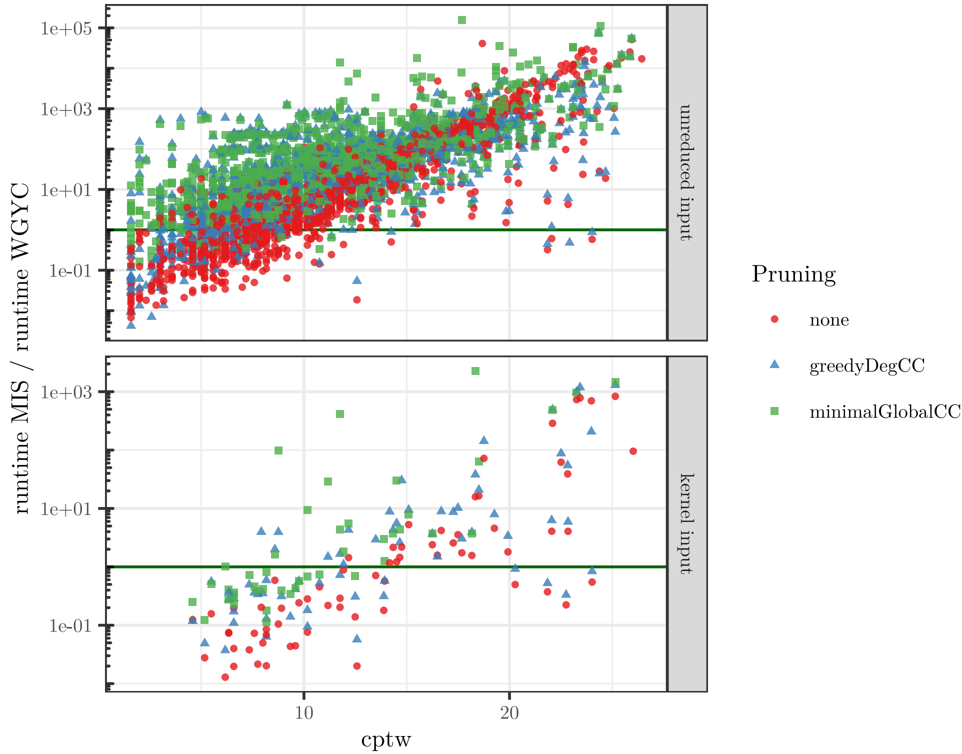


Figure 5.1: Runtime of our algorithm relative to the We Got You Covered algorithm on real-world networks and the *girgs* dataset. The results with unreduced input are shown on top while the results with a kernel input are shown on the bottom. The color and shape indicate whether pruning was used, and if that is the case, which upper bound was used. The x-axis shows the cp-treewidth while the y-axis shows the runtime of our algorithm divided by the runtime of the WGYC solver. Note the logarithmic scale of the y-axis.

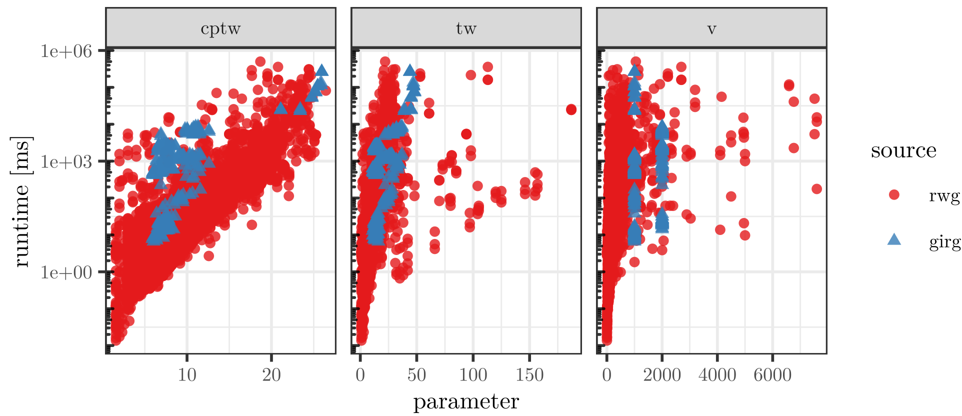
Table 5.1: Number of instances from the real-world networks and *girgs* dataset, which our algorithm solved faster than the We Got You Covered solver. Each instance was evaluated on the unreduced and kernel input and with and without pruning. Additionally, pruning was evaluated with two different upper bounds with bounds with reductions enabled. As a lower bound, *greedyMax* was used. We also give the average and maximum for the number of vertices and cp-treewidth for all those instances.

Upper Bound	#	avg. N	max N	avg. cptw	max cptw	Total
Unreduced input						1600
None	397	162	5000	6.2	24.4	
GreedyDegCC	153	103	1440	5.7	24.4	
MinimalGlobalCC	36	75	201	5.2	11.3	
Kernel input						72
None	43	201	1416	11.3	24.4	
GreedyDegCC	35	186	1416	11.0	24.4	
MinimalGlobalCC	22	45	111	7.9	12.5	

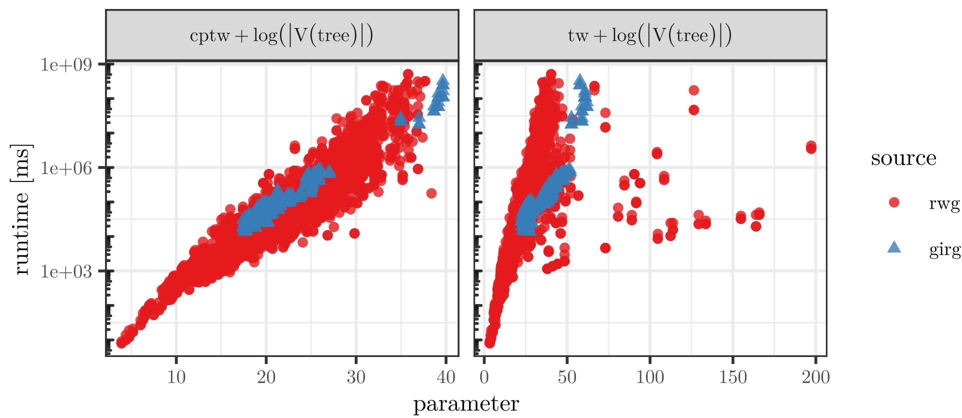
size is rather big and thus, the number of vertices by itself does not appear to be a good measure for runtime. On the other hand, the treewidth does seem to show an exponential relationship with the runtime for the majority of instances. This exponential relationship appears as a linear one in the graph, as the y-axis is logarithmic. However, especially for larger treewidths, this relationship falls apart, with a wide spread of runtimes for similar treewidths. This indicates there is some other influence to the runtime, which is not covered by the treewidth itself. Lastly, when examining the graph using the cp-treewidth, it shows a similar overall pattern to the treewidth. However, the issue with larger values, apparent for the regular treewidth, is not present here. Though, especially the girgs seem to behave quite differently in comparison to the real-world networks.

From Lemma 2.3 we expect to see an exponential relationship between the cp-treewidth and the runtime. However, this does not appear to be enough to fully parameterize the runtime. This is due to the fact, that this does not take the number of bags in the tree decomposition into account. As such, the cp-treewidth is therefore a better approximation of the maximum runtime per bag. When adding the size of the tree decomposition as a linear component into the parameter and creating a composition in the form of $\{\text{tw}, \text{cptw}\} + \log(|V(\text{tree})|)$ we get the result shown in Figure 5.2b. We find that this parameter with the cp-treewidth is indeed a much closer representation of the runtime. The parameter using the treewidth, on the other hand, seems to diverge for larger values of the parameter. To further investigate this, we consider specifically instances with a high ratio between cp-treewdith and treewidth and a low ratio. This is shown in Figure 5.2c, where the colour and shape represent whether the instance has a low or high ratio between the cp-treewidth and treewidth. Here, we can see that the parameter using the cp-treewidth has a linear relationship for both low and high ratios which are compatible with each other. Additionally, for a high ratio between the cp-treewidth and treewidth, the parameter using the treewidth mirrors this relationship. This is expected, as a high ratio between these two parameters indicates that they are quite similar for that instance. However, looking at the results for a low ratio between these two parameters, we find that the parameter doesn't show any apparent relationship to the runtime. Thus, the combined metric using the regular treewidth has some inherent flaw and isn't able to consistently model the runtime for arbitrary graphs. On the other hand, the combined metric using the cp-treewidth is a much better representation for the runtime.

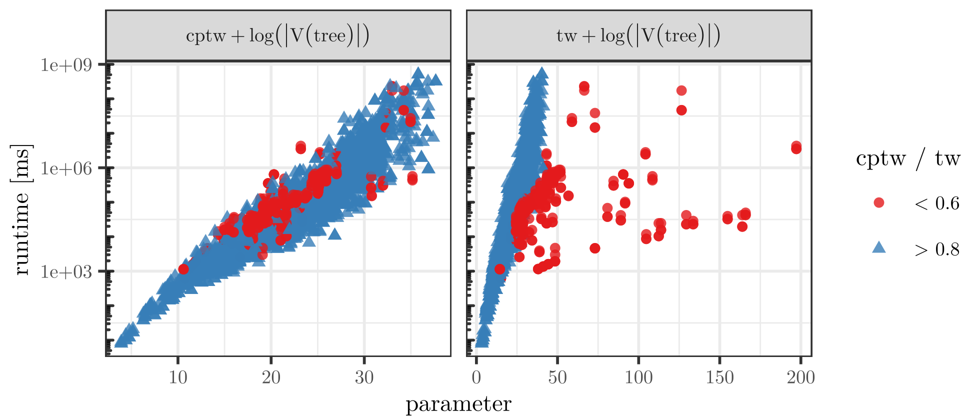
In conclusion, we can see that describing the running time dependency as exponential in the cp-treewidth is less accurate than exponential in the cp-treewidth and linear in the size of the tree decomposition. Additionally, the treewidth is not a good approximation for our runtime, because it falls apart for instances with a low ratio between the cp-treewidth and treewidth.



(a) Using simple graph parameters as comparison.



(b) Using a combined metric as comparison.



(c) Combined metric, differentiated between instances with high/low ratio of cptw and tw.

Figure 5.2: Runtime of our algorithm in respect to different simple graph parameters (a), a combined metric (b) and results differentiated between instances with a low and high ratio between cptw and tw (c). The real-world networks, as well as the *girgs* dataset are used for the input. The runtime is shown on the y-axis in logarithmic scale. The x-axis shows the graph graph parameter with different ones from left to right. For (a) and (b) the color and shape indicate whether the graph is a girg or real-world network. For (c) the color and shape indicate whether the ratio between cptw and tw is low or high.

6 Conclusion

In this thesis, we have developed and optimized a dynamic program on a tree decomposition to solve the Maximum Independent Set problem. We found that there is no advantage in using a clique-partitioned tree decomposition in comparison to a regular tree decomposition. However, the cp-treewidth is a much better parameter to estimate the required runtime. Additionally, we found that this algorithmic technique by itself is not competitive with other state of the art solvers for this problem. However, applying the technique of reduction rules found in these state of the art solvers, we found our algorithm to be competitive on instances of low cp-treewidth. That said, we only focus on the runtime of our algorithm which takes a tree decomposition as input. If we include the runtime to calculate a tree decomposition, our algorithm is significantly slower than the comparison. Thus, this algorithm is only useful, when a tree decomposition is already present anyway. Additionally, our algorithm is only competitive with the application of reduction rules, therefore the tree decomposition must be of the kernel not the original graph. Consequently, this dynamic program is only sensible in some specific cases.

In contrast to the reduction rules, which lead to a significant improvement for our algorithm, the application of upper and lower bounds yielded only a small benefit on some instances. Although theoretically promising, we are unable to benefit from this pruning in practice. Next, we investigated whether the choice of root of the tree decomposition affects the speed of our algorithm. We found that this is indeed the case, however we could not find a specific relation with the eccentricity of the chosen root. In addition, we found that the cp-treewidth is a better parameter for the runtime of our algorithm than the regular treewidth. However, both of these parameters are not sufficient to estimate the runtime correctly. To achieve this, we need to consider another parameter, the size of the tree decomposition. Together these two parameters can be used to estimate the runtime roughly relative to $2^{\text{cptw}} \cdot |V(\text{tree})|$.

6.1 Future Work

In this thesis, we have explored the implementation of a dynamic program on a tree decomposition with a focus on clique-partitioned tree decompositions to solve the Maximum Independent Set problem. Our algorithm has shown promise to outperform other state of the art solvers for instances with a low cp-treewidth. Further research into the optimization of this algorithm could improve it's performance. One possible area of optimization is the application of upper/lower bound pruning. We found, that pruning is indeed theoretically quite effective at reducing the search space. However, in our experiments the additional runtime of the chosen bounds together with their inaccuracy has lead to worse performance than without pruning. Further research could investigate whether this is an inherent limit of this approach or whether other bounds could be used to achieve a speedup for this algorithm.

Nevertheless, the overall success of our algorithm indicates that this could be a viable strategy for other graph problems which might take advantage of the structure of a tree decomposition. Additionally, other parameters and other graph decompositions could be used in a similar manner.

Bibliography

[AI16] Takuya Akiba and Yoichi Iwata. “Branch-and-Reduce Exponential/FPT Algorithms in Practice: A Case Study of Vertex Cover”. In: *Theoretical Computer Science* Volume 609 (Jan. 4, 2016), pp. 211–225. ISSN: 0304-3975. DOI: *10.1016/j.tcs.2015.09.023*.

[AMW17] Michael Abseher, Nysret Musliu, and Stefan Woltran. “Htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond”. In: *Integration of AI and OR Techniques in Constraint Programming*. Edited by Domenico Salvagnin and Michele Lombardi. Cham: Springer International Publishing, 2017, pp. 376–386. ISBN: 978-3-319-59776-8. DOI: *10.1007/978-3-319-59776-8_30*.

[AN02] Jochen Alber and Rolf Niedermeier. “Improved Tree Decomposition Based Algorithms for Domination-like Problems”. In: *LATIN 2002: Theoretical Informatics*. Edited by Sergio Rajsbaum. Berlin, Heidelberg: Springer, 2002, pp. 613–627. ISBN: 978-3-540-45995-8. DOI: *10.1007/3-540-45995-2_52*.

[AP84] S Arnborg and A Proskurowski. “Linear Time Algorithms for NP-hard Problems on Graphs Embedded in k-Trees”. 1984. URL: *https://cds.cern.ch/record/150911* (visited on 02/08/2024).

[Aro21] Chris Aronis. “The Algorithmic Complexity of Tree-Clique Width”. Nov. 3, 2021. arXiv: *2111.02200*. URL: *http://arxiv.org/abs/2111.02200* (visited on 02/08/2024).

[BBL13] Hans L. Bodlaender, Paul Bonsma, and Daniel Lokshtanov. “The Fine Details of Fast Dynamic Programming over Tree Decompositions”. In: *Parameterized and Exact Computation*. Edited by Gregory Gutin and Stefan Szeider. Cham: Springer International Publishing, 2013, pp. 41–53. ISBN: 978-3-319-03898-8. DOI: *10.1007/978-3-319-03898-8_5*.

[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 (Aug. 1, 2015), pp. 86–111. ISSN: 0890-5401. DOI: *10.1016/j.ic.2014.12.008*.

[Bel54] Richard Bellman. “The Theory of Dynamic Programming”. In: *Bulletin of the American Mathematical Society* Volume 60 (1954), pp. 503–515. ISSN: 1936-881X. DOI: *10.1090/S0002-9904-1954-09848-8*.

[BF22] Thomas Bläsius and Philipp Fischbeck. “On the External Validity of Average-Case Analyses of Graph Algorithms”. In: *30th Annual European Symposium on Algorithms (ESA 2022)*. Edited by Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman. Vol. 244. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 21:1–21:14. ISBN: 978-3-95977-247-1. DOI: *10.4230/LIPIcs.ESA.2022.21*.

[BFSW22] Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. "An Efficient Branch-and-Bound Solver for Hitting Set". In: *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2022, pp. 209–220. DOI: *10.1137/1.9781611977042.17*.

[BKW23] Thomas Bläsius, Maximilian Katzmann, and Marcus Wilhelm. "Partitioning the Bags of a Tree Decomposition Into Cliques". Feb. 17, 2023. arXiv: *2302.08870*. URL: *http://arxiv.org/abs/2302.08870* (visited on 10/17/2023).

[Blä+19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. "Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs". In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Edited by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Vol. 144. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 21:1–21:14. ISBN: 978-3-95977-124-5. DOI: *10.4230/LIPIcs.ESA.2019.21*.

[Bod88] Hans L. Bodlaender. "Dynamic Programming on Graphs with Bounded Treewidth". In: *Automata, Languages and Programming*. Edited by Timo Lepistö and Arto Salomaa. Berlin, Heidelberg: Springer, 1988, pp. 105–118. ISBN: 978-3-540-39291-0. DOI: *10.1007/3-540-19488-6_110*.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0-262-03384-4.

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

[Dah14] Mostafa H. Dahshan. "Maximum Independent Set Approximation Based on Bellman-Ford Algorithm". In: *Arabian Journal for Science and Engineering* Volume 39 (Oct. 1, 2014), pp. 7003–7011. ISSN: 2191-4281. DOI: *10.1007/s13369-014-1159-7*.

[dBer+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*. New York, NY, USA: Association for Computing Machinery, June 20, 2018, pp. 574–586. ISBN: 978-1-4503-5559-9. DOI: *10.1145/3188745.3188854*.

[DFH19] M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. "The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration". In: *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. Edited by Bart M. P. Jansen and Jan Arne Telle. Vol. 148. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 25:1–25:23. ISBN: 978-3-95977-129-0. DOI: *10.4230/LIPIcs.IPEC.2019.25*.

[DMŠ23] Clément Dallard, Martin Milanič, and Kenny Štorgel. "Treewidth versus Clique Number. II. Tree-independence Number". Oct. 17, 2023. arXiv: *2111.04543*. URL: *http://arxiv.org/abs/2111.04543* (visited on 03/11/2024).

[ELS13] David Eppstein, Maarten Löffler, and Darren Strash. "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *ACM Journal of Experimental Algorithmics* Volume 18 (Nov. 28, 2013), 3.1:3.1–3.1:3.21. ISSN: 1084-6654. DOI: *10.1145/2543629*.

[ES11] David Eppstein and Darren Strash. "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *Experimental Algorithms*. Edited by Panos M. Pardalos and Steffen Rebennack. Berlin, Heidelberg: Springer, 2011, pp. 364–375. ISBN: 978-3-642-20662-7. DOI: *10.1007/978-3-642-20662-7_31*.

[FGK09] Fedor Fomin, Fabrizio Grandoni, and Dieter Kratsch. "A Measure & Conquer Approach for the Analysis of Exact Algorithms". In: *J. ACM* Volume 56 (Aug. 1, 2009). DOI: *10.1145/1552285.1552286*.

[FHMW21] Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. "DynASP2.5: Dynamic Programming on Tree Decompositions in Action". In: *Algorithms* Volume 14 (Mar. 2021), p. 81. ISSN: 1999-4893. DOI: *10.3390/a14030081*.

[HLSS19] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track". Aug. 20, 2019. arXiv: *1908.06795*. URL: *http://arxiv.org/abs/1908.06795* (visited on 11/30/2023).

[IOY13] Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. "Linear-Time FPT Algorithms via Network Flow". In: *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Dec. 18, 2013, pp. 1749–1761. ISBN: 978-1-61197-338-9. DOI: *10.1137/1.9781611973402.127*.

[Lam+19] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs". In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 144–158. DOI: *10.1137/1.9781611975499.12*.

[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 für Informatik, 2019, 12:1–12:18. ISBN: 978-3-95977-101-6. DOI: *10.4230/LIPIcs.ICDT.2019.12*.

[NT75] G. L. Nemhauser and L. E. Trotter. "Vertex Packings: Structural Properties and Algorithms". In: *Mathematical Programming* Volume 8 (Dec. 1, 1975), pp. 232–248. ISSN: 1436-4646. DOI: *10.1007/BF01580444*.

[RS86] Neil Robertson and P. D Seymour. "Graph Minors. II. Algorithmic Aspects of Tree-Width". In: *Journal of Algorithms* Volume 7 (Sept. 1, 1986), pp. 309–322. ISSN: 0196-6774. DOI: *10.1016/0196-6774(86)90023-4*.

[Tri19] James Trimble. "Peaty". GitHub repository. 2019. URL: *https://github.com/jamestrimble/peaty*.

[XN13] Mingyu Xiao and Hiroshi Nagamochi. "Confining Sets and Avoiding Bottleneck Cases: A Simple Maximum Independent Set Algorithm in Degree-3 Graphs". In: *Theoretical Computer Science* Volume 469 (Jan. 21, 2013), pp. 92–104. ISSN: 0304-3975. DOI: *10.1016/j.tcs.2012.09.022*.